THÈSE

présentée à

l'UNIVERSITÉ DE METZ

pour obtenir le grade de Docteur
de l'Université de Metz

Spécialité : Informatique

Michaël KRAJECKI

Équilibre de charge dynamique : étude et mise en œuvre dans le cadre des applications à nombre Fini de Tâches Indépendantes et Irrégulières.

Soutenue à Metz, le 21 avril 1998

Composition du jury :

Directeur de thèse:	Yvon GARDAN	Professeur à l'Université de Reims
Rapporteurs:	Dider ARQUÈS	Professeur à l'Université de Marne-la-Vallée
	Michel TRÉHEL	Professeur à l'Université de Franche-Comté
Examinateurs:	Zineb HABBAS	Maître de Conférences à l'Université de Metz
	Jean Pierre JUNG	Professeur à l'Université de Metz
	Dominique MÉRY	Professeur à l'Université Henri Poincaré
	Ibrahima SAKHO	Professeur à l'Université de Metz

EN INFORMATIQUE DE METZ

THÈSE

présentée à

l'UNIVERSITÉ DE METZ

pour obtenir le grade de Docteur
de l'Université de Metz

Spécialité : Informatique

Michaël KRAJECKI

Équilibre de charge dynamique : étude et mise en œuvre dans le cadre des applications à nombre Fini de Tâches Indépendantes et Irrégulières.

Soutenue à Metz, le 21 avril 1998

Composition du jury :

Directeur de thèse:	Yvon GARDAN	Professeur à l'Université de Reims
Rapporteurs:	Dider ARQUÈS	Professeur à l'Université de Marne-la-Vallée
	Michel TRÉHEL	Professeur à l'Université de Franche-Comté
Examinateurs:	Zineb HABBAS	Maître de Conférences à l'Université de Metz
	Jean Pierre JUNG	Professeur à l'Université de Metz
	Dominique MÉRY	Professeur à l'Université Henri Poincaré
	Ibrahima SAKHO	Professeur à l'Université de Metz

LABORATOIRE DE RECHERCHE EN INFORMATIQUE DE METZ

Remerciements

Je tiens à exprimer mes remerciements et toute ma gratitude à Zineb HABBAS, Maître de Conférences à l'IUT de Metz et à Francine HERRMANN, Maître de Conférences à l'Université de Metz, pour avoir accepté d'encadrer cette thèse, pour leurs nombreux conseils, toujours avisés, et pour la confiance qu'elles m'ont manifesté.

Je remercie Yvon GARDAN, Professeur à l'Université de Reims, qui a dirigé cette thèse et qui m'a permis par ses remarques d'améliorer la rédaction de ce document.

Je remercie aussi mes rapporteurs Michel TRÉHEL, Professeur à l'Université de Franche-Comté et Didier ARQUÈS, Professeur à l'Université de Marne-la-Vallée, pour avoir rapporté mes travaux, ainsi que les autres membres du jury : Dominique MÉRY, Professeur à l'université Nancy I, Ibrahima SAKHO Professeur à l'université de Metz et le directeur du LRIM, Jean Pierre JUNG, Professeur à l'université de Metz.

Merci à Daniel SINGER, Maître de Conférences à l'Université de Metz, qui m'a encouragé et m'a apporté son soutien dès l'instant où j'ai voulu suivre un DEA en informatique.

Je remercie Mohamed NADIF, Maître de Conférences à l'IUT de Metz dont l'aide m'a été précieuse pour mener à bien l'étude statistique de ce travail et pour m'avoir aidé à préparer mes premiers cours à l'IUT.

Je remercie Pierre-Paul MÉREL pour son amitié, et l'affiche du dernier film de la série des *James Bond* qui a apporté un peu de couleur à notre bureau.

Merci aussi à Yann LANUEL, Maître de Conférences à l'Université de Metz, pour son aide concernant le développement de l'application du lancer de rayons.

Je tiens à remercier tous les membres du LRIM et plus particulièrement Cathy, Denise, Denis, Dominique, Jocelyne, Martine, Robin et Stéphane avec qui j'ai partagé de nombreuses pauses-café.

Mes derniers remerciements vont à ma famille, mes parents et Valérie, car sans leur constant soutien et leurs encouragements, je n'aurais pas pu mener à bien cette thèse.

Table des matières

Introduction

Nous pouvons constater que depuis les années 50, la puissance des ordinateurs double environ tous les 18 mois, cette loi empirique est connue sous le nom de la *loi de Moore*. Par conséquent, les différentes communautés scientifiques ont la possibilité de résoudre des problèmes plus rapidement et plus précisément grâce à l'informatique.

Cependant, malgré les progrès réalisés par les machines séquentielles, elles restent encore trop « lentes » pour des applications scientifiques particulières. Aujourd'hui, nous pouvons penser que nous ne sommes pas loin de la limite de puissance des processeurs car les problèmes d'intégration des circuits et de refroidissement sont de plus en plus difficiles à résoudre. Ainsi, la seule méthode permettant d'augmenter sensiblement les capacités de traitement d'une machine est de multiplier le nombre de processeurs travaillant en parallèle.

Le parallélisme permet l'utilisation simultanée de plusieurs processeurs pour résoudre plus rapidement un problème. Cependant, multiplier la puissance de la machine par le nombre de processeurs est un idéal qui est très difficile à atteindre. En effet, la différence existante entre la performance théorique de la machine et la performance pratique observée est très souvent importante. Nous pouvons en partie expliquer cette différence par la difficulté de programmation d'une machine parallèle.

Une application parallèle est divisée en plusieurs composants (*tâches*), chacun doit alors être placé sur un des processeurs qui constituent la machine parallèle. Si la répartition des tâches n'est pas optimisée, les performances du programme seront réduites. Le placement de ces tâches est un des problèmes principaux du parallélisme. Pour obtenir une utilisation efficace d'une machine parallèle, il est nécessaire de résoudre efficacement ce problème. Ce placement peut être choisi avant le début de l'exécution (placement *statique*) ou bien au cours de l'exécution, on parle dans ce cas, *d'équilibrage dynamique* des charges.

L'algorithmique parallèle permet l'étude de questions posées lorsqu'on parallélise un algorithme mais les réponses varient en fonction des applications. Dans cette thèse, nous nous intéressons à une famille d'applications dont la charge globale peut être décomposée en un nombre fini de tâches indépendantes pour lesquelles le temps d'exécution est inconnu. La famille d'applications dénommée FTII[1] que nous retenons dans notre étude est formellement caractérisée par le caractère irrégulier de ses tâches qui sont en nombre fini. L'objectif de cette thèse est de fournir pour les algorithmes FTII un environnement d'aide à la parallélisation. Dans cet environnement, nous proposons :

1. Cinq algorithmes MIMD d'équilibre de charge dynamique.

2. Un modèle mathématique basé sur les matrices pour valider les algorithmes théori-

1. algorithme à nombre Fini de Tâches Indépendantes et Irrégulières

quement.

3. Un environnement de programmation sur une machine parallèle virtuelle.

Nous déduisons de cet environnement une méthode de décision de parallélisation efficace d'une application FTII. En d'autres termes, étant données une application FTII et une machine parallèle cible, l'environnement permettra de choisir la stratégie d'équilibre de charge la plus adaptée.

Parmi les applications FTII, nous présentons plusieurs applications de CAO (Conception Assisté par Ordinateur). En effet, les applications CAO sont souvent très lourdes et le parallélisme est un candidat naturel pour améliorer les performances. Pour illustrer cet environnement, nous considérons le lancer de rayons comme étude de cas.

Nous organisons la suite de ce mémoire de la façon suivante :

Le chapitre 1 est une introduction à l'algorithmique parallèle, les modèles parallèles et leurs difficultés sont mis en avant. Ainsi, nous montrons les différences entre les modèles SIMD et MIMD proposés par M. J. Flynn [Fly 66] et le modèle *BSP* qui est plus récent [Val 90b]. Pour conclure ce chapitre, les modèles théoriques P-RAM et D-RAM sont utilisés pour définir la complexité des algorithmes parallèles et nous exprimons les problèmes liés à la sémantique du parallélisme.

Dans le chapitre 2, nous proposons une classification des stratégies d'équilibre de charge dynamique. Nous définissons les principales techniques utilisées : algorithmes centralisés client-serveur, algorithmes distribués de type source et serveur initiative, et les algorithmes semi-distribués. Nous définissons ensuite les propriétés qui caractérisent un algorithme d'équilibre de charge : gestion de l'information, mécanisme de sélection et méthode d'appariement.

Dans le chapitre 3, nous présentons de façon formelle le cadre dans lequel nous menons notre étude. Dans un premier temps, nous proposons deux définitions caractérisant les applications FTII ainsi que la notion de stratégie d'équilibre de charge associée. Ensuite, nous développons un nouveau modèle matriciel permettant la validation des algorithmes d'équilibre de charge pour les applications FTII. Nous introduisons les cinq algorithmes d'équilibre de charge que nous avons développés pour les applications FTII et nous les validons à l'aide du modèle matriciel. Pour conclure ce chapitre, une étude statistique permet de classifier les différents algorithmes.

Le chapitre 4 est consacré à la proposition de l'environnement de parallélisation FTII qui découle de l'étude précédente. Nous adoptons une approche orientée objets qui nous permet d'être à la fois indépendant de l'application FTII ainsi que de la machine parallèle sous-jacente. De plus, grâce à cette approche, la gestion dynamique de la répartition de la charge est transparente à l'utilisateur.

Dans le dernier chapitre, nous présentons l'algorithme du lancer de rayons comme un exemple d'application FTII. Dans un premier temps, le principe de l'algorithme est rappelé et nous présentons ensuite les principales optimisations classiques existantes. La seconde partie du chapitre est consacrée au lancer de rayons parallèle. Bien que des machines parallèles dédiées ont été construites pour réduire les temps de calcul [Luc 90], nous nous intéressons uniquement aux solutions logicielles proposées pour des machines parallèles générales. Les différents algorithmes sont classés en deux grandes catégories : les algorithmes orientés *flot de rayons* et les algorithmes *flot d'objets*. Ces deux approches sont parfois panachées dans un même algorithme, on parle alors d'algorithme mixte. La dernière partie de ce chapitre présente les résultats pratiques que nous avons obtenus.

L'ensemble des algorithmes a été développé sur une machine parallèle (CM-5) et sur un réseau de stations de travail. Ainsi, nous montrons que les résultats expérimentaux observés sont conformes à ceux obtenus à l'aide de l'analyse matricielle.

Enfin, nous concluons sur l'ensemble de ce travail et nous présentons quelques perspectives pour cette étude.

Chapitre 1

Algorithmique parallèle

1 Introduction

De nombreux problèmes algorithmiques nécessitent de plus en plus de ressources pour être résolus. Ainsi la visualisation scientifique, les problèmes de satisfaction de contraintes demandent des moyens de calculs sans cesse croissants afin de résoudre des problèmes toujours plus ambitieux [LYJ 96, Mon 74, MHHS 96]. La taille mémoire nécessaire pour modéliser ces problèmes est, elle aussi, de plus en plus importante. C'est pour répondre à ces attentes que des solutions parallèles sont apparues. Il existe deux grandes branches du parallélisme, une branche parallèle (qui s'appuie sur le modèle PRAM) et une branche distribuée (plus proche du modèle DRAM). Il n'existe pas de frontière très bien définie qui sépare distinctement ces deux approches du parallélisme. Dans chaque cas, différents modèles théoriques plus ou moins proches des machines réelles ont été proposés. Dans ce chapitre nous rappelons les principes de base de l'algorithmique parallèle : le contrôle du parallélisme, les différents modèles, les notions de granularité, d'algorithme efficace et d'algorithme optimal.

2 Les trois dimensions du parallélisme

Dans le monde des machines séquentielles, il existe une distinction très forte entre le modèle d'exécution et le modèle de programmation. Le modèle d'exécution utilisé est la *machine de von Neumann* et les modèles de programmation sont nombreux : algorithmique classique, programmation logique et fonctionnelle, programmation orientée objets... Cette distinction permet l'écriture de programmes efficaces et indépendants de la machine cible. Les modèles pour le parallélisme sont à la fois des modèles d'exécution et de programmation, c'est pourquoi l'absence de cette distinction pose un problème de portabilité des applications [Bou 93].

Plusieurs critères sont habituellement proposés pour classifier les différents modèles de parallélisme. Les principaux modèles sont souvent un compromis entre chacun de ces critères : le mode de contrôle du parallélisme, le mode d'accès à la mémoire et le grain de parallélisme.

2.1 Le contrôle du parallélisme

2.1.1 Les modèles SIMD et MIMD

C'est historiquement la première classification introduite par M. J. Flynn [Fly 66]:

- SIMD pour *Single Instruction Multiple Data*: le contrôle est centralisé sur un seul processeur, les autres processeurs sont synchronisés entre chaque instruction du même algorithme qu'ils exécutent ;
- MIMD pour *Multiple Instruction Multiple Data*: le contrôle est réparti sur tous les processeurs et la synchronisation, lorsqu'elle est nécessaire, est réalisée par communication. Chaque processeur peut exécuter un algorithme différent ;
- SPMD pour *Simple Program Multiple Data*: un seul programme est dupliqué sur l'ensemble des processeurs. Les processeurs sont synchronisés par bloc d'instructions.

2.1.2 Le modèle BSP

Le modèle BSP (*Bulk Synchronous Parallel*) a été proposé en 1990 par L. G. Valiant [Val 90b]. C'est à la fois un modèle d'exécution (comme la machine de von Neumann) et un modèle de programmation pour le parallélisme.

2.1.2.1 Le modèle d'exécution BSP est caractérisé par les propriétés suivantes :

- le système est composé d'un ensemble de processeurs qui exécutent des programmes qui peuvent être différents ;
- le système dispose d'un réseau de communication permettant des communications point à point entre n'importe quelle paire de processeurs ;
- il existe un mécanisme de synchronisation efficace pour un sous ensemble ou pour tous les processeurs.

Les performances obtenues par une machine BSP sont déterminées à l'aide de quatre paramètres qui sont le nombre de processeurs, la puissance de chaque processeur, le coût de synchronisation et celui de communication.

Le nombre de processeurs ainsi que leur puissance ont évidemment une influence directe sur les performances globales du système. Si une étape est définie comme une instruction de calcul de base (souvent une opération en virgule flottante), alors la puissance de chaque processeur peut être mesurée en nombre d'étapes réalisées par seconde.

Les capacités du réseau d'interconnexion jouent également un rôle prépondérant. Ainsi on mesure le coût d'une synchronisation en nombre d'étapes. Cette mesure correspond au nombre d'opérations qu'aurait pu réaliser un processeur pendant le temps nécessaire à la synchronisation. De même, on mesure le coût d'une communication en nombre d'étapes par mot machine. Ainsi, si le coût d'une communication est de 5 étapes par mot machine, et qu'un processeur envoie un message dont la longueur est de 1 mot, il est bloqué pendant 5 étapes.

2.1.2.2 Le modèle de programmation BSP s'appuie sur la notion de super étape (*superstep*). Un programme BSP est constitué d'un certain nombre de super étapes. Entre chaque étape, il y a synchronisation des processeurs et échange de messages. Une super étape est composée de plusieurs étapes qui s'appliquent à des données locales. Les communications sont autorisées et non bloquantes. La terminaison de chaque communication intervient au plus tard à la barrière de synchronisation qui termine la super étape courante.

L'approche BSP permet d'écrire un programme indépendant de la machine cible. Cette application pourra être exécutée sur un réseau de stations de travail comme sur une machine parallèle, par exemple une *Origin 2000* de *Silicon Graphics*. Une présentation plus détaillée du modèle BSP peut être trouvée dans [McC 95, Val 90a].

2.2 Le type de mémoire et son accès

Ce critère spécifie les caractéristiques de la mémoire et du système de communication entre les processeurs. Ces caractéristiques sont essentielles dans la définition d'une mesure de complexité et d'efficacité d'un algorithme parallèle.

2.2.1 Le modèle PRAM

Une mémoire peut être *partagée* par l'ensemble des processeurs. Les processeurs communiquent par l'intermédiaire de cette mémoire. Le modèle associé est nommé PRAM pour *Parallel Random Access Memory* [AL 92].

2.2.1.1 L'accès multiple à la mémoire est régi par un ensemble de règles résolvant les conflits éventuels d'accès en lecture ou en écriture. Par exemple l'accès CREW (*Concurrent Read Exclusive Write*) autorise l'accès concurrent en lecture mais seulement l'accès exclusif en écriture. L'accès CRCW (*Concurrent Read Concurrent Write*) autorise l'accès concurrent en lecture et en écriture. Mais dans ce cas, il faut préciser la règle de résolution des conflits choisie (maximum des valeurs, somme des valeurs, etc.).

Exemple 1.1 : recherche du minimum avec une règle CRCW.

On recherche le minimum de n valeurs placées initialement dans un tableau $T[1...n]$ en mémoire partagée. On utilise aussi un tableau auxiliaire $Max[1...n]$ partagé. L'algorithme 1.1 est appliqué afin de déterminer le minimum.

Algorithme 1.1 *Minimum*

pour tous les i tels que $1 \leq i \leq n$ **faire en parallèle** /* première étape */
    Max[i] ⟵ 0
finpour
pour tous les couples (i, j) tels que $1 \leq i, j \leq n$ **faire en parallèle** /* deuxième étape */
    si T[i] < T[j] **alors**
        Max[j] ⟵ 1
    finsi
finpour
pour tous les couples (i, j) tels que $1 \leq i, j \leq n$ **faire en parallèle** /* troisième étape */
    si Max[i] = 0 et Max[j] = 0 et $i < j$ **alors**
        Max[j] ⟵ 1

```
        finsi
finpour
pour tous les i tels que 1 ≤ i ≤ n faire en parallèle          /* quatrième étape */
        si Max[i] = 0 alors
            Min ⟵ T[i]
        finsi
finpour
```

Si on dispose de n^2 processeurs et que l'on associe à chacun un couple (i, j) alors il se peut que dans l'étape 2 plusieurs processeurs tentent d'écrire au même instant dans $Max[j]$. Il faut donc définir une règle de résolution de conflits d'écriture : dans ce cas, tous les processeurs veulent écrire la même valeur 1. Comme exemple de règle pour l'accès concurrent en écriture, on peut prendre le sup des valeurs.

Après l'étape 2, on obtient la propriété $Max[i] = 0$ si et seulement si $T[i]$ est un minimum. On peut donc rajouter les étapes suivantes pour obtenir ce minimum dans la variable Min en mémoire partagée. L'étape 3 contient aussi une écriture qui peut être concurrente. Avec la règle du sup, on obtiendra $Max[i] = 0$ seulement pour la valeur de i tel que $T[i]$ et i sont minimum.

Le PRAM est le modèle théorique le plus répandu. La relative simplicité d'expression des algorithmes dans le modèle PRAM ainsi que les travaux sur les possibilités de simulation de ce modèle sur une architecture réelle à mémoire distribuée montrent l'intérêt d'écrire des algorithmes en PRAM. Cependant du point de vue matériel, il est très difficile, voire impossible de réaliser physiquement une mémoire partagée accessible en temps constant par un grand nombre de processeurs.

2.2.1.2 Le concept de mémoire virtuelle partagée.

Le concept MVP a été proposé par K. Li en 1986 [Li 86]. Ce concept permet de mettre en œuvre le modèle PRAM sur une machine à mémoire distribuée. L'espace d'adressage mémoire est unique. La mémoire est structurée en un ensemble de *pages*. Chaque processeur possède son propre ensemble de pages et peut accéder à l'ensemble des pages par communication. Une partie de la mémoire est réservée par un système de cache. Ainsi, les pages qui ne sont pas locales, sont stockées dans le cache. Au niveau de chaque processeur, un mécanisme de gestion des pages est utilisé. Quand un utilisateur veut accéder à une page, le mécanisme de gestion vérifie si cette page est présente en mémoire locale ou non. Si la page est présente, aucune communication n'est nécessaire et la page est délivrée. Par contre si elle est absente, une requête est envoyée en direction du processeur propriétaire de cette page. Le système peut choisir par la suite d'éliminer une page contenue dans son cache, soit parce qu'il manque de place, soit pour des raisons de cohérence des données.

Ce concept permet donc à l'utilisateur d'écrire son application sans se soucier de la localité des données traitées. Il est très efficace quand le problème à résoudre comporte des données cohérentes. Mais, si un processeur doit accéder à un grand nombre de pages de façon aléatoire, le nombre de messages émis risque de saturer le réseau de communication.

2.2.2 Le modèle DRAM

Une mémoire peut également être *distribuée* sur tous les processeurs qui communiquent par l'envoi de messages grâce à un algorithme de routage réalisant un certain protocole de communication qui dépendra du type de réseau d'interconnexion entre les processeurs. Le modèle associé est nommé DRAM pour Distributed Random Access Memory.

2.2.2.1 Le modèle DRAM prend en compte les caractéristiques physiques du réseau d'interconnexion des processeurs. Les algorithmes décrits en DRAM ont une structure qui dépend des protocoles de communication utilisés, de l'implantation matérielle de la machine et de la topologie du réseau (grille, hypercube). Leur mise en œuvre sur la machine réelle cible pour laquelle ils ont été écrits est immédiate. Par contre, l'expression des communications complique leur conception et toute modification matérielle remet en cause l'algorithme.

2.2.2.2 Le réseau d'interconnexion permettant aux processeurs de communiquer peut être vu comme un graphe à p sommets (où p est le nombre de processeurs) [McC 93]. Chaque processeur peut envoyer un message directement à tout sommet adjacent (processeur) dans le graphe en un temps constant. Chaque arête du graphe peut transmettre un message unitaire dans un temps unitaire et possède une queue pour stocker les messages qu'elle doit transmettre.

Un réseau d'interconnexion est caractérisé par deux paramètres :

le degré est défini par le nombre maximum d'arêtes issues d'un sommet ;

le diamètre est la distance maximale (le nombre d'arêtes minimum) séparant deux sommets.

Quand le degré du réseau d'interconnexion est faible, chaque processeur peut gérer efficacement ses messages. Par contre, quand le diamètre du réseau augmente, le temps de communication entre deux processeurs peut être très variable. En effet, le temps de communication entre deux sommets quelconques est proportionnel à la distance qui les sépare et cette distance est toujours majorée par le diamètre. À l'inverse, si le degré est plus important, la gestion des messages au niveau des différents processeurs sera moins efficace, mais le diamètre du réseau sera réduit.

Nous résumons dans les paragraphes suivants les principales topologies utilisées par les machines parallèles.

L'anneau. C'est la topologie la plus simple 1.1. Chaque processeur possède exactement deux voisins (à gauche et à droite). Le degré de l'anneau est 2 et son diamètre est égal à $p/2$.

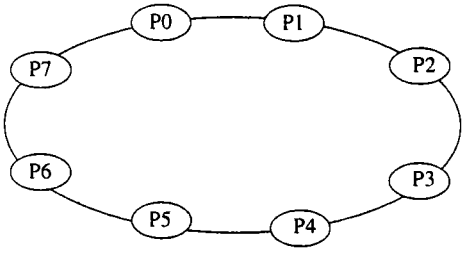


FIG. 1.1 –*réseau d'interconnexion en anneau pour 8 processeurs*

La grille torique. Les processeurs sont disposés sur une grille régulière (voir figure 1.2. Chacun d'entre eux possède quatre liens de communication en direction des processeurs nord, sud, est, ouest. Les processeurs situés sur les bords sont reliés aux processeurs situés sur les bords opposés. Le degré de la grille torique est égal à 4 et son diamètre est en $O\left(\sqrt{p}\right)$.

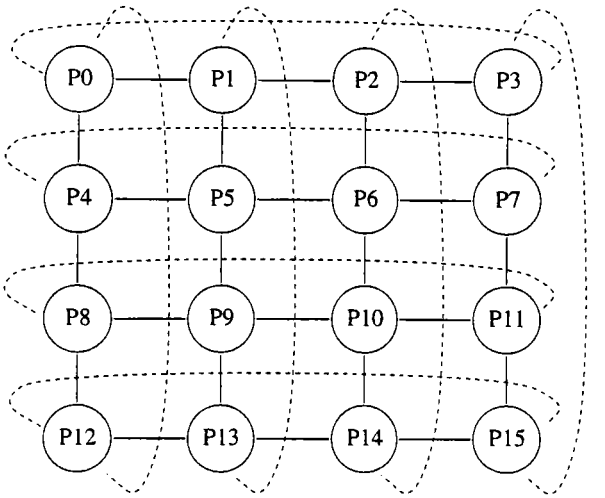


FIG. 1.2 –*Machine à 16 processeurs utilisant une grille torique*

Le fat tree est un arbre binaire ou 4-aire, voir figure 1.3. La bande passante augmente au fur et à mesure que l'on s'approche de la racine [Lei 85]. Cette topologie assure qu'il n'y aura pas de dégradation des performances et que le système est facilement extensible. Le degré du *fat tree* est égal à 4 et son diamètre est en $O(\log p)$. La fat tree est l'architecture utilisée par la CM-5 (*Connection Machine* 5 de *Thinking Machine*).

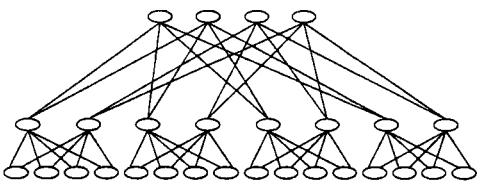


FIG. 1.3 –*organisation en fat tree pour 16 processeurs*

L'hypercube de dimension q est composé de 2^q processeurs numérotés de 0 à 2^{q-1}. Deux processeurs sont adjacents si et seulement si leur décomposition binaire ne diffère que d'un seul bit (voir figure 1.4). Le degré et le diamètre de l'hypercube sont tous les deux égaux à $\log p$. L'hypercube est une structure facilement extensible, ainsi pour doubler le nombre de processeurs, il suffit d'augmenter le degré de 1. De plus, il existe plusieurs chemins pour acheminer un message en direction d'un processeur donné, il est donc possible de prévoir une gestion des pannes. D. Nassimi and S. Sahni ont proposé plusieurs algorithmes spécifiques pour l'hypercube [NS 81].

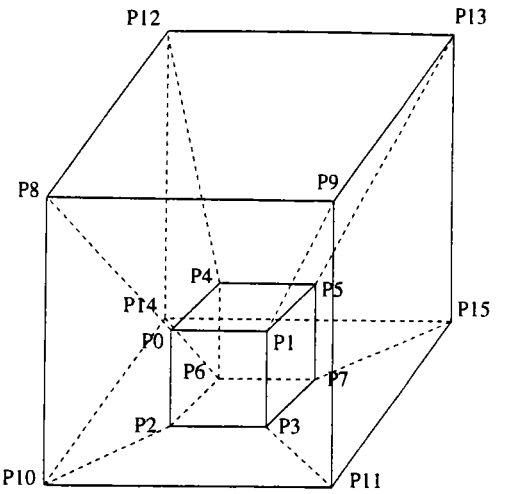


FIG. 1.4 –*hypercube de dimension 4 (16 processeurs)*

Exemple 1.2 : recherche du minimum en DRAM sur l'hypercube.

On recherche le minimum de n valeurs stockées dans un tableau T, avec $n = 2^m$. Le processeur P_i possède dans sa mémoire locale la valeur $T[i]$. L'algorithme 1.2 est appliqué.

Algorithme 1.2 *Hypercube*

pour tous les i tels que $0 \leq i \leq n-1$ **faire en parallèle**
  pour $k = 0$ **à** $m - 1$ **faire** /* les m voisins du processeur i */
    j $\longleftarrow$ (i XOR 2^k) /* j est le voisin de i sur la dimension k */
    si T[i] < T[j] **alors**
      T[j] $\Leftarrow$ T[i]
    finsi
  finpour
finpour

Remarques : le symbole $\Leftarrow$ signifie la communication d'une valeur d'un processeur à un processeur voisin ; à la fin de l'algorithme, tous les processeurs i ont la valeur minimum dans leur variable locale $T[i]$.

Nous avons vu que le modèle PRAM permet une expression simple des algorithmes parallèles, mais ce modèle est relativement éloigné des machines parallèles existantes. Par contre, le modèle DRAM permet une représentation plus fine de la machine parallèle, mais l'écriture des algorithmes parallèles est alors beaucoup plus délicate. Ces arguments favorisent la recherche d'un modèle X-RAM, modèle théorique unique incorporant à la fois une mémoire partagée et une mémoire distribuée. Le modèle X-RAM se constitue à la fois en tant que modification du modèle PRAM et du modèle DRAM. En effet, on peut le considérer comme une extension du modèle PRAM avec un graphe d'interconnexion complet, la mémoire est alors dite *fortement couplée*. On peut également considérer le modèle X-RAM comme une extension du modèle DRAM comportant une mémoire partagée.

2.3 La granularité du travail

Définition 1.1 *(granularité du travail)*
On appelle granularité du travail la quantité de travail réalisée par chaque processeur. Soit n_o le nombre d'opérations à réaliser, alors la granularité est $\frac{n_o}{n_p}$ où n_p est le nombre de processeurs.

Plus le grain est *fin*, proche de l'instruction, plus le nombre de processeurs doit être grand. On parle alors de parallélisme massif, le nombre de processeurs est de l'ordre d'un millier et ils fonctionnent en mode SIMD. Plus le grain est *gros*, proche du programme, plus le nombre de processeurs est faible (de l'ordre de quelques dizaines). Le modèle MIMD est appliqué et les processeurs coopèrent par échange de messages.

Il est possible d'exécuter un algorithme parallèle prévu pour n processeurs sur une machine comportant seulement m processeurs ($m < n$). Avant de présenter la démonstration de ce résultat apportée par Brent, nous définissons le temps d'exécution parallèle.

Définition 1.2 *(temps d'exécution parallèle)*
Le temps d'exécution d'une application parallèle est égal au temps d'exécution du processeur qui termine en dernier.

Théorème 1.1 *(Théorème de Brent) [Bre 74]*
Soit A un algorithme ayant un temps de calcul parallèle t_{par} et nécessitant en tout n_o opérations alors A peut être simulé avec n_p processeurs en utilisant un temps de calcul en $O(\frac{n_o}{n_p} + tpar)$.

Preuve : à chaque étape i de l'algorithme A, on simule les $n_o(i)$ opérations qui sont effectuées par les n_p processeurs. Cette simulation prendra au plus un temps $t(i) = \frac{n_o(i)}{n_p} + 1$. En sommant ce temps pour toute étape i de 1 à t_{par}, on obtient t le temps de calcul parallèle sur n_p processeurs :

$$t = \sum_{i=1}^{t_{par}} t(i) = \sum_{i=1}^{t_{par}} \left(\frac{n_o(i)}{n_p} + 1 \right) = \frac{n_o}{n_p} + t_{par} \text{ en sachant que } n_o = \sum_{i=1}^{t_{par}} n_o(i). \qquad \square$$

Remarque : On peut donc écrire des programmes à grain fin, généralement dédiés aux machines SIMD, et les exécuter sur une machine de type MIMD. Pour cela, il faut appliquer une politique de répartition des tâches sur l'ensemble des processeurs physiques. Un processeur exécutera plusieurs tâches et non plus une seule.

Ce théorème permet aussi de faire abstraction de la machine, en écrivant un algorithme parallèle utilisant autant de processeurs que nécessaires (on parle alors de processeurs *virtuels*). Puis, l'implémentation effective de cet algorithme est réalisée à l'aide d'une fonction permettant l'allocation de la charge sur l'ensemble des processeurs physiques.

Exemple 1.3 : recherche du maximum par la méthode de l'arbre binaire équilibré.
On veut chercher le maximum de n valeurs où n est une puissance de 2 ($n = 2^m$). Soit T un tableau d'entiers de dimension $2 \times n$ qui initialement contient les n valeurs dont on recherche

le maximum aux positions : $T[n]...T[2 \times n - 1]$. Pour résoudre ce problème, nous appliquons l'algorithme 1.3.

Algorithme 1.3 *Maximum*

pour k = m − 1 **descendant à** 0 **faire**
  pour tous les j tels que $2^k \leq j \leq 2^{k+1} - 1$ **faire en parallèle**
    T[j] ⟵ Max(T[2 × j], T[2 × j + 1])
  finpour
finpour

TAB. 1.1 –*Exemple d'exécution de l'algorithme 1.3 avec 16 processeurs et* $m = 3$.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initialisation									3	7	8	3	9	2	3	1
$k = 2, 4 \leq j \leq 7$					7	8	9	3								
$k = 1, 2 \leq j \leq 3$			8	9												
$k = 0, j = 1$		9														

À la fin, cet algorithme aura affecté à $T[1]$ le maximum recherché, il utilise $n/2$ processeurs (pour $k = m-1$) et il prendra un temps $m = \log(n)$. Si on ne dispose, à présent, que d'un nombre de processeurs $n_p < n/2$ alors on peut simuler l'algorithme précédent par le moyen suivant :

Algorithme 1.4 *Maximum*

On assigne à chacun des $n_p$processeurs un groupe de n/n_p valeurs.
pour tous les processeurs **faire en parallèle**
  Recherche séquentielle du maximum local à son groupe de valeurs.
finpour
On applique l'algorithme parallèle 1.3 aux n_p valeurs obtenues.

À la fin, cet algorithme aura donc pris un temps $O\left(\frac{n}{n_p} + log(n_p)\right)$. Donc, si on utilise $n_p = \frac{n}{log(n)}$ processeurs, $n_o = n$ comparaisons et un temps parallèle théorique $t_{par} = log(n)$ alors le théorème de Brent donne une complexité temporelle en :

$$O\left(\frac{n}{\frac{n}{log(n)} + log(n)}\right) = O(log(n)).$$

Le théorème de Brent peut donner, dans certains cas, une méthode constructive pour obtenir des algorithmes parallèles optimaux en partant d'un grand nombre de processeurs *virtuels*. Il suffit alors de réduire ce nombre jusqu'à atteindre un nombre réaliste de processeurs.

3 Complexité d'un algorithme parallèle

Dans un système parallèle ou réparti, l'exécution d'un algorithme parallèle demande la mise à disposition d'un certain nombre de ressources (nombre de processeurs, espace mémoire,...). Le but de l'analyse de la complexité est de mesurer les différentes ressources nécessaires.

Le modèle classique RAM, dérivé du modèle de la machine de Turing représente le modèle fondamental des machines séquentielles. C'est à partir de ce modèle que sont définies les différentes classes de complexité en temps et en espace mémoire. Ainsi la classe P représente la classe des problèmes qui sont résolus en un temps polynomial par une machine de Turing déterministe. De même, les problèmes NP sont définis comme des problèmes résolus en un temps polynomial par une machine de Turing non déterministe.

Pour le parallélisme, les mesures de complexité sont différentes suivant le modèle parallèle choisi. Ainsi, pour le modèle PRAM, la notion de complexité en communication n'est pas définie, car chaque processeur peut accéder en un temps constant à toute donnée de la mémoire. Une présentation détaillée de la complexité parallèle pour le modèle PRAM est exposé dans [AFR+ 94, chap. 5] et la complexité des algorithmes distribués est introduite par C. Lavault dans [Lav 95]. Les mesures de complexités des algorithmes parallèles sont donc la complexité en temps, la complexité en communication et, comme pour le modèle RAM, la complexité en espace mémoire.

3.1 Complexité en temps

Le temps est souvent le premier critère considéré pour évaluer les performances d'un algorithme parallèle.

La définition 1.2 qui exprime le temps d'exécution, suppose que cette mesure est effectuée par un observateur extérieur au système car dans le cas d'un système asynchrone, il n'y a pas d'horloge centralisée. Dans le cas où le système est synchrone, le temps d'exécution parallèle est déterminé par le nombre de cycles d'horloge décomptés du début de l'application à sa terminaison.

Définition 1.3 *(complexité en temps)*
La complexité en temps d'une application parallèle est égal au temps d'exécution maximal de l'algorithme pour une exécution quelconque.

3.2 Complexité en communication

Quand les communications sont explicites, il est possible de définir une complexité en communication. Ainsi, cette notion est très importante pour la mesure de complexité du modèle DRAM. Par contre, pour le modèle PRAM, les communications sont implicites. Il n'est donc pas possible d'exprimer une complexité en communication. En effet, l'échange d'informations entre les processeurs s'effectue par l'intermédiaire de la mémoire centralisée. Dans ce modèle, l'écriture ou la lecture d'une information mémoire s'effectue en un cycle d'horloge.

La définition suivante peut être proposée quand les communications sont explicites :

Définition 1.4 *(complexité en communication)*
La complexité en communication d'une application parallèle est égale au nombre maximal

de messages échangés (ou le nombre maximal d'octets échangés) par les processeurs au cours d'une exécution quelconque de l'algorithme.

La complexité en communication est fondamentale dans le cas de l'algorithmique distribuée, car ce sont les communications qui déterminent les performances de l'application parallèle. C'est pourquoi, dans la majorité des cas, il est nécessaire de minorer le nombre de communications. Pour obtenir ce résultat, il est parfois utile de dupliquer le calcul d'une valeur sur une partie des processeurs plutôt que d'attendre qu'un processeur diffuse ce résultat.

3.3 Accélération et efficacité

Grâce à la définition du temps d'exécution parallèle d'une application, nous pouvons introduire les définitions concernant l'accélération et l'efficacité d'une application parallèle.

Définition 1.5 *(accélération)*
On appelle accélération d'un algorithme parallèle A, le rapport entre le temps t_{seq} de l'algorithme séquentiel optimal par le temps $t_{par}(n_p)$ de l'algorithme parallèle fonction du nombre de processeurs n_p, $Acc(A) = \dfrac{t_{seq}}{t_{par}(n_p)}$.

Remarque : l'accélération est maximale lorsqu'elle est égale au nombre de processeurs, c'est à dire lorsque le travail réalisé par l'algorithme séquentiel optimal a été parfaitement réparti sur les processeurs et que cette répartition n'a pas induit de coût supplémentaire. Cependant, nous pouvons remarquer que l'expression de l'accélération est dépendante du nombre de processeurs, c'est pourquoi il est nécessaire de définir une notion *d'efficacité* où de *taux d'activité.*

Définition 1.6 *(efficacité)*
En utilisant les notations introduites précédemment, on définit l'efficacité d'un algorithme parallèle A comme étant le rapport $\dfrac{t_{seq}}{t_{par}(n_p) \times n_p}$.

Remarque : l'efficacité est proche de 1 quand l'accélération est maximale. Si un programme obtient une efficacité de 0,6, cela signifie qu'il n'a réellement exécuté l'algorithme que pendant 60% du temps de l'exécution. Le reste du temps est réparti en communications, synchronisations, équilibre de charge...

4 Preuves pour un algorithme parallèle

La sémantique permet de mieux comprendre un programme, d'énoncer des résultats sur sa correction, sa terminaison, etc...

Si, dans le cas séquentiel, il existe des techniques satisfaisantes permettant de vérifier qu'un programme termine ou répond à sa spécification, en revanche, en parallélisme le

problème de sémantique est beaucoup plus complexe . Pour pouvoir raisonner formellement et écrire des preuves sur des programmes parallèles, il faut se placer dans un domaine mathématique dont les objets sont les modèles du programme.

Il existe différents modèles permettant de définir la sémantique de programmes parallèles : système de transitions ou réseau de Pétri, logique temporelle...

Avant de présenter quelques modèles pour le parallélisme, nous illustrons dans un exemple un problème classique de sémantique qui est celui de la terminaison.

Exemple 1.4 : un algorithme de terminaison.

Lorsqu'un algorithme parallèle est distribué, un processeur seul ne peut pas déterminer s'il a effectivement terminé son travail car il peut toujours recevoir un message et une tâche à effectuer d'un processeur voisin.

C'est pourquoi, on utilise des algorithmes de détection de terminaison, parmi lesquels l'algorithme du jeton proposé par Dijkstra *et al.* [DFvG 83]. Les auteurs supposent l'existence d'un anneau réel ou virtuel reliant entre eux l'ensemble des processeurs qui sont numérotés de 1 à n, et donc une fonction $suivante(i) = i + 1 \ \% \ n$ est parfaitement définie (l'opérateur % représente le reste de la division entière). Chaque processeur i possède trois variables locales :

$CouleurJeton(i)$ = blanc ou noir ;

$Actif(i)$ = vrai ou faux (représente l'état du processeur) ;

$Envoi(i)$ = nombre de messages envoyés à un processeur le précédent dans l'anneau depuis le dernier passage du jeton.

Algorithme 1.5 *Algorithme de terminaison*

```
répéter jusqu'à CouleurJeton(0) = blanc
        CouleurJeton(i) ⟵ blanc
        pour i = 1 à n faire
            si CouleurJeton(i) = noir alors
              CouleurJeton(suivant(i)) ⇐ noir
            sinon
              si ¬Actif(i) ∧ Envoi(i) = 0 alors
                CouleurJeton(suivant(i)) ⇐ blanc
              sinon
                 CouleurJeton(suivant(i)) ⇐ noir
              finsi
              Envoi(i) ⟵ 0
            finsi
        finpour
finrépéter
```

Si, à la fin d'un tour de l'anneau virtuel, le jeton est noir, il reste encore des tâches à exécuter et l'algorithme global n'est pas terminé.

4.1 Réseaux de Pétri

Les réseaux de Pétri sont un modèle de description de systèmes d'événements asynchrones [Pet 77]. Un réseau de Petri est un graphe biparti composé d'un ensemble fini de places (représentées par des cercles), d'un ensemble fini de transitions (représentées par

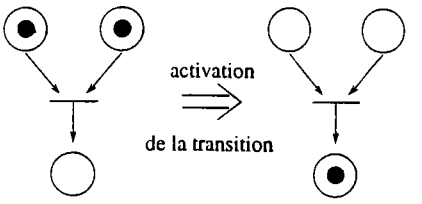


FIG. 1.5 -*activation d'une transition.*

des traits) et d'un nombre fini d'arcs qui relient les places et les transitions. Le graphe ainsi construit est marqué par des jetons contenus dans des places.

Exemple 1.5 : activation d'une transition.

Pour pouvoir activer une transition, il est nécessaire que chaque place située en amont de la transition dispose d'un jeton. Quand la transition est activée, un jeton est produit au niveau de toutes les places en aval de la transition (voir la figure 1.5). Ainsi, la marquage du réseau de Petri évolue et se modifie au gré des transitions activées.

La méthode des réseaux de Petri permet l'expression des mécanismes fondamentaux des systèmes distribués : synchronisation, choix non déterministe, partage de ressource en mutuelle exclusion...

Pour illustrer les réseaux de Petri, nous présentons le cas de l'exclusion mutuelle pour deux processeurs (ou processus). La figure 1.6 présente le réseau de Petri correspondant.

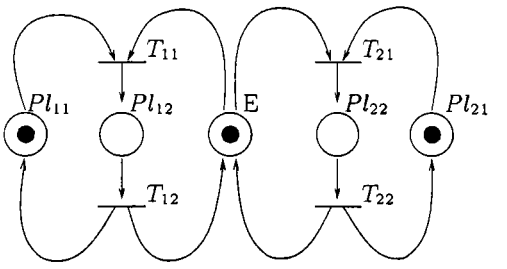


FIG. 1.6 -*exclusion mutuelle pour deux processeurs.*

Quand la place Pl_{i1} contient le jeton, le processeur P_i n'est pas en section critique. De même, quand Pl_{i2} contient un jeton, le processeur P_i est en section critique.

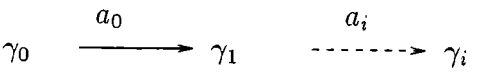
Au départ, les places Pl_{i1} et E contiennent chacune un jeton (marquage initial). Pour que P_i entre en section critique, il doit activer la transition T_{i1}. Si le processeur P_1 est déjà en section critique, il n'y a pas de jeton en E, et P_2 ne peut entrer en section critique. Ce jeton sera disponible en E quand P_1 activera la transition T_{12} pour quitter la section critique. Au départ, les deux processeurs peuvent accéder à la section critique, mais on ne sait pas qui va activer en premier la transition T_{i1} (expression de l'indéterminisme du parallélisme).

Nous pouvons remarquer que cette solution ne garantit pas l'équité, c'est à dire que P_1 aura la possibilité de rentrer en section critique aussi souvent que P_2.

Les réseaux de Petri permettent donc une expression simple du parallélisme et de ses propriétés, mais quand le système à modéliser est complexe, le graphe devient très rapidement important (nombre de places et de transitions) et compliqué. Ainsi le manque de structuration des réseaux de Petri est le principal inconvénient de cette méthode formelle.

4.2 Algèbre de processus

Les algèbres de processus (ou calcul de processus) sont des systèmes formels décrivant de manière abstraite des systèmes de calculs organisés sous forme de collection de composants appelés *processus* ou *agents* qui s'exécutent en parallèle et coopèrent entre eux par communication. Étant donné un ensemble de processus primitifs, des processus plus complexes peuvent être construits de manière hiérarchique par un ensemble d'opérateurs de combinaison de processus. La sémantique d'un processus est exprimé sous la forme d'une trace des actions qu'il a exécutées (voir la figure 1.7).



γ_i : configuration ou état.
a_i : action observéee entre γ_i et γ_{i+1}.

FIG. 1.7 –*trace d'un processus.*

CCS[1] et CSP[2] sont des exemples d'algèbres de processus [Mil 83, Hoa 78]. L'ensemble de ces algèbres est cohérent par rapport aux modèles de calcul tels que le λ-calcul. Il est possible de définir une notion d'équivalence des programmes en comparant les différentes traces (*bissimulation forte* et *faible*).

Ces méthodes ont trouvé de nombreuses applications, parmi lesquels nous pouvons noter la spécification de protocoles de communication ou de systèmes réactifs. Ces méthodes permettent aussi une liaison entre un calcul de processus et une logique modale. Il est donc possible de spécifier, construire puis raffiner un programme grâce à l'algèbre de processus.

4.3 Logique temporelle

La logique temporelle est un cadre rigoureux recouvrant des aspects syntaxiques et sémantiques. L'informatique fournit des outils de démonstration automatique pour les théories logiques. Il est donc possible de proposer une spécification des programmes à l'aide de la logique temporelle. Grâce à cette spécification formelle, il est possible de démontrer des propriétés du programme attendu.

La logique de Hoare, qui s'appuie sur la méthode assertionelle proposée par R. W. Floyd, est une méthode de spécification formelle pour les algorithmes séquentiels qui utilise la logique comme un cadre formel afin d'exprimer rigoureusement des démonstrations [Hoa 69, Flo 67]. De même, une méthode assertionelle pour les systèmes concurrents a été développée par A. U. Shankar [Sha 93].

TLA[3] a été proposé par L. Lamport [Lam 94]. C'est une logique temporelle qui permet de spécifier et de vérifier des systèmes parallèles. TLA permet de démontrer des propriétés d'*invariance* et de *fatalité*. L'invariance permet d'affirmer que quelque chose de mauvais ne peut pas arriver (par exemple un inter-blocage des processeurs), mais elle ne permet pas de garantir que quelque chose va arriver (par exemple la terminaison de l'application

1. *Calculus for Communication Systems*
2. *Communicating Sequential Processes*
3. *Temporal Logic of Actions*

parallèle). Les propriétés de terminaison, de livraison d'une message ou bien d'accessibilité en section critique sont des propriétés du type fatalité. TLA^+ est un langage qui permet la définition et la vérification de formules TLA [Lam 95]. TLA+ comporte un outil informatique qui permet de vérifier de manière automatique des propriétés simples sur des programmes exprimés à l'aide de la syntaxe TLA.

5 Conclusion

Cette brève introduction de l'algorithmique parallèle montre la nécessité de se référer à des modèles théoriques et à des concepts fondamentaux avant de procéder à la conception et à l'implémentation d'une application parallèle. En retour, les principes et les notions théoriques présentés s'enrichissent et s'affinent par une mise en œuvre.

C'est dans cette optique que nous étudions les stratégies d'équilibre de charge et l'algorithme de lancer de rayons parallèle. Notre objectif est de comparer, à la fois sur le plan théorique (à l'aide d'une analyse matricielle) et sur le plan pratique, les algorithmes d'équilibre de charge dynamique dans le cadre d'une famille d'applications que nous caractériserons au chapitre 3.

Dans le chapitre suivant, nous présentons une classification des algorithmes d'équilibre de charge dynamique. En particulier, nous définissons la notion d'algorithme d'équilibre de charge et présentons les différentes approches possibles.

Chapitre 2

L'équilibre de charge : différentes méthodes

1 Introduction

Un des buts majeurs des systèmes répartis et parallèles est de faciliter le partage de ressources. M. Livny et M. Melman ont montré que dans un système distribué, il y a une forte probabilité pour qu'il existe au moins un processeur déchargé [LM 82]. C'est pourquoi, on a recours à des algorithmes d'équilibre de charge afin d'obtenir une efficacité maximale. Nous pouvons définir la notion d'équilibre de charge de la façon suivante :

Définition 2.1 *(équilibre de charge)*
Une stratégie d'équilibre de charge (load balancing) est un algorithme qui permet de maintenir une répartition équitable des tâches à accomplir entre tous les processeurs. Quand le but de l'algorithme est de maintenir tous les processeurs dans un état actif, on parle dans ce cas de partage de charge (load sharing).

En pratique, ce problème a fait l'objet d'un grand nombre de travaux qui ont abouti à de nombreux algorithmes. Les stratégies d'équilibre de charge sont comparées selon différents critères tels que le choix des indicateurs de charge, la méthode d'appariement, le caractère distribué ou centralisé de la stratégie [GS 93]. Cette étude a pour objet de proposer une classification des stratégies d'équilibre de charge dynamique dans le cadre du lancer de rayons parallèle. Cette classification distingue les algorithmes à prise de décision centralisée des algorithmes à prise de décision distribuée [Kra 97].

Un programme parallèle peut être représenté comme un ensemble de tâches qui communiquent. Écrire un programme parallèle demande en premier lieu de placer chacunes des tâches sur l'ensemble des processeurs disponibles. Si cette répartition du travail n'est pas équitable, les performances globales du système seront pénalisées. Le problème de l'équilibre de charge peut être abordé à la compilation de l'application (équilibre de charge *statique*) ou à l'exécution (équilibre de charge *dynamique*) [CK 88]. Après une rapide présentation des stratégies statiques, nous présentons une classification arborescente des stratégies d'équilibre de charge dynamique.

2 Équilibre de charge statique

De nombreuses techniques d'équilibre de charge statique ont été proposées dans le but de répartir l'ensemble du travail à la compilation de l'application. Pour appliquer ces méthodes, il est nécessaire de disposer de plusieurs informations concernant les caractéristiques des tâches avant l'exécution (par exemple, le temps d'exécution de chaque tâche, le nombre de tâches à exécuter, ...). Si n est le nombre de processeurs et t le nombre de tâches alors le nombre de solutions à explorer est n^t. Ce problème est connu comme NP-difficile [GJ 79]. Une approche détaillée de ces méthodes est proposée dans [AFR+ 94, chap. 8]. On distingue généralement les algorithmes exacts des méthodes heuristiques.

2.1 Les algorithmes exacts

Les algorithmes exacts énumèrent toutes les solutions possibles. Le but de la stratégie est de proposer une distribution des tâches qui minimise une fonction de coût qui représente le temps d'exécution de l'application parallèle. Comme ce problème est NP-difficile, il n'existe pas d'algorithme qui calcule la solution exacte en un temps polynomial. C'est pourquoi, ce type de stratégie statique ne peut s'appliquer qu'à des problèmes de taille restreinte.

2.2 Les algorithmes basés sur des heuristiques

Les méthodes heuristiques proposent une solution approchée. Les algorithmes *glouton* construisent une solution de proche en proche en plaçant les tâches une à une sur l'ensemble des processeurs [Lee 91]. Les algorithmes glouton permettent d'obtenir rapidement une solution, mais la qualité de cette solution est très dépendante de l'ordre choisi pour distribuer les tâches.

Les algorithmes itératifs améliorent une solution initiale pour obtenir une répartition correcte de l'ensemble des tâches. La méthode du *recuit simulé* est basée sur l'observation de phénomènes de la physique statistique [Ber 89]. Pour obtenir un métal ayant une structure régulière, il est chauffé et refroidi le plus régulièrement possible. Quand la température est élevée, les molécules se réorganisent et modifient le niveau d'énergie local. La probabilité que ce phénomène se produise diminue avec la température. Quand la température est suffisamment basse, le métal demeure dans un état stable où l'énergie est minimale. Les algorithmes qui appliquent cette méthode pour l'équilibre de charge statique sont présentés généralement de façon intuitive car il est difficile de justifier théoriquement ce type de stratégies. Le recuit simulé fournit de bons résultats mais présente plusieurs inconvénients : c'est un algorithme coûteux dont le résultat est imprévisible. De plus, il est délicat à implémenter car les paramètres qui le caractérisent sont difficiles à déterminer.

La *recherche tabou* construit une solution à partir d'une première répartition des tâches [GL 92]. L'algorithme modifie localement la répartition des tâches afin de l'améliorer. Pour éviter de boucler, une liste des actions déjà effectuées est construite. Grâce à la liste tabou, la méthode évite de faire marche arrière.

3 Équilibre de charge dynamique

Les stratégies d'équilibre de charge statique ne prennent pas en compte le caractère dynamique des applications. Ainsi, quand il n'est pas possible de prévoir la charge engendrée par une application, il est impossible de proposer une répartition statique des tâches. C'est pourquoi il est nécessaire de répartir les charges dynamiquement à l'exécution de l'application.

3.1 Une approche hiérarchique de l'équilibre de charge

Pour obtenir un algorithme parallèle efficace, il est nécessaire de répondre à deux questions primordiales:

1. Comment réduire au maximum les communications?

2. Comment répartir équitablement l'ensemble du travail entre tous les processeurs?

Dans la littérature, de nombreuses méthodes visant à équilibrer la charge entre les différents processeurs ont été proposées. Chacune d'entre elle vise à réduire le nombre total de communications tout en répartissant efficacement les tâches sur l'ensemble des processeurs.

Il est cependant très difficile de comparer les différentes méthodes proposées. En effet, il n'existe pas un modèle d'exécution unique pour le parallélisme, mais plutôt un modèle d'exécution par type de machine [Bou 93]. Ainsi, plusieurs algorithmes d'équilibre de charge dynamique ont été développés pour un modèle d'exécution particulier (MIMD, SIMD ou SPMD). De plus, on distingue généralement les algorithmes MIMD pour machine parallèle des algorithmes MIMD pour systèmes distribués (réseau de stations). En effet, dans le cas d'une machine parallèle, les communications sont peu coûteuses car le réseau d'interconnexion est performant. De plus, on considère généralement que la machine est dédiée à une seule application à la fois. Par contre, dans le cas d'un réseau de stations, les communications sont plus pénalisantes et plusieurs applications s'exécutent en temps partagé. Les hypothèses sont donc très différentes et c'est pourquoi il est très difficile de comparer toutes ces méthodes entre elles. Une première approche pour essayer de comparer les différentes stratégies d'équilibre de charge est de proposer une classification hiérarchique fondée sur le critère suivant:

- Qui est chargé de répartir équitablement la charge entre les différents processeurs?

Cette décision peut être prise de manière centralisée ou distribuée. Dans le cas d'une prise de décision centralisée, on appliquera généralement une stratégie de type client-serveur. Dans ce type de stratégie, un processeur est chargé de répartir équitablement l'ensemble des tâches sur l'ensemble des processeurs. Ainsi, la répartition de la charge est effectuée au niveau global, c'est à dire que tous les processeurs sont concernés par cette opération. Cette décision peut aussi être prise de façon complètement distribuée sur l'initiative de n'importe lequel des processeurs. Il peut en résulter un équilibrage de la charge au niveau local (sur une partie des processeurs) ou au niveau global. Enfin, cette décision peut être prise de façon semi-distribuée, un processeur responsable d'un sous ensemble des processeurs choisit de déclencher une nouvelle phase d'équilibre de charge pour son groupe (voir figure 2.1).

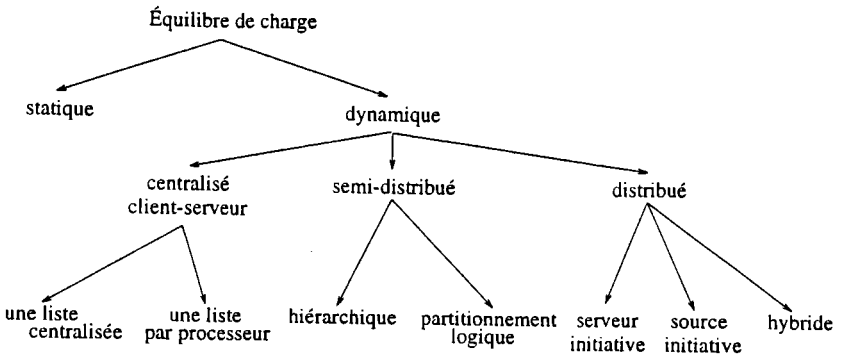


FIG. 2.1 – *Classification des méthodes d'équilibre de charge dynamique*

3.2 Les algorithmes centralisés de type client-serveur

Les algorithmes de type client-serveur sont simples à implémenter et sont, dans un grand nombre de cas, efficaces. C'est pourquoi, ils sont très largement utilisés par la communauté scientifique [Eck 94b]. On distingue deux approches :

1. Chaque processeur dispose d'une file d'attente contenant les tâches qui sont à sa charge ;
2. Un processeur unique possède une file d'attente contenant l'ensemble des tâches du système.

3.2.1 Une file d'attente centralisée pour l'ensemble des processeurs

Quand il est possible de centraliser l'ensemble des tâches à exécuter sur un seul processeur, un algorithme très simple peut être utilisé [Kho 92]:

- le processeur qui dispose de la file d'attente des tâches est le serveur. Il transmet une tâche à un processeur quand celui-ci en fait la demande ;
- les autres processeurs sont tous clients et demandent les tâches à exécuter dès qu'ils sont disponibles ;
- le programme se termine quand il n'y a plus de tâches dans la file d'attente.

Ce type d'algorithme est très performant. En effet, il réalise un bon équilibre de charge entre les processeurs sans aucune hypothèse sur les tâches. Il n'exige aucune information *a priori* sur une tâche (temps d'exécution, ressource mémoire) ce qui est un avantage important. Le coût en communication de cette méthode est réduit.

Proposition 2.1 *Soit nb_t le nombre de tâches de l'application et n le nombre de processeurs. Alors, le nombre total de communications engendrées par l'algorithme d'équilibre de charge est $nb_t \times 2 + (n-1) \times 2$ messages.*

Preuve : en effet, la distribution des nb_t tâches demandent $nb_t \times 2$ messages (une demande d'un client et la réponse du serveur) et la détection de la terminaison par l'ensemble des processeurs demandent $(n-1) \times 2$ messages (chacun des $n-1$ clients demande une tâche et le serveur lui répond que l'application se termine). □

Proposition 2.2 *Si on suppose que la tâche la plus coûteuse a un temps d'exécution de t_{max} secondes, alors la détection de la terminaison de l'algorithme par tous les processeurs aura lieu dans un intervalle de temps inférieur ou égal à t_{max} secondes.*

Preuve : le pire des cas peut être défini à l'aide de deux hypothèses :

- la dernière tâche de la file d'attente est la plus coûteuse, elle sera donc distribuée en dernier ;
- tous les processeurs terminent leur traitement à un même instant, un seul processeur sera à nouveau chargé par le serveur.

Dans ces conditions, tous les processeurs, à l'exception du processeur qui a reçu la dernière tâche, détecteront la fin de l'algorithme. t_{max} secondes plus tard, le dernier client détecte à son tour la fin de l'exécution. □

Du fait de la proposition précédente, il est nécessaire que la granularité soit suffisamment fine pour minorer de façon convenable cet intervalle de temps. Cependant, plus la granularité est fine, plus le coût en communication est important. Il faut donc parvenir à un compromis permettant d'éviter de saturer le réseau de communication tout en conservant une efficacité correcte. Ainsi, le serveur doit pouvoir distribuer une tâche à chaque processeur sans que le premier processeur servi n'est terminé le calcul de la tâche qui lui a été confiée. Si cette hypothèse est vérifiée à chaque instant, les différents processeurs obtiendront rapidement une nouvelle tâche. On peut exprimer cette hypothèse en utilisant les notations précédentes comme suit :
$\forall i \in \{1..nb_t\}, t_i > 2(n-1) \times t_{com}$ où t_i est le temps d'exécution de la tâche i et t_{com} est le temps moyen d'une communication.

Si l'on ne dispose d'aucune connaissance sur le temps d'exécution des tâches, en particulier le temps minimum d'exécution des tâches, il est difficile de déterminer quelle est la granularité à adopter. C'est pourquoi, James W. Kho propose de modifier dynamiquement la granularité [Kho 92]. Ainsi, le serveur essaie de détecter si un de ses clients a attendu longtemps une nouvelle tâche à traiter. Pour atteindre ce but, le serveur mesure le coefficient $k = t_{com}/t_i$. Si ce coefficient est trop important, il peut choisir d'augmenter la granularité du problème. Par conséquent, le nombre de requêtes adressées au serveur sur un intervalle de temps donné va diminuer et il sera alors plus disponible. Au contraire, si le nombre de requêtes reçues par le serveur est trop faible (le coefficient est proche d'une valeur minimale), il peut choisir d'affiner la granularité de son problème afin de réduire le temps d'exécution moyen des tâches. En pratique, il est très difficile de savoir dans quelle mesure il est nécessaire de modifier la granularité du problème. Si la modification de la granularité est trop grossière, le serveur risque de suivre un comportement contradictoire : il commence par affiner le problème, puis décide de rendre la granularité plus grossière, puis l'affine à nouveau et ainsi de suite sans jamais se stabiliser. De plus, c'est à l'utilisateur de déterminer les bornes inférieures et supérieurs acceptables pour le coefficient k. Enfin, cette méthode suppose qu'il est possible de modifier la taille des tâches à traiter et que cette taille a une influence sur le coût en temps de calcul de chacune des tâches.

3.2.2 Une file d'attente par processeur

Certains algorithmes centralisés de type client-serveur s'appuient non pas sur une file d'attente centralisée sur un processeur, mais sur un modèle où chaque processeur gère

sa propre file d'attente des tâches. Chaque processeur initialise sa file d'attente avec la liste des tâches qu'il doit exécuter. Dans ce cas, la résolution du problème de l'équilibre de charge est différente. Le serveur se charge simplement de maintenir à jour un certain nombre d'informations pertinentes (nombre de processus en attente sur chaque processeur, charge mémoire, nombre d'accès disque, ...) sur l'état global du système. Il va ainsi régulièrement mettre à jour ses indicateurs de charge en interrogeant l'ensemble des processeurs.

La méthode proposée dans [PB 90] applique le principe suivant :

- le serveur maintient à jour une file d'attente des processeurs surchargés ;
- quand un processeur P_i devient déchargé, il contacte le serveur ;
- le serveur interroge alors les processeurs de sa file jusqu'à trouver un processeur susceptible de partager sa charge et transmet le numéro de ce processeur à P_i.

3.2.3 Avantages et inconvénients des méthodes centralisées

Les algorithmes centralisés possèdent plusieurs avantages. Leur mise en œuvre est généralement assez simple. La politique de mise à jour des informations est aisée, car elle est réalisée sur un seul processeur. L'efficacité obtenue est bonne, car le surcoût dû à la stratégie reste faible puisque les processeurs clients ne sont pas pénalisés.

Par contre, quand le nombre de processeurs augmente de façon importante, le serveur peut devenir un goulot d'étranglement. En particulier, la mise à jour des informations disséminées (nombre de processus en attente sur chaque processeur, charge mémoire, ...) sur l'ensemble du système peut être très pénalisante.

3.3 Les algorithmes distribués

Les algorithmes distribués mettent en œuvre une gestion distribuée de la répartition de la charge. À la différence des méthodes centralisées, ce n'est plus un processeur qui a la responsabilité de l'équilibre de charge. Ainsi chaque processeur peut entreprendre une phase d'équilibre de charge. On distingue quatre grandes catégories de stratégies distribuées :

1. Les algorithmes de type source initiative ;
2. les algorithmes de type serveur initiative ;
3. les algorithmes hybrides ;
4. les algorithmes des enchères.

La distinction entre algorithme source et serveur initiative a été introduite par Y. T. Wang et R. J. T Morris en 1985 [WM 85]. Les auteurs proposent de modéliser un système distribué de la façon suivante (voir figure 2.2):

- les tâches arrivent dans le système par l'intermédiaire d'un certain nombre de processeurs (appelé les processeurs *source*) ;
- les tâches sont calculées sur des processeurs appelés *serveur* ;

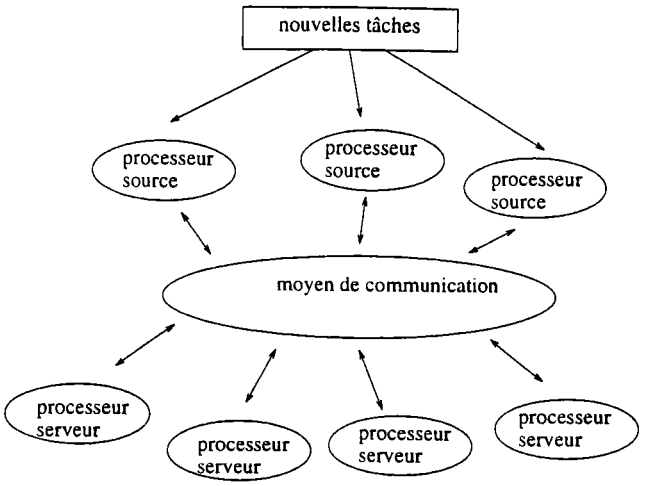


FIG. 2.2 –*définition logique d'un système distribué [WM 85]*

– un processeur peut être à la fois *source* et *serveur*.

Ainsi, on définit un algorithme comme *source initiative* si la gestion de l'équilibre de charge est à l'initiative des processeurs possédant une charge élevée. De même, quand la décision est prise par un processeur oisif ou faiblement chargé, on qualifie cet algorithme de *serveur initiative*. Il existe de nombreuses méthodes qui se comportent à la fois comme des algorithmes source et serveur initiative, on parle alors de *stratégie hybride*. Enfin, on peut considérer une dernière catégorie de stratégies distribuées, les algorithmes de type *enchère*, qui consistent à mettre en place une économie de marché. De cette façon, une tâche peut acheter une ressource (temps CPU, mémoire , ...) et un processeur peut s'offrir une tâche à exécuter.

3.3.1 Les algorithmes source initiative

Les algorithmes source initiative délèguent la gestion de l'équilibre de charge aux processeurs surchargés [GK 94, EB 93, CG 94, BS 85]. Eager *et al.* proposent trois algorithmes de ce type afin de les comparer suivant le niveau d'information qu'ils utilisent [ELZ 86a]. Les auteurs font deux hypothèses sur l'homogénéité du système:

– tous les processeurs sont identiques ;

– tous les processeurs sont soumis à la même loi d'arrivée des tâches.

3.3.1.1 Une stratégie source initiative aléatoire. C'est la technique la plus simple. Elle ne nécessite aucune information sur l'état des autres processeurs. Quand un processeur est surchargé, il choisit aléatoirement un processeur à qui il transmet une tâche. Un processeur qui reçoit une tâche peut adopter plusieurs attitudes. En effet, si un processeur qui reçoit une charge décide de la transférer à nouveau car il est lui-même surchargé, le système risque d'avoir un comportement instable. Au contraire, si ce processeur choisit de conserver malgré tout cette tâche, il risque d'être fortement pénalisé. Afin d'éviter ces deux cas extrêmes, les auteurs autorisent le transfert d'une tâche un nombre limité de fois. D'après les auteurs, cette méthode très simple, permet d'améliorer les performances

du système avec une stratégie distribuée n'utilisant aucune information sur l'état global du système.

3.3.1.2 Une stratégie source initiative à seuil. Comme pour la méthode précédente, quand un processeur P_i est surchargé, il choisit aléatoirement un processeur P_j. Avant de transférer une tâche, le processeur P_i vérifie si ce transfert ne risque pas de surcharger le processeur destination (la charge de P_j est au dessus d'un certain seuil). Quand le transfert n'est pas possible, un autre processeur est désigné. Cette méthode est appliquée jusqu'à trouver un processeur dont la charge est en dessous du seuil ou bien le nombre de tentatives est supérieure à une constante définie par l'utilisateur. L'objectif de cette stratégie est d'éviter de réaliser des transferts de tâches inutiles. Comme pour la stratégie aléatoire, un processeur n'essaie pas de faire le meilleur choix mais simplement un choix qui améliore sa situation de façon locale. Le niveau d'information utilisé par cette méthode est peu élevé.

3.3.1.3 Une stratégie source initiative du meilleur choix. Un processeur surchargé interroge un certain nombre de processeurs et tente de réaliser le meilleur choix parmi ce groupe. La mise en œuvre de cette stratégie demande un plus grand nombre d'informations concernant le système. Un processeur surchargé interroge un groupe de processeurs choisis aléatoirement. Il transfert une tâche en direction du processeur le moins chargé si le processeur destination n'est pas déjà surchargé.

Les résultats obtenus par simulation montrent que les trois stratégies améliorent de façon significative les performances globales du système. La stratégie la moins performante est, comme on pouvait s'y attendre, la stratégie aléatoire. En effet, plusieurs transferts inutiles sont réalisés par cette stratégie qui n'a aucune information sur l'état global du système. Par contre, la stratégie du meilleur choix améliore très peu les résultats obtenus par la stratégie du seuil. Ainsi, les auteurs concluent que l'on peut développer des stratégies source initiative efficaces qui utilisent un nombre réduit d'informations sur l'état du système.

3.3.1.4 Un anneau logique. Les stratégies précédentes ont un inconvénient majeur : il est possible que des tâches soient transférées inutilement ce qui pénalise les performances globales du système. Afin d'éviter ce défaut, Guyennet *et al.* proposent de gérer différemment l'ensemble des processeurs [GSS 92, Spi 94]. Il propose un algorithme source initiative qui utilise un anneau logique pour réaliser efficacement un équilibrage dynamique de la charge. Cet anneau logique est constitué par les processeurs déchargés. Chacun des processeurs qui le constituent connaît deux autres processeurs déchargés de l'anneau qui sont situés à sa gauche et à sa droite. De même, chaque processeur dans un état normal ou surchargé connaît un point d'entrée (un numéro de processeur) sur l'anneau logique des processeurs déchargés (voir figure 2.3).

À partir de cette structure, un processeur surchargé peut facilement déplacer une tâche vers un processeur déchargé. Il peut dans le cas présent mettre en œuvre différentes méthodes afin de choisir ce processeur oisif :

- il peut confier directement une tâche au processeur qui est son point d'entrée sur l'anneau ;

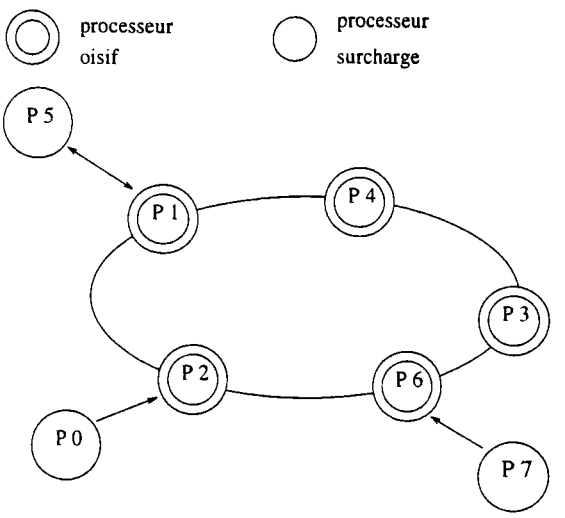


FIG. 2.3 –*anneau logique des processeurs déchargés*

- il peut interroger une partie des processeurs de l'anneau par l'intermédiaire de son point d'entrée afin de faire un meilleur choix ;
- il peut enfin choisir d'interroger tout l'anneau afin de faire le meilleur choix.

La simulation réalisée par les auteurs montre que cet algorithme est très performant. De plus, pour obtenir une efficacité maximale de cet algorithme, il est préférable d'interroger plusieurs processeurs déchargés avant de transférer une tâche, mais il est inutile d'interroger tous les processeurs qui composent l'anneau. Comme pour les algorithmes proposés par Eager *et al.*, c'est la méthode qui utilise un nombre moyen d'informations concernant le système qui obtient les meilleurs résultats.

3.3.2 Les algorithmes serveur initiative

Les algorithmes serveur initiative peuvent être vus comme les algorithmes duaux des algorithmes source initiative. Le fait que la responsabilité de l'équilibre de charge soit confiée aux processeurs surchargés est une critique généralement formulée envers les algorithmes source initiative. En effet, leur état est déjà pénalisant pour le système et ce type de stratégie accroît encore leur charge de travail. C'est pourquoi, un grand nombre de chercheurs ont proposé des stratégies où le contrôle de la charge est laissé sous la responsabilité des processeurs oisifs [NXG 85, BS 85, LMR 91, WR 93]. Nous proposons de détailler deux algorithmes particuliers qui mettent en œuvre une stratégie serveur initiative. La première apporte une solution globale au problème de l'équilibre de charge [LK 86]. Par contre, la seconde méthode apporte une solution locale à ce problème [WR 89].

3.3.2.1 La méthode du gradient. F. C. H. Lin et R. M. Keller proposent une stratégie serveur initiative qui apporte une solution globale au problème de l'équilibre de charge. Les auteurs supposent qu'une hypothèse de voisinage est vérifiée. Ainsi, un processeur p possède plusieurs liens de communication en direction de différents processeurs. Ce groupe de processeurs forme le voisinage de p. Il n'est pas possible pour p, de communiquer directement avec un processeur qui n'appartient pas à son voisinage. Le principe de cet

algorithme est de permettre à un processeur oisif de le signaler à ses voisins dans le but de récupérer un travail à effectuer. Cette information se propage de proche en proche, par l'intermédiaire des voisins, en direction de tout le système si le problème ne peut être résolu de façon locale.

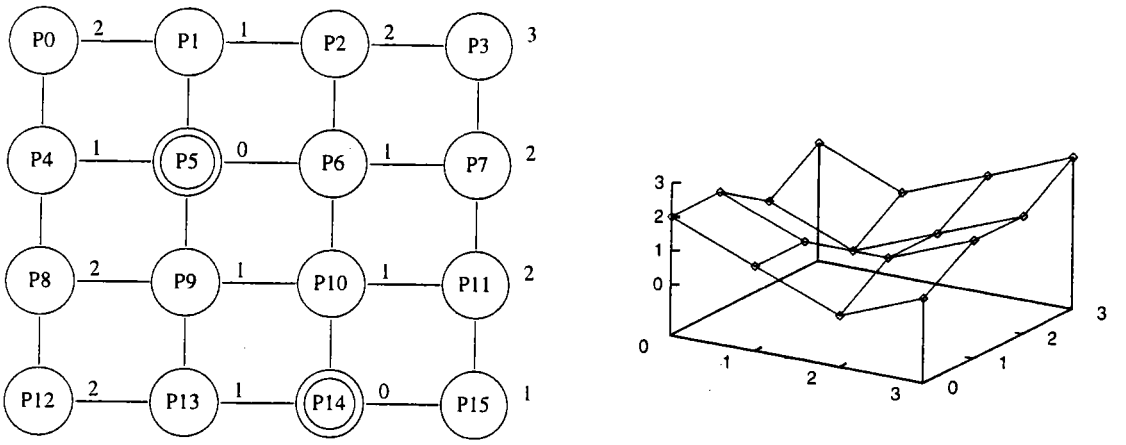


FIG. 2.4 –*évaluation de la variable proximité et surface du gradient associée [LK 86]*

Les auteurs définissent un seuil minimal et un seuil maximal. Quand un processeur est en dessous du seuil minimal, il est considéré comme étant déchargé, de même un processeur est surchargé si sa charge est au dessus du seuil maximal (voir paragraphe 3.7.2.1). L'intérêt principal de cette méthode à double seuil est d'éviter un changement trop rapide de l'état déchargé vers l'état surchargé et inversement.

Chaque processeur mesure la distance (le nombre de processeurs) qui le sépare du processeur déchargé qui lui est le plus proche. Quand un processeur a une charge inférieure au seuil minimal, il a une proximité de 0. De façon générale, la proximité de p est définie de la façon suivante:
$proximité(p) = min(proximité(n_i)) + 1$ où les n_i sont les voisins de p. Au départ, la proximité de tous les processeurs est initialisée à w_{max} (la distance maximale séparant deux processeurs). Dans cette configuration, l'équilibre de charge est satisfaisant car chaque processeur est au dessus du seuil minimal. À l'aide de cette variable, une surface, appelée *surface du gradient*, est construite. La hauteur d'un point (qui représente un processeur p) est déterminée par la valeur de $proximité(p)$. Ainsi, la surface a une hauteur nulle en un point si le processeur représenté par ce point est déchargé. De même, si la hauteur de la surface est de d pour un processeur particulier, cela signifie qu'il n'est pas oisif et qu'il se situe à une distance d d'un processeur déchargé.

Quand un processeur P est surchargé, il réalise un transfert d'une de ses tâches en direction d'un de ses voisins s'il existe un processeur oisif dans le système ($proximité(P) < w_{max}$). Il choisit le voisin dont la variable *proximité* est minimale. Ainsi la tâche est conduite de proche en proche vers ce processeur qui est inoccupé (voir la figure 2.4). De point de vue visuel, on a l'impression que les charges émises par des processeurs surchargés suivent la pente de la surface du gradient.

Cet algorithme peut devenir instable dans certain cas. En effet, si le nombre de processeurs déchargé est réduit, un processeur qui est déchargé risque de se retrouver très rapidement surchargé car tous les processeurs surchargés vont émettre dans sa direction des tâches sans aucune concertation. Afin d'éviter cet inconvénient F. J. Muniz et E. J.

Zaluska proposent de modifier légèrement cette stratégie [MZ 95]. Quand un processeur surchargé P_i décide de transférer une charge en direction du processeur P_j, il lui signale au préalable son intention. P_j est libre d'accepter ou de refuser ce transfert. Cette optimisation permet d'éviter le transfert inutile de charge à travers le système. Cependant, le coût en communication de la méthode du gradient est non négligeable puisqu'un grand nombre de messages de contrôles est induit pour s'assurer de la validité des informations du système avant de réaliser un déplacement de tâche.

3.3.2.2 Une stratégie locale de type serveur initiative. M. Willebeek-Lemair et A. P. Reeves proposent une stratégie locale de type serveur-initiative [WR 89]. Ils justifient ce choix par le fait qu'une politique locale est plus adaptée pour une machine fortement parallèle qu'une stratégie globale. En effet, quand le nombre de processeurs est important, il est très coûteux de maintenir à jour un niveau d'information global. Cependant, nous pouvons nuancer cette affirmation des auteurs par le fait général qu'une stratégie trop locale peut prendre de mauvaises décisions car les informations disponibles peuvent être insuffisantes. Dans le cas précis de l'équilibrage de charge, des tâches risquent d'être déplacées inutilement vers un processeur déjà chargé, alors qu'il peut exister des processeurs déchargés qui ne recevront pas de travail car leurs voisins immédiats ne sont pas assez chargés.

Comme pour l'algorithme du gradient, par hypothèse, chaque processeur possède K voisins. Le but de cette stratégie est de maintenir un équilibre de charge local sur l'ensemble des groupes de voisinage définis. Le principe de l'algorithme est le suivant :
La charge moyenne locale (pour chaque groupe de processeurs) est calculée régulièrement :
$\bar{L_p} = \frac{1}{K+1}(l_p + \sum_{k=1}^{K} l_k)$ avec l_i la charge du processeur i.
Chaque processeur k surchargé ($l_k > \bar{L_p}$) calcule sa surcharge :
$h_k = l_k - \bar{L_p}$ ($h_k = 0$ pour les autres)
La surcharge totale du système est : $\bar{H_k} = \sum_{i=1}^{k} h_k$.
Le processeur p peut recevoir $\bar{L_p} - l_p$ charges supplémentaires. Il répartit sa demande sur l'ensemble de ses voisins :
$\delta_k = (\bar{L_p} - l_p) \times \frac{h_k}{\bar{H_k}}$ avec δ_k le nombre de tâches prises par p au processeur k.

Cet algorithme vise à maintenir un état stable, en ne modifiant la charge que de façon locale. Il est intéressant de noter que l'utilisateur n'a pas besoin de définir un seuil pour déterminer si un processeur est surchargé ou non. En effet ce seuil est calculé automatiquement à l'aide de la variable $\bar{L_p}$. L'algorithme peut donc s'adapter localement à l'état du système. Si un déséquilibre important existe au niveau d'un groupe de voisins, un grand nombre de tâches sera transféré des processeurs surchargés vers les processeurs déchargés. Cependant, il est à noter que le nombre de messages de contrôle nécessaires est très important car chaque processeur doit communiquer régulièrement un indicateur de charge à tous ses voisins. Ainsi, pour réaliser une mise à jour de ces informations au niveau du système, chaque processeur émet K messages en direction de ses différents voisins et reçoit K messages en retour. On peut donc estimer qu'une mise à jour du système nécessite $N \times K$ échanges de messages où N est le nombre de processeurs. Pour que cet algorithme soit efficace, il est donc nécessaire de disposer d'un mécanisme de communication efficace

entre processeurs voisins.

3.3.3 Comparaison des méthodes source et serveur initiative

Les deux stratégies ont un avantage commun par rapport aux méthodes centralisées, elles évitent la formation de goulots d'étranglement. Par contre, pour obtenir un même niveau d'information qu'une stratégie centralisée, le nombre de messages échangés par les méthodes distribuées est plus important. Cependant, il est à noter que des stratégies simples (nécessitant un nombre réduit de messages de contrôle) permettent d'obtenir de bons résultats [ELZ 86a].

Eager *et al.* ont comparé les stratégies source et serveur initiative dans le cadre des systèmes distribués [ELZ 86b, ELZ 86c]. Ils parviennent aux conclusions suivantes :

- aucune des deux stratégies source et serveur initiative n'est plus efficace dans tous les cas de figures ;
- quand la charge globale du système est faible à moyenne, les stratégies de type source initiative sont plus efficaces que les stratégies serveur initiative ;
- quand la charge globale du système est très élevée, les algorithmes de type serveur initiative sont plus performants que les stratégies source initiative.

Dans le cas d'une machine parallèle dédiée à une seule application, on constate que les méthodes serveur initiative sont très efficaces en début d'exécution et que les stratégies source initiative sont plus performantes en fin d'exécution. En effet, quand l'application débute, il reste encore un grand nombre de tâches à calculer. Le nombre de processeurs déchargés est réduit et il est donc très facile de trouver un processeur surchargé avec qui réaliser un équilibrage de la charge. Par contre, en fin d'exécution, les processeurs surchargés sont rares car l'essentiel des tâches a déjà été calculé. C'est pourquoi, il est préférable que les processeurs surchargés soient à l'origine des nouvelles phases d'équilibre de charge.

3.3.4 Les algorithmes hybrides et de type enchères

D'après les comparaisons menées entre les algorithmes source et serveur initiative, nous pouvons constater qu'aucune des deux méthodes n'est parfaite. Afin de conserver les avantages de ces méthodes sans les inconvénients, il est possible de proposer une stratégie hybride qui est à la fois source et serveur initiative. Les algorithmes des enchères vont mettre en place une économie de marché afin de limiter les défauts des stratégies source et serveur initiative.

3.3.4.1 Les algorithmes hybrides se comportent à la fois comme source et serveur initiative. C'est à dire que la décision de réaliser une phase d'équilibre de charge peut être prise par un processeur surchargé ou déchargé. L'algorithme paire proposé par R. Bryant et R. Finkel est un algorithme hybride [BF 81]. Les processeurs essaient de former des paires afin de maintenir un équilibre de charge local. Un processeur ne peut appartenir qu'à une seule paire. La phase de négociation pour trouver un processeur avec qui réaliser une paire est prépondérante. Une paire est détruite quand il n'est plus possible pour les deux processeurs qui la constitue d'échanger des tâches. Dans ce cas, chacun des deux

processeurs essaie de constituer une nouvelle paire avec des processeurs différents. Cet algorithme essaie de limiter le nombre de phases d'équilibre de charge en maintenant des relations privilégiés entre les processeurs aussi longtemps que possible.

3.3.4.2 Les algorithmes de type enchère essaient de mettre en place une économie de marché afin de réaliser un équilibre de charge correct du système. Il existe dans les système des clients (demandeurs de services) et des prestataires de service. Ainsi, quand un client fait un appel d'offre (par exemple, une demande de temps de calcul), chaque prestataire de service répond par une offre. Le client choisit l'offre la moins coûteuse. De nombreux algorithmes de type enchère ont été proposés [Smi 80, CK 84, SS 84]. D. Ferguson *et al.* proposent un algorithme de type enchère [FYN 88]. L'économie mise en place par cet algorithme est complètement définie par les actions de deux types d'agents que sont les tâches et les processeurs. Quand une tâche est créée dans le système, elle reçoit une certaine somme d'argent. Elle doit s'acheter au moindre coût les ressources en temps de calcul dont elle a besoin pour s'exécuter. Une tâche est autorisée à migrer à travers l'ensemble du système, mais elle devra payer à chaque fois qu'elle choisit d'utiliser un lien de communication. Le fait de faire payer un processus pour emprunter un lien de communication permet d'éviter une trop forte fluctuation de la charge à l'intérieur du système. Les processeurs envoient régulièrement des publicités aux processeurs voisins afin de faire connaître le coût de leur service. Ce prix est fixé en fonction de la disponibilité du processeur. Ainsi, un processeur déchargé fixe un prix très bas pour attirer un maximum de clients à lui, alors qu'un processeur surchargé choisit de fixer un prix élevé pour son service. Cet algorithme a été simulé par les auteurs et ils obtiennent des résultats très intéressants. Il est à noter que les auteurs supposent qu'une hypothèse très forte est vérifiée: on connaît à la création d'une tâche les ressources qui lui seront nécessaires. De plus, la mise à jour des informations par l'intermédiaire de publicités engendre un grand nombre de messages qui risquent de saturer le réseau de communication.

3.4 Les algorithmes semi-distribués

L'ensemble de cette étude montre que les approches distribuées et centralisées ont chacune leurs avantages et inconvénients. Les approches distribuées sont généralement extensibles car elles utilisent des informations locales, mais le placement obtenu n'est pas toujours satisfaisant. Les stratégies centralisées quant à elles, sont efficaces et faciles à implémenter pour un nombre réduits de processeurs. Par contre, elles sont plus difficilement extensibles quand le nombre de processeurs augmente. C'est pourquoi il est souhaitable de proposer des stratégies semi-distribuées afin de conserver un équilibre de charge global correct en évitant les problèmes liés à la gestion centralisée des informations. Il est possible de distinguer les algorithmes semi-distribués hiérarchiques [Eck 94a, WR 93] des algorithmes à partitionnement logique [AG 91].

3.4.1 Les algorithmes hiérarchiques

Les algorithmes hiérarchiques proposent d'organiser l'ensemble des processeurs sous la forme d'un arbre. Chaque nœud père de l'arbre dispose des informations connues au niveau de ses différents fils. Ainsi, plus le nœud est haut dans l'arbre, plus il dispose d'informations sur l'ensemble du système. La méthode HBM (*Hierarchical Balancing Method*)

est une stratégie asynchrone globale [WR 93]. Le système est organisé sous la forme d'un arbre binaire. À chaque niveau de l'arbre, un processeur est responsable de l'équilibre de charge des sous-arbres dont il est le père (voir la figure 2.5). Pour initialiser la structure arborescente, chaque feuille de l'arbre (qui est un processeur) envoie la valeur de son indicateur de charge à son père. Le déséquilibre de charge est mesuré à chaque niveau de l'arbre. Si le déséquilibre est supérieur à un seuil fixé par l'utilisateur entre deux sous arbres A_1 et A_2, la charge doit alors être redistribuée entre ces deux arbres. Le nœud responsable de A_1 et A_2 prévient chacun des processeurs et leur indique le niveau h où a été détecté le déséquilibre. Chaque processeur de A_1 est alors en mesure de déterminer, en fonction de h, le processeur de A_2 avec qui il est associé. De même, chaque processeur de A_2 connaît le processeur de A_1 avec qui il va échanger un certain nombre de tâches. Dans le cas particulier où l'architecture du système est un hypercube, chaque processeur de A_1 est relié par un lien direct avec son partenaire de l'arbre A_2 (voir la figure 2.5). Quand les paires sont formées, un équilibre de charge locale est appliqué. Par exemple, si le processeur 0 détecte un déséquilibre au niveau $h = 0$, alors le processeur 0 est associé au processeur 4. De même, trois autres paires de processeurs sont créées: $(1,5)$, $(2,6)$, $(3,7)$.

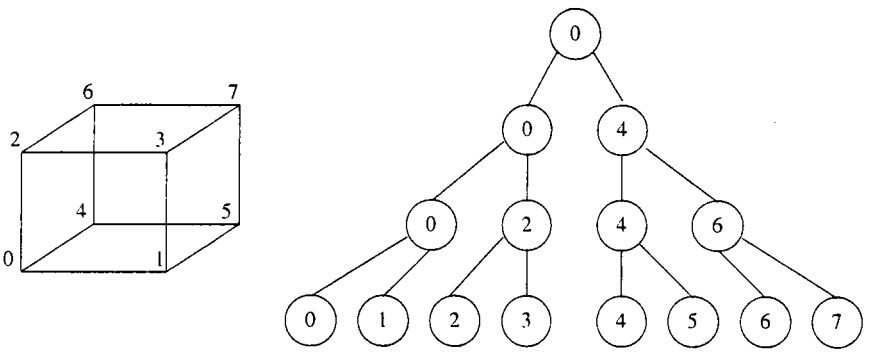


FIG. 2.5 –*Organisation hiérarchique d'un hypercube comprenant 8 processeurs [WR 93]*

Dans le principe, cet algorithme garantit qu'à un niveau h, tous les processeurs du niveau $h-1$ ont la même charge. En pratique, pour que cette propriété soit vérifiée, il est nécessaire que le temps de mise à jour des informations dans l'arbre soit très inférieur au temps de calcul d'une tâche.

3.4.2 Les algorithmes à partitionnement logique

Il est possible de diviser le système en un ensemble de sous-groupes afin de réduire le coût d'une phase d'équilibre de charge. Ainsi, un processeur est responsable d'un groupe de processeurs. Il maintient sur son domaine, un équilibre correct de la charge et peut choisir de réaliser une phase d'équilibre au niveau supérieur en coopérant avec d'autres groupes. I. Ahmad et A. Ghafoor proposent un algorithme qui décompose le système en un ensemble de sphères [AG 91]. Pour chaque sphère, un processeur est responsable de l'équilibre de charge. Au niveau de chaque sphère, une stratégie de type client-serveur est appliquée. Le processeur P_S, responsable de la sphère S, mesure régulièrement la charge de sa sphère qui est représentée par le nombre total de tâches présentes au niveau des différents processeurs qui compose la sphère S. De plus, P_S maintient à jour une liste des processeurs classés par ordre croissant en fonction de leur estimateur de charge. Le

premier processeur de cette liste est donc le processeur le moins chargé de la sphère S. L'algorithme appliqué est le suivant:

Algorithme 2.1 *Partitionnement*

si la charge du processeur le moins chargé est inférieure à t_1 **alors**
transférer une tâche sur ce processeur
sinon si la différence de charge entre S et la sphère la moins chargée est supérieur à t_2 **alors**
transférer une tâche vers cette sphère
finsi

Ainsi, une stratégie de type source initiative est appliquée au niveau supérieur. Si une sphère estime qu'elle est surchargée, elle choisit une sphère distante à qui envoyer une tâche. Ce type d'algorithme permet d'obtenir un équilibre de charge global correct en minimisant les coûts de communication.

3.5 Une classification horizontale : les propriétés d'une stratégie

La classification hiérarchique que nous avons présentée, n'est pas suffisante pour caractériser complètement une stratégie d'équilibre de charge. Dans le but d'affiner cette première approche, il est nécessaire d'étudier différents points [Fon 94, CLZ 92, GS 93]:

- comment sont gérées les informations concernant l'état du système?
- quelle est la méthode utilisée pour évaluer la charge?
- quelle est la méthode d'appariement?
- la méthode est elle préemptive, c'est à dire autorise-t-elle la migration de tâche?
- quelles sont les qualités de la stratégie (stabilité, extensibilité, ...)?

3.6 La gestion de l'information

La gestion de l'information est un élément déterminant d'une stratégie. En effet, c'est en s'appuyant sur ces informations que les décisions concernant la répartition de la charge vont être prises. C'est pourquoi le système doit pouvoir disposer d'un nombre important d'informations fiables sur l'état du système. Cependant, cette gestion de l'information ne doit pas être trop coûteuse. La gestion de l'information se caractérise par: la méthode utilisée pour la mise à jour des données et le niveau d'information mis en œuvre.

3.6.1 Mise à jour de l'information

On distingue généralement trois grandes méthodes de mise à jour de l'information:

1. mise à jour périodique ;
2. mise à jour à la demande ;
3. mise à jour lors des changements d'état.

3.6.1.1 Mise à jour périodique. La méthode la plus simple consiste à mettre à jour régulièrement les informations disponibles. Dans une approche centralisée, chaque processeur envoie régulièrement sa charge vers le serveur. Par exemple, S. Zhou propose deux algorithmes centralisés (les méthodes *GLOBAL* et *CENTRAL*) utilisant cette technique pour mettre à jour l'information [Zho 88]. Un vecteur de charge est ainsi créé au niveau du serveur et il peut alors choisir de le communiquer à d'autres processeurs. Dans une approche distribuée, chaque processeur informe régulièrement quelques processeurs (ses voisins) ou l'ensemble du système.

L'intervalle de temps δt entre deux mises à jour des informations est délicat à déterminer. Si δt est trop petit, les nombreuses mises à jour peuvent conduire à une saturation du réseau de communication sans permettre d'améliorer les performances du système. Au contraire, si δt est trop grand, les informations connues par certains processeurs risquent d'être caduques. Il en résulterait une mauvaise décision concernant la modification de la charge.

3.6.1.2 Mise à jour à la demande. Quand un processeur a besoin d'une estimation de la charge d'un processeur distant, il lui adresse une requête. Dans le cas d'une stratégie source initiative, ce sont les processeurs surchargés qui formulent des requêtes [ELZ 86a]. Si la stratégie est de type serveur initiative, les requêtes sont formulées par les processeurs inactifs [KGV 94]. La stratégie centralisée proposée par S. Patil et P. Banerjee réalise aussi une mise a jour à la demande [PB 90]. En effet, avant de prendre une décision de rééquilibrage (à la demande d'un processeur oisif), le serveur met à jour l'ensemble des informations dont il dispose en interrogeant les processeurs supposés surchargés.

Le nombre de communications engendrées est très réduit, ce qui représente le principal avantage de cette technique. Par contre, un processeur ne dispose pas instantanément des informations qui lui sont nécessaires car il doit attendre que sa requête soit exécutée par le processeur distant.

3.6.1.3 Mise à jour lors des changements d'état. Si un processeur estime que son état a changé de manière sensible, il choisit d'en informer d'autres processeurs. Par exemple, si un processeur passe d'un état surchargé à un état normal ou déchargé, il peut transmettre cette information à l'ensemble du système par un *broadcast* ou seulement à ses voisins. Ainsi la stratégie proposée par K. G Shin et Y-C. Chang répartit l'ensemble des processeurs dans différentes partitions [SC 89]. Au niveau de chaque partition, un processeur dont l'état est modifié le signale à l'ensemble des processeurs constituant cette partition.

La mise à jour lors des changements d'état vise à réduire le nombre global de messages échangés par l'ensemble des processeurs. Cette technique est une méthode intermédiaire entre une mise à jour périodique et une mise à jour à la demande. La mise à jour lors des changements d'état évite de réaliser un certain nombre de communications inutiles et un processeur accède sans délai d'attente aux informations dont il a besoin.

3.6.2 Niveau d'information

Plus la connaissance de l'état du système est complète, plus il est facile de répartir au mieux l'ensemble des tâches à calculer. Cependant, disposer d'informations globales sur

l'état du système est très coûteux en terme de communication. Par contre, une connaissance locale de la charge du système nécessite un nombre réduit d'échanges de messages. Les décisions prises à l'aide d'informations locales améliorent généralement les performances du système, mais il est possible qu'une décision erronée soit prise car aucune information globale n'est connue. Il est donc nécessaire de réaliser un compromis entre niveau de connaissance et qualité de l'équilibre de charge obtenu.

3.6.2.1 Informations locales sur l'état du système. Prendre une décision à l'aide d'informations locales afin de modifier la répartition de la charge présente principalement deux avantages :

1. la mise à jour de ces informations n'est pas pénalisante car elles concernent un nombre réduit de processeurs ;
2. les informations sont disponibles rapidement.

L'algorithme *RID* utilise des informations locales afin de modifier sa charge [WR 93]. C'est à partir de la connaissance de la charge de ses voisins qu'il détermine son comportement. Le principal inconvénient de ce critère de localité est que ce type de stratégie est mal adaptée dans le cas d'un fort déséquilibre de la charge. En effet, si une partie du système seulement est surchargée, le rééquilibrage de la charge sur l'ensemble du système sera très long car la charge va se propager de proche en proche. Par contre, si le système est relativement bien équilibré, une modification locale de la charge suffit à maintenir une bonne efficacité du système.

3.6.2.2 Informations globales sur l'état du système. Disposer d'informations globales sur l'état du système, permet de rééquilibrer facilement la charge sur l'ensemble du système. Ainsi l'algorithme du gradient présenté au paragraphe 3.3.2.1 permet de résoudre au niveau global le problème de l'équilibre de charge. Pour maintenir à jour les informations, elles sont diffusées de proche en proche en utilisant la notion de voisinage. Afin de réduire le nombre total de communications, l'algorithme vecteur met en œuvre une méthode originale [BS 85]. Supposons que le système comporte n processeurs, chaque processeur P_i maintient à jour un vecteur de charge V^i contenant l valeurs. V_0^i est la charge locale du processeur P_i. Les $l-1$ autres valeurs représentent les charges de $l-1$ processeurs du système choisis aléatoirement. Régulièrement, le processeur P_i choisit au hasard un processeur P_j et lui transmet la moitié inférieure de son vecteur V^i. De même, le processeur P_j envoie la moitié de V^j. Le processeur P_i met à jour V^i de la façon suivante :

$$V_{2c}^i = V_c^i, \forall c \in [l/2, l-1] \text{ et } V_{2c+1}^i = V_c^j, \forall c \in [0, l/2-1].$$

Plus l est grand, plus l'information connue sera précise, mais plus la mise à jour des informations prendra de temps. Il est donc nécessaire de déterminer une valeur de l qui réalise un bon compromis entre le niveau d'information et le coût de la mise à jour afin de permettre une prise de décision efficace en évitant une surcharge du réseau de communication.

3.7 Évaluation de la charge

La méthode utilisée par une stratégie pour évaluer la charge peut influer de façon significative sur le comportement global de l'algorithme. Pour obtenir un algorithme performant, il est nécessaire de définir un indicateur de charge adapté au problème à résoudre. À l'aide de cet indicateur de charge, le mécanisme de sélection est capable de déterminer l'état de chaque processeur : le processeur est il surchargé, déchargé ou oisif?

3.7.1 Indicateurs de charge

L'indicateur de charge est un paramètre d'une stratégie d'équilibre de charge qui est dépendant de l'application cible. La longueur de la file d'attente des processus à traiter est souvent utilisée comme indicateur de charge pour les systèmes distribués. On peut aussi choisir de considérer la taille mémoire des processus, le nombre de requêtes d'entrées-sorties, etc.

La notion de charge élémentaire est, elle aussi, étroitement liée à l'application cible. Dans certains cas, il possible de proposer plusieurs définitions pour une tâche élémentaire. Dans le cadre du lancer de rayons parallèle, une tâche peut être définie comme le calcul de la couleur d'un pixel. Si cette définition est choisie, on connaît alors toutes les tâches au début de l'exécution , mais le temps de calcul de chaque tâche est inconnu. Il est aussi possible de définir une tâche comme un calcul d'intersection entre un rayon et la scène. Dans ce cas, on ne connaît pas le nombre total de tâches à traiter car un rayon primaire peut générer des rayons d'ombrage, de réflexion et de transparence.

3.7.2 Mécanisme de sélection

Le mécanisme de sélection est chargé de déterminer l'état de chaque processeur. C'est en s'appuyant sur cette information, que la stratégie d'équilibre de charge décide des processeurs qui participent au rééquilibrage de la charge. Pour parvenir à ce résultat, trois techniques peuvent être appliquées. En premier lieu, il est possible de déterminer l'état de chaque processeur à l'aide de seuils. Une estimation de l'état d'un processeur peut aussi être obtenue par comparaison à d'autres processeurs. Enfin, les méthodes systématiques sélectionnent l'ensemble des processeurs, sans tenir compte des indicateurs de charge, afin que chaque processeur participe à la nouvelle phase d'équilibre de charge.

3.7.2.1 Utilisation de seuils. L'utilisation d'un seuil unique S_{min} permet de définir deux états pour un processeur : quand la charge d'un processeur est en dessous du seuil S_{min}, le processeur est considéré comme étant déchargé. Par contre, si sa charge est supérieure à S_{min}, le processeur est surchargé. Cette méthode est très simple, mais elle présente deux inconvénients : la valeur du seuil doit être fixé par l'utilisateur et l'état d'un processeur peut rapidement être modifié. Ainsi, il est possible qu'un processeur qui était considéré comme surchargé à l'instant précédent peut être désigné comme oisif à l'instant suivant. Il peut alors en résulter une fluctuation de la charge qui est pénalisante pour le comportement global du système.

Afin d'éviter cet inconvénient, Ni *et al.* ont proposé l'utilisation de deux seuils S_{min} et S_{max} [NXG 85] (voir figure 2.6). Ainsi l'état d'un processeur P peut être défini à l'aide de sa charge L_P comme déchargé, normal ou surchargé :
Si $L_P < S_{min}$ alors P est déchargé.

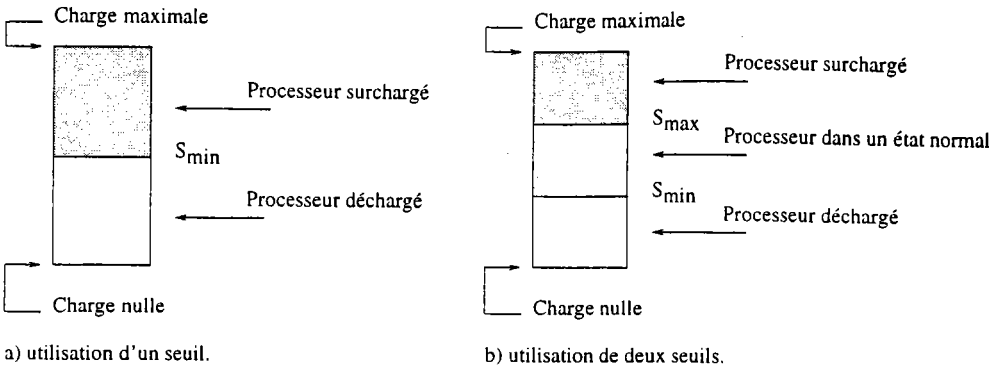


FIG. 2.6 –*Évaluation de l'état d'un processeur en fonction de sa charge à l'aide de seuils.*

Si $S_{min} < L_P < S_{max}$ alors la charge de P est normale.
Si $L_P > S_{max}$ alors P est surchargé.

L'utilisation de deux seuils permet d'obtenir une évaluation stable de l'état d'un processeur en utilisant peu de ressources. Cependant, comme dans le cas d'un seuil unique, c'est la responsabilité de l'utilisateur de fixer des valeurs convenables pour les deux seuils.

3.7.2.2 Par comparaison. Un processeur peut déterminer son état en comparant la charge locale à la charge d'autres processeurs. À la différence des méthodes à seuils, un processeur détermine son comportement futur en fonction d'informations distantes qu'il doit demander. Dans les algorithmes proposés par Lüling *et al.*, un processeur participe à une nouvelle redistribution des tâches si sa différence de charge avec ses voisins est supérieur à une valeur d [LMR 91]. Cette valeur peut être calculée dynamiquement par l'algorithme. L'évaluation par comparaison de l'état d'un processeur est donc plus adaptée à la charge réelle du système, mais des communications sont nécessaires. Dans le cas d'une évolution rapide et non prévisible de la charge globale du système, il est préférable d'appliquer ce type de méthode car il est difficile de fixer des seuils pour déterminer l'état d'un processeur sans communication.

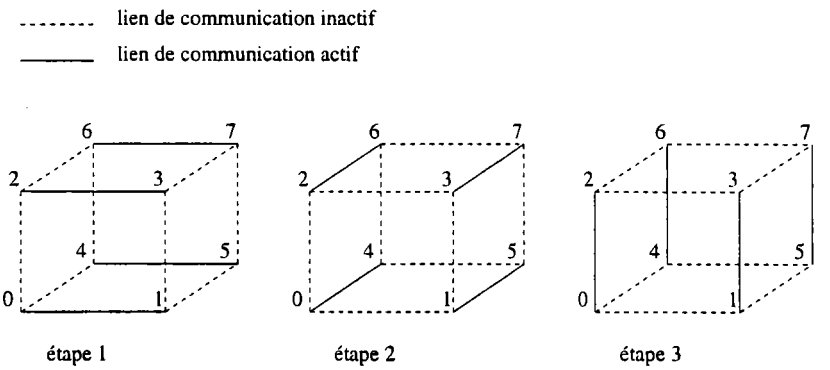


FIG. 2.7 –*Méthode d'échange dimensionnelle pour un hypercube de dimension trois [XL 92]. La phase d'équilibre de charge comporte trois étapes.*

3.7.2.3 Méthodes systématiques. Les méthodes systématiques ne se préoccupent pas de l'état des processeurs pour déterminer s'ils seront émetteurs ou récepteurs. Xu et

Lau proposent une méthode systématique qui s'applique à un hypercube [XL 92, XML 95]. La méthode d'échange dimensionnelle nécessite une synchronisation de l'ensemble des processeurs. Un hypercube de dimension k est composé de 2^k processeurs. Chaque processeur possède k voisins avec qui il peut communiquer directement. L'algorithme comporte k étapes. À l'étape i, chaque processeur échange une partie de sa charge avec son $i^{ème}$ voisin (voir la figure 2.7). Quand la $k^{ème}$ étape est terminée, la charge est équitablement distribuée sur l'ensemble des processeurs. Ce type de stratégie est difficilement applicable aux machines MIMD, car il faut disposer d'un mécanisme de synchronisation efficace. Cependant, les méthodes systématiques sont particulièrement bien adaptées aux machines SIMD où la synchronisation entre chaque instruction est systématique.

3.8 L'appariement des processeurs

Quand un processeur a décidé qu'il doit rééquilibrer sa charge, c'est la politique d'appariement qui lui indique avec qui il doit échanger des tâches. Cette politique est très dépendante de la méthode choisie pour gérer l'information (centralisée, distribuée). En effet, il est plus facile d'appliquer une méthode d'appariement centralisée (au niveau du serveur) quand la gestion de l'information est elle même centralisée. On peut distinguer dans chaque cas (centralisé, distribué), les politiques aveugles, par sondage ou utilisant un critère de recherche.

3.8.1 Appariement aveugle

Quand la méthode d'appariement est aveugle, une méthode aléatoire peut être appliquée par chaque processeur. Quand un processeur décide d'envoyer ou de recevoir des tâches, il émet une requête en direction d'un processeur choisi aléatoirement. Ainsi la stratégie source initiative aléatoire proposée par Eager *et al.* émet des tâches sans aucune information sur l'état du processeur auquel elle sont destinées [ELZ 86a].

Il est aussi possible d'éviter un choix aléatoire du partenaire, une variable *cible* peut être alors centralisée au niveau d'un serveur. La variable *cible* désigne le prochain processeur à utiliser pour une phase d'équilibre de charge. Quand un processeur veut trouver un partenaire, il interroge le serveur qui lui transmet la valeur de *cible*. Le serveur incrémente ensuite *cible* de un en modulo. Grâce à une méthode systématique, nous pouvons garantir que tous les processeurs participeront aux phases d'équilibre de charges de manière équitable.

Le principal avantage de l'appariement aveugle est qu'aucune information sur l'état du système est nécessaire. Par contre, le choix d'appariement réalisé n'est pas toujours satisfaisant. En effet, il est possible qu'un processeur déjà surchargé soit contraint d'accepter des tâches d'un processeur qui est lui aussi déchargé. Dans ce cas, la situation globale du système n'est pas améliorée, mais des communications inutiles ont perturbé l'application.

3.8.2 Appariement par sondage

Plutôt que d'appliquer une méthode aveugle, il est possible de procéder par sondage. Quand un processeur cherche à former un paire, il choisit parmi un groupe le processeur le mieux adapté.

Si une notion de voisinage existe, il est possible de regrouper tous les processeurs qui sont voisins dans un même groupe [LM 93]. Un processeur oisif peut alors choisir de s'appareiller avec le processeur voisin dont la charge est la plus élevée.

Le groupe peut être construit à chaque nouvelle phase d'équilibre de charge. Ainsi on fixe le nombre de processeurs qui constitue le groupe, et les processeurs sont choisis aléatoirement. Le processeur qui veut s'appareiller, choisit parmi ce groupe le meilleur candidat suivant ses critères. Eager *et al.* ont montré que cette solution améliore de façon significative la répartition de la charge tout en maintenant le surcoût en communication à un niveau raisonnable [ELZ 86a].

3.8.3 Appariement utilisant un critère de recherche

Un processeur peut choisir pour s'appareiller un processeur qui vérifie un critère très précis. Cette recherche peut être réalisée au niveau local ou global. C. Fonlupt propose l'algorithme SIMD *central* qui utilise un critère de recherche [Fon 94]. Cet algorithme s'appuie sur des communications globales. Il met en place un mécanisme de sélection à double seuil. Pendant la phase d'initialisation, la charge théorique moyenne est calculée (charge globale divisée par le nombre de processeurs). Puis l'algorithme s'exécute en cinq étapes :

1. Seuls les processeurs oisifs sont actifs. Ils sont énumérés, c'est à dire qu'un numéro d'ordre leur est affecté.

2. Chaque processeur oisif envoie son index (numéro de processeur) au processeur dont l'index correspond à son numéro d'ordre dans la liste des processeurs oisifs.

3. Les processeurs surchargés (qui ont une charge supérieure à la moyenne) deviennent actifs. A leur tour, ils sont énumérés.

4. Les processeurs actifs récupèrent l'adresse des processeurs inactifs par l'intermédiaire du processeur dont l'index correspond au numéro d'ordre dans la liste. Les processeurs dont l'index est petit servent donc de boîtes aux lettres. En effet, ils contiennent chacun l'adresse d'un processeur inactif.

5. La phase de communication termine l'exécution de l'algorithme.

Cet algorithme se caractérise par les points suivants: c'est un algorithme à double seuil. Nous pouvons remarquer que ce n'est pas forcément le processeur surchargé le plus proche d'un processeur oisif qui lui transmettra une partie de sa charge. Le critère utilisé pour l'appariement vise à choisir pour le i-ème processeur déchargé, le i-ème processeur surchargé.

Ces appariements sont généralement efficaces, car ils sont pertinents. Par contre, le coût en communication peut être élevé.

3.9 Système préemptif

Un stratégie d'équilibre de charge est définie comme *préemptive* si elle autorise la migration de tâches. Quand une tâche t a déjà commencé à s'exécuter sur un processeur P_i et que la stratégie décide de terminer son exécution sur un processeur P_j, il est nécessaire

de déplacer non seulement la tâche t sur P_j, mais aussi son environnement d'exécution. C'est pourquoi la migration de tâche est généralement plus coûteuse que le déplacement d'une tâche qui n'a pas commencé à s'exécuter. Les système préemptifs sont généralement des systèmes distribués où le multitâche est autorisé [Bar 89]. Quand chaque processeur est multitâche, il faut décider de déplacer une tâche avant le début de son exécution, c'est à dire à sa création. Dans le cas contraire, la tâche commence à s'exécuter sur le processeur où elle a été créée. Par la suite, le seul moyen de redistribuer cette tâche sera la migration. Dans le cas des systèmes préemptifs, les algorithmes source initiative sont plus efficaces que les algorithmes serveur initiative. En effet, il est difficile pour un processeur serveur, de trouver une tâche qui vient d'être créée sur un processeur distant. La plupart du temps, il sera obligé de migrer des tâches pour équilibrer la charge. Par contre, un processeur source peut facilement essayer de déplacer une tâche nouvellement créée avant de l'exécuter.

3.10 Qualités d'une stratégie

Afin de comparer différentes stratégies, il est possible de considérer plusieurs points qui permettent de définir le comportement général d'une stratégie [DFM 94, CLZ 92]:

stabilité: une stratégie stable évite de déplacer de façon inutile une tâche. Les algorithmes stables utilisent un nombre conséquent d'informations sur le système pour prendre une décision en toute connaissance de cause. Quand la connaissance du système est incomplète, il est possible de faire de mauvais choix.

Intrusion minimale: la stratégie doit perturber aussi peu que possible l'exécution de l'application. En particulier, elle doit éviter de générer des messages de contrôle et de consommer des ressources CPU.

Extensibilité: la stratégie doit conserver un bon comportement quand le nombre de processeurs augmente. Il est généralement admis que les stratégies distribuées sont facilement extensibles. Par contre, il est plus délicat d'étendre un algorithme centralisé à un nombre important de processeurs.

Efficacité: la stratégie doit permettre de résoudre plus rapidement le problème posé.

4 Les différentes méthodes d'analyse théorique pour le parallélisme

Nous avons présenté dans le chapitre consacré à l'algorithmique parallèle, les principales méthodes pour spécifier les algorithmes parallèles. Nous avons introduit les réseaux de Pétri, les algèbres de processus et les méthodes basées sur la logique. Cependant, l'ensemble de ces techniques d'analyse théorique est trop général pour permettre l'étude des algorithmes d'équilibre de charge.

C'est pourquoi de nouveaux outils ont été développés pour répondre à cette attente. Ainsi, les files d'attente et les méthodes matricielles permettent une analyse fine des stratégies d'équilibre de charge. Les files d'attente sont très utilisées dans les systèmes distribués où il y a une arrivée régulière des tâches. Les méthodes matricielles ont permis l'étude

de stratégies d'équilibre de charge SIMD et SPMD. Cependant, l'analyse des stratégies d'équilibre de charge des algorithmes MIMD reste encore délicate à mener.

4.1 Théorie des files d'attente

Le modèle des files d'attente a été proposé par le mathématicien Erlang au début du siècle pour étudier les réseaux téléphoniques. C'est une des premières méthodes utilisées pour modéliser les problèmes d'équilibre de charge pour les systèmes distribués. Ce modèle suppose que le système est soumis à une arrivée régulière de tâches infinie régie par une loi probabiliste. Le système distribué est caractérisé par le nombre de processeurs (*serveurs*) disponibles ainsi que par le nombre de *files d'attente*. Les tâches (*clients*) sont caractérisées par un *temps de service* déterminé à l'aide d'une loi statistique [Pui 97].

4.1.0.1 Notation des files d'attente : A/B/m/n/p/t. Les files d'attente sont complètement définies par les paramètres suivants :

A : distribution statistique des temps entre les arrivées de processus (tâches) dans le système. Ce paramètre est obligatoire. Pour l'étude des systèmes distribués, il prend généralement la valeur M, qui signifie que la loi appliquée est une *loi de Poisson.*

B : distribution statistique des temps de services. Ce paramètre est obligatoire, comme le paramètre A, il prend généralement la valeur M, (loi de Poisson).

m : nombre de serveurs.

n : longueur de la file d'attente. Par défaut, la capacité de la file d'attente est supposée infinie.

p : nombre de clients. Par défaut, le nombre de tâches qui arrivent dans le système est infini.

t : politique de gestion de la file d'attente. Par défaut, la règle appliquée est FIFO (*First In First Out*). Cependant, d'autres politiques peuvent être utilisées, par exemple BIFO (*Biggest In First Out*).

Système distribué sans équilibre de charge. Un système distribué soumis à une arrivée régulière de tâches peut être modélisé à l'aide d'une file d'attente de type M/M/1. En effet, un système composé de n processeurs se comporte comme n files d'attente M/M/1 indépendantes. Si on suppose que pendant un intervalle de temps suffisamment petit, il peut y avoir au plus une arrivée de tâche ou une consommation de tâche ou rien, alors il est possible de déterminer analytiquement le *nombre moyen de tâches* dans le système et le *temps de réponse moyen* notés respectivement Q et R [All 78] :

$$Q = \frac{\rho}{1-\rho} \text{ et } R = \frac{Q}{\lambda} = \frac{\frac{\rho}{\lambda}}{1-\rho} = \frac{\frac{1}{\mu}}{1-\rho} = \frac{s}{1-\rho}.$$

Où :

λ est le paramètre de la loi d'arrivée des tâches ;

μ est le paramètre de la loi de service d'une tâche ;

ρ est la charge $\left(\rho = \frac{\lambda}{\mu}\right)$;

s est le temps moyen de service $\left(s = \frac{1}{\mu}\right)$.

Ainsi, pour une file M/M/1, si la loi d'arrivée est égale à 0,9 ($\lambda = 0,9$), c'est à dire qu'une tâche entre en moyenne toutes les 0,9 secondes dans la file et que le temps moyen de service est de 1 seconde ($\mu = 1$, $s = 1$), alors le temps de réponse est égale à 10 secondes.

Système distribué avec un équilibrage parfait. Si on suppose qu'il est possible d'obtenir un rééquilibrage parfait de la charge pour un coût nul, le système peut être modélisé à l'aide d'un modèle M/M/n, où n est le nombre de processeurs. En effet, on suppose qu'il existe une unique file d'attente où arrivent toutes les tâches du système. Chaque processeur a accès à cette file unique et retire une tâche à chaque fois qu'il est disponible. Le temps de transfert d'une tâche de la file d'attente vers un processeur est supposé nul. Ainsi, pour un système comportant 4 processeurs (modélisé par le modèle M/M/4), si la charge ρ est égale à 0,9, le temps moyen de réponse est de 2,9 secondes.

Ces deux modèles (M/M/1 et M/M/n) sont très souvent utilisés pour comparer différentes stratégies d'équilibre de charge [ELZ 86a]. Ainsi, si les résultats obtenus par une stratégie est proche de ceux obtenus par le système M/M/n, alors la stratégie à un comportement satisfaisant. De même, si la stratégie à un comportement proche de M/M/1, l'algorithme peut être considéré comme peu performant.

4.2 Méthodes matricielles itératives

Les méthodes matricielles supposent que le réseau d'interconnexion est représenté par un graphe. Formellement, une matrice *d'adjacence* peut alors décrire ce graphe. Un graphe de n sommets est représenté par une matrice carrée M de dimension $n \times n$, où $\alpha_{ij} = 1$ s'il existe une arête entre les sommets i et j, $\alpha_{ij} = 0$ sinon. Les méthodes d'analyse matricielles utilisent cette même matrice pour modéliser les échanges entre les processeurs. En particulier, ces méthodes permettent l'analyse des algorithmes SIMD [HLM+ 90, Cyb 89]. Il est alors possible de montrer que la méthode *converge, i. e.* si la stratégie est appliquée un certain nombre de fois, le système distribué s'approche d'un état stable. Il est aussi possible de définir la *vitesse de convergence* qui est égale au nombre d'étapes nécessaires pour atteindre cet état stable. De même, les modèles matriciels autorisent la mesure théorique de certains indices de performance pour le modèle SPMD [FG 96].

4.2.1 Analyse matricielle d'un algorithme SIMD.

G. Cybenko a analysé l'algorithme de diffusion (*diffusion scheme*) pour une machine SIMD [Cyb 89]. La méthode matricielle développée par l'auteur permet l'étude de la convergence de la stratégie d'équilibre de charge.

4.2.1.1 L'algorithme de diffusion. Le système est soumis à une arrivée régulière de tâches. Toutes les tâches sont équivalentes et ont un temps de service identique. C'est un algorithme distribué qui applique une stratégie locale de répartition de charge à intervalles réguliers. Chaque processeur est voisin d'un nombre fini de processeurs. Le principe de l'algorithme est le suivant. À chaque fois que la méthode de diffusion est appliquée, chaque processeur équilibre sa charge avec ses voisins. Si P_i est voisin de P_j, alors $\alpha_{ij}\left(w_j^{(t)} - w_i^{(t)}\right)$ charges sont envoyées par P_i en direction de P_j (où $w_i^{(t)}$ et $w_j^{(t)}$ sont les charges respectives des processeurs P_i et P_j) ; si cette valeur est négative, c'est P_j qui envoie des tâches à P_i. La valeur de α_{ij} est comprise entre 0 et 1. L'auteur a montré que cet algorithme converge si les entrées et sorties du système sont masquées.

4.2.1.2 Analyse de l'algorithme de diffusion. Afin de pouvoir présenter cette étude, nous introduisons les notations suivantes :

- le système est composé de n processeurs numérotés de 0 à $n-1$;
- $w_i^{(t)}$ représente la charge du processeur i à l'instant t ;
- la charge du système à l'instant t est représentée par le *vecteur de charge*

$$w^{(t)} = \begin{pmatrix} w_0^{(t)} \\ \vdots \\ w_i^{(t)} \\ \vdots \\ w_{n-1}^{(t)} \end{pmatrix}.$$

- La matrice $M = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots \\ \alpha_{10} & \alpha_{11} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$ de dimensions $n \times n$ modélise le comportement de la stratégie. M représente les échanges qui ont lieu entre les différents processeurs du système.
- α_{ij} représente l'échange de charge entre les processeurs i et j.

En appliquant la méthode de diffusion, la charge du processeur i à l'instant $t+1$ est déterminée par l'équation suivante :

$$w_i^{(t+1)} = w_i^{(t)} + \sum_j \alpha_{ij}\left(w_j^{(t)} - w_i^{(t)}\right) + \eta_i^{t+1} - c.$$

Où η_i^{t+1} représente les tâches arrivées à l'instant $t+1$ et c le nombre de tâches consommées entres les instants t et $t+1$. Comme toutes les tâches sont équivalentes et que l'algorithme est appliqué à intervalle régulier, la valeur de c est une constante. Le but de cette étude est de montrer que la stratégie d'équilibre de charge converge vers la *distribution uniforme* (le vecteur de charge dont toutes les entrées sont égales à $\frac{\sum_i w_i}{n}$). Pour étudier cette convergence, les entrées et sorties sont cachées, l'auteur obtient donc :

$$w_i^{(t+1)} = \left(1 - \sum_j \alpha_{ij}\right) w_i^{(t)} + \sum_j \alpha_{ij} w_j^{(t)}.$$

Cette équation met en évidence les deux contraintes suivantes:

1. $\alpha_{ij} \geq 0, \forall (i,j)$;
2. $1 - \sum_i \alpha_{ij} \geq 0, \forall i$.

Il est donc possible de représenter le comportement du système distribué à l'aide de l'équation matricielle suivante:

$$w^{(t+1)} = Mw^{(t)}, \text{ où } m_{ij} = \begin{cases} \alpha_{ij} & \text{si } i \neq j \\ 1 - \sum_k \alpha_{ik} & \text{si } i = j \end{cases}$$

Définition 2.2 *Une matrice est dite non négative, si et seulement si chacune de ses entrées est non négative.*

Définition 2.3 *Une matrice non négative dont toutes les sommes des lignes sont égales à 1 est dite stochastique.*

Définition 2.4 *Une matrice stochastique dont les sommes des colonnes sont égales à 1 est dite doublement stochastique.*

Par définition de la matrice M, toutes ses entrées sont positives et les sommes des lignes et colonnes sont toutes égales à 1, M est donc doublement stochastique. Comme les sommes des lignes sont égales à 1, la charge globale du système est conservée à tout instant.

G. Cybenko montre alors que l'algorithme de diffusion converge sous certaines conditions.

Théorème 2.3 *L'algorithme de diffusion converge vers la distribution uniforme si et seulement le graphe induit est connexe et si au moins une des deux conditions suivantes est vérifiée:*

1. $\exists j | \left(1 - \sum_i \alpha_{ij}\right) > 0$;
2. *le graphe induit n'est pas biparti.*

Preuve : le *graphe induit* associé à une matrice de diffusion M est essentiellement le graphe de communication où quelques arêtes ont été supprimées. Les arêtes supprimées correspondent aux liens de communication qui ne sont pas utilisées par la stratégie de diffusion représentée par M. Un graphe est dit *biparti* (ou *2-stable*) si l'on peut diviser les nœuds en deux ensembles distincts dans lesquels il n'existe aucune arête reliant deux nœuds d'un même ensemble.

La démonstration de ce résultat s'appuie sur la théorie de *Perron-Frobenius* [Var 62]. En particulier, pour que M converge, il est nécessaire que -1 ne soit pas une valeur propre de M. L'auteur montre alors que pour que -1 ne soit pas une valeur propre de M, il est

nécessaire est suffisant qu'une des deux propriétés énoncées précédemment soit vérifiée. □

G. Cybenko établit enfin les vitesses de convergence de son algorithme pour un réseau de communication de type hypercube (voir paragraphe 2.2.2.2 du chapitre 1).

Par ailleurs, Hosseini *et al.* proposent une méthode matricielle appliquée à la notion de graphe coloré [HLM+ 90]. Le graphe est coloré avec un minimum de couleurs afin d'éviter que deux arêtes issues d'un même sommet soient de la même couleur. Périodiquement, toutes les couleurs sont activées une à une. Un processeur ne peut communiquer qu'avec un processeur voisin dont le lien de communication est de couleur c, si et seulement si c est la couleur active. La notion de couleur permet d'ordonnancer et de synchroniser les communications.

La matrice M qui modélise le comportement de la stratégie d'équilibre de charge est alors le produit des matrices M_1 à M_k où k est le nombre de couleurs et M_i modélise les échanges quand la couleur i est active. Les auteurs étendent les résultats obtenus par Cybenko pour l'hypercube, à un graphe de communication quelconque.

4.2.2 Analyse matricielle d'un algorithme SPMD.

M. A. Frankin et V. Govindan ont développé une méthode matricielle afin d'analyser le comportement d'algorithmes SPMD de type source initiative [FG 96]. Les algorithmes d'équilibre de charge étudiés utilisent des techniques d'appariement différentes.

La solution proposée par G. Cybenko ne peut être appliquée dans ce cas. En effet, le modèle qu'a proposé l'auteur est uniquement valide quand la stratégie d'équilibre de charge n'est pas adaptative. Dans ce cas, l'algorithme suit toujours le même comportement quelle que soit la situation, ce qui conduit à une représentation de la stratégie par une matrice invariable dans le temps.

La solution proposée par M. A. Frankin et V. Govindan est plus générale. La matrice qui représente la stratégie d'équilibre de charge varie dans le temps. Cependant, ce modèle suppose que l'application est composée de plusieurs phases de calcul et d'équilibre de charge. Il n'est alors pas possible à un processeur d'initier une phase d'équilibre de charge si l'étape courante est une étape de calcul.

4.2.2.1 Présentation de la méthode matricielle. Les auteurs supposent que deux hypothèses sont vérifiées :

1. le système comprend P processeurs, notés $P_1, ..., P_p$. La puissance de chaque processeur est variable, C_i est la puissance de P_i. Les auteurs supposent que $C_1 > C_2 > ... > C_p$.

2. L'application est divisée en un nombre important de tâches toutes équivalentes.

L'algorithme d'équilibre de charge est exécuté à des instants dénotés k ($k = 1, 2, 3, ...$). Il peut être appliqué à intervalles réguliers, ou à la fin de chaque itération dans le cas d'une application itérative. Tous les processeurs sont synchronisés avant l'application de la stratégie. En utilisant les notations introduites précédemment, le modèle dynamique associé au processeur P_i peut être représenté ainsi :

$w_i^{(k+1)} = w_i^{(k)}$ - les tâches envoyées par P_i (équilibre de charge)
+ les tâches reçues par P_i (équilibre de charge)
+ les tâches arrivées pendant l'intervalle $[k,\ k+1]$ (dynamique de l'application)
- les tâches consommées pendant l'intervalle $[k,\ k+1]$ (dynamique de l'application)

La graphe d'interconnexion est représenté par la matrice H. Quand une phase d'équilibre de charge est déclenchée, chaque processeur est libre de participer ou non à cette nouvelle phase. Chaque processeur compare sa charge à un seuil unique afin de connaître son état (voir paragraphe 3.7.2.1 du chapitre 2). La participation des processeurs à l'étape k est représentée par la matrice carrée $A^{(k)}$ de dimension p. Un élément de la diagonale est à 1, si le processeur correspondant participe à la phase d'équilibre de charge, tous les autres éléments sont à 0. La matrice $M^{(k)}$ modélise les échanges entre les processeurs. M_{ij} est la fraction de ses tâches qu'envoie P_i à P_j (*i. e.* P_i envoie $M_{ij}^{(k)}.w_i^{(k)}$ tâches à P_j). Nous pouvons remarquer que la matrice $A^{(k)}$ peut facilement être incorporée dans $M^{(k)}$.

Le modèle dynamique associé au processeur P_i peut donc être décrit ainsi :

$$w_i^{(k+1)} = w_i^{(k)} - \sum_{j \neq i} M_{ij}^{(k)} A_{ii}^{(k)} w_i^{(k)} + \sum_{i \neq j} M_{ji}^{(k)} A_{jj}^{(k)} w_j^{(k)} + \lambda_i^{(k)} - \mu_i^{(k)}.$$

Où $\lambda^{(k)}$ et $\mu_i^{(k)}$ représentent les arrivées et départs des tâches sur chaque processeur. Les deux sommes représentent les échanges de tâches entre les processeurs impliqués par la stratégie d'équilibre de charge. La première somme correspond aux envois réalisés par chaque processeur et la seconde aux réceptions. En définissant $M_{ii}^{(k)} = -\sum_{j \neq i} M_{ij}^{(k)}$, le modèle est représenté par :

$$w_i^{(k+1)} = w_i^{(k)} + \sum_{j} M_{ji}^{(k)} A_{jj}^{(k)} w_j^{(k)} + \lambda_i^{(k)} - \mu_i^{(k)}.$$

4.2.2.2 Application de la méthode. Les auteurs comparent trois stratégies de type source initiative dans le cadre d'une application de type *n-body problem*. Il s'agit de calculer la position d'un ensemble de particules en fonction de la position et de la masse de chacune d'entre elle. L'espace 2D est divisé en autant de régions qu'il y a de processeurs. Chaque processeur calcule la position des particules présentes dans sa région. Si une particule sort de la région dont il s'occupe, il la transmet au processeur responsable de la région dans laquelle elle est entrée.

Les paramètres f_{task} (le temps de calcul de chaque tâche, ici le calcul de la position d'une particule), λ (les arrivées de tâches) et μ (les départs de tâches) sont déterminés expérimentalement pour ce problème. Le modèle matriciel est alors appliqué pour comparer la stratégie aveugle, la méthode de diffusion et un algorithme utilisant un niveau d'information global.

L'étude théorique présentée par les auteurs leur permet de conclure que la stratégie par diffusion est la plus efficace pour le problème qu'ils avaient à résoudre. Cette étude est de plus validée expérimentalement, ce qui permet de s'assurer de la validité du modèle matriciel proposé.

5 Conclusion

L'équilibre de charge est un élément essentiel d'une application parallèle. Il est insuffisant de proposer un découpage d'un problème en un ensemble de tâches pour obtenir une résolution en parallèle efficace. En effet, si la répartition des tâches est inadaptée, le gain en performance sera pratiquement nul. Nous avons vu qu'il est possible d'aborder le problème de l'équilibre de charge d'un point de vue centralisé ou distribué. Chacune des méthodes présente des avantages et des inconvénients. Les stratégies centralisées apportent une solution simple au problème posé en limitant le nombre de communications engendrées. Cependant, il est difficile d'étendre ces algorithmes quand le nombre de processeurs est conséquent. Les méthodes distribuées sont plus délicates à implémenter mais sont facilement extensibles.

Afin de proposer une stratégie d'équilibre de charge efficace, il est nécessaire que cette méthode possède plusieurs qualités. Elle doit être stable, simple à mettre en œuvre, éviter de surcharger inutilement le réseau de communication. Dans la mesure du possible, elle doit être indépendante de l'application et de la machine cible et en particulier du nombre de processeurs.

Il est difficile de comparer les différentes stratégies dans le cas général. En effet, les hypothèses des auteurs sont très différentes suivant la machine cible choisie, l'application à paralléliser, le modèle de programmation, ... C'est pourquoi afin de comparer les performances effectives des différentes stratégies, il est nécessaire de choisir une application ainsi qu'une machine cible afin d'harmoniser les hypothèses de départ. M. Willebeek-Lemair et A. P. Reeves ont comparé différents algorithmes pour la résolution d'un problème de *Branch and Bound* [WR 93]. La machine cible comporte 32 processeurs et la synchronisation est peu coûteuse. La migration de tâche n'est pas permise. Les auteurs comparent les performances de cinq algorithmes : un algorithme source initiative, un algorithme serveur initiative, la méthode semi-distribuée HBM, le modèle du gradient et, la méthode d'échange dimensionnelle. Il est à noter qu'aucun algorithme centralisé n'est présenté dans cette étude. La méthode d'échange dimensionnelle semble obtenir les meilleurs résultats. Mais, les auteurs rappellent que la synchronisation n'est pas coûteuse, ce qui peut expliquer ce résultat. Pour des systèmes comportant un nombre plus important de processeurs, il semble que ce soit la stratégie serveur initiative qui soit la mieux adaptée.

Les outils théoriques présentés dans la dernière partie de ce chapitre, permettent de comparer et valider les algorithmes d'équilibre de charge avant de les implanter. De plus, il est possible d'évaluer le comportement de ces stratégies dans des situations très variées. En effet, il est possible de considérer les algorithmes quand la charge globale du système est faible, ou bien quand la répartition initiale est relativement satisfaisante.

Cependant l'ensemble des méthodes proposées dans ce chapitre, n'autorise pas l'analyse d'un algorithme d'équilibre de charge dynamique pour une application MIMD. En effet, la solution des files d'attente ne peut pas toujours être appliquée car il n'y a pas forcément d'arrivée de tâches. De même, les solutions matricielles existantes ne sont pas assez générales. La méthode de G. Cybenko suppose que le comportement de la stratégie d'équilibre de charge est invariable dans le temps est que le modèle est synchrone. Enfin, la solution proposée par M. A. Frankin et V. Govindan s'applique uniquement à un modèle SPMD.

Les difficultés liées à l'analyse d'une stratégie MIMD sont nombreuses. En effet, le comportement de la stratégie varie en fonction de l'état du système. De plus certains

processeurs cherchent à équilibrer leurs charges alors que d'autres calculent des tâches. L'ensemble de ces difficultés rend difficile la modélisation d'un système MIMD dont la charge est répartie à l'aide d'un stratégie d'équilibre de charge dynamique.

Dans le chapitre suivant, nous nous proposons d'établir une classification des différentes stratégies d'équilibre de charge possibles dans le cadre d'une famille d'applications que nous définirons au préalable. Nous développons un modèle matriciel MIMD adapté à notre problème et nous proposons cinq stratégies d'équilibre de charge dynamique. Nous étudions les propriétés de chaque stratégie et nous les comparons à l'aide d'une étude statistique.

Chapitre 3

Équilibre de charge dynamique pour des applications FTII

1 Introduction

Nous avons vu au chapitre précédent qu'il est possible de proposer des stratégies d'équilibre de charge permettant de réduire le temps d'exécution d'une implantation parallèle d'une application. Une classification simple des méthodes d'équilibre de charge dynamique a été présentée. Dans ce chapitre, nous proposons un ensemble cohérent d'algorithmes permettant de couvrir cette classification arborescente dans le cadre d'une famille d'applications que nous appelons les applications *FTII*[1].

Dans la première partie de ce chapitre, nous présentons de façon formelle la notion d'application FTII ainsi que celle d'équilibrage de charge associé.

Afin de valider théoriquement les mesures de performance de nos différents algorithmes d'équilibre de charge, nous proposons un nouveau modèle de type matriciel orienté MIMD. Ainsi, nous consacrerons une grande partie du chapitre à la présentation de chaque algorithme d'équilibre de charge suivi de sa validation relative au modèle matriciel proposé.

2 Présentation formelle d'une application FTII

2.1 Définitions

Une application FTII est une application à nombre *Fini de Tâches Indépendantes et Irrégulières*: c'est à dire, une application dont le travail peut être divisé en un nombre fini d'éléments (appelés *tâches)*. Toutes les tâches sont *indépendantes*: une tâche ne peut faire aucune hypothèse quant à l'exécution d'une autre tâche. Par conséquent, il n'y a pas de communication entre les tâches ; l'ordonnancement des tâches est réalisé sans règles de précédence et aucune synchronisation n'est nécessaire. De plus, chaque tâche est caractérisée par son temps d'exécution séquentiel qui est imprévisible. Enfin, le même algorithme est appliqué pour calculer toutes les tâches et donc deux tâches sont différenciées par les données qu'elles traitent.

Comme le temps de chaque tâche est imprévisible, nous montrerons qu'une répartition dynamique des tâches est nécessaire dans le but d'obtenir une efficacité pratique de la

1. application à nombre Fini de Tâches Indépendantes et Irrégulières

parallélisation d'une application FTII.

Afin d'établir une définition plus précise d'une application FTII, nous proposons de définir la notion de tâche de la manière suivante :

Définition 3.1 *(tâche irrégulière)*
Soit $\mathcal{A}$ une application qui s'applique sur un domaine $\mathcal{D}_\mathcal{A}$. Le domaine $\mathcal{D}_\mathcal{A}$ représente l'ensemble des données à traiter par $\mathcal{A}$. Une tâche t de $\mathcal{A}$ définit un ensemble d'opérations élémentaires que doit accomplir l'application $\mathcal{A}$ sur un sous-domaine $\mathcal{D}_t$ de $\mathcal{D}_\mathcal{A}$ ($\mathcal{D}_t \subset \mathcal{D}_\mathcal{A}$). Par définition, la tâche t est irrégulière s'il est impossible de prévoir le coût associé à t (temps de calcul) avant le début de l'exécution.

Remarque : la notion d'irrégularité d'une tâche que nous adoptons est légèrement différente de celle généralement acceptée dans le cadre de la parallélisation automatique. Une tâche est souvent considérée irrégulière si le graphe de communication associé est particulièrement difficile à construire ou bien s'il est très différent des graphes «classiques». Dans notre cas, l'irrégularité d'une tâche caractérise le fait qu'il est très difficile d'obtenir une estimation du temps d'exécution d'une tâche avant de la calculer.

Exemple 3.1 : calcul d'une intégrale simple.

Supposons, que l'application $\mathcal{A}$ détermine une valeur approchée de l'intégrale $\int_a^b f(x)dx$. où a et b sont réels et $f(x)$ est une fonction continue sur l'intervalle $[a, b]$. Pour déterminer ce résultat, l'application approche l'aire représentée par l'intégrale par le calcul de l'aire de n rectangles (voir figure 3.1). Le domaine de cette application est défini par l'intervalle $\mathcal{D}_\mathcal{A} = [a, b]$.

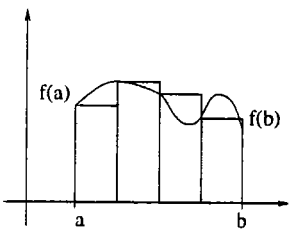


FIG. 3.1 –*calcul approché de $\int_a^b f(x)dx$. à l'aide du calcul de l'aire de 4 rectangles.*

Il est possible de définir la notion de tâche pour l'application $\mathcal{A}$ ainsi : le problème est composé de n tâches (le calcul de l'aire des n rectangles) et le domaine de chaque tâche est un intervalle $[a_t, b_t] \subset [a, b]$, $\forall t \in [0,\ n-1]$.

Remarque : si une application $\mathcal{A}$ qui s'applique sur un domaine $\mathcal{D}_\mathcal{A}$, est partagé en n tâches (numérotées de 0 à $n-1$) de domaines $\mathcal{D}_t, (0 \leq t \leq n-1)$, il est nécessaire que le domaine défini par $\bigcup_{t=0}^{n-1} \mathcal{D}_t$ soit égal au domaine $\mathcal{D}_\mathcal{A}$. En effet, dans le cas contraire, une partie des calculs accomplis par l'application $\mathcal{A}$ n'est pas présente au niveau de la partition réalisée par les n tâches.

De plus, si on suppose que les domaines des tâches sont disjoints deux à deux ($\mathcal{D}_t \cap \mathcal{D}_u = \emptyset, \forall t,\ u,\ t \neq u$), le temps d'exécution de l'algorithme t_{seq} est alors égal à $\sum_{t=0}^{n-1} t_{cal}(t)$ où $t_{cal}(t)$ est le temps d'exécution de la tâche t.

Afin d'obtenir une parallélisation efficace des applications FTII, nous supposons qu'une fonction injective f_d permet d'associer à chaque tâche[2], un domaine. En effet, pour éviter de surcharger le réseau d'interconnexion lorsqu'une tâche est déplacée, il est préférable de transmettre l'identifiant d'une tâche plutôt que son domaine dont la représentation est généralement plus longue (en nombre de mots mémoire). Nous discuterons plus en avant cet aspect au paragraphe 2.2.3.

Définition 3.2 *(application FTII)*
Une application $\mathcal{A}$ (de domaine $\mathcal{D}_A$) est une application FTII si elle vérifie les hypothèses suivantes :

1. *Le travail de $\mathcal{A}$ peut être divisé en un nombre fini n de tâches indépendantes numérotées de 0 à $n-1$ avant le début de l'exécution. Le même ensemble d'opérations élémentaires est appliqué pour traiter chaque tâche.*

2. *Il existe une fonction injective f_d telle que :*

$$\begin{aligned} f_d : [0, n-1] \in \mathbb{N} &\longrightarrow \mathcal{D}_A \\ t &\longmapsto \mathcal{D}_t \end{aligned}$$

3. *Les domaines des tâches sont disjoints deux à deux : $\mathcal{D}_t \cap \mathcal{D}_u = \emptyset$ $(t \neq u)$.*

4. *Le temps de chaque tâche $t_{cal}(t)$ est imprévisible $(t \in [0,\ n-1])$.*

Exemple 3.2 : calcul d'une intégrale simple (suite).
Nous pouvons exprimer la fonction f_d pour l'application qui détermine une valeur approchée de l'intégrale (voir exemple 3.1). La fonction f_d qui détermine le domaine de chaque tâche en fonction de son identifiant est :

$$\begin{aligned} f_d : [0, n-1] \in \mathbb{N} &\longrightarrow [a, b] \\ t &\longmapsto \left[a + \frac{b-a}{n} \times t, a + \frac{b-a}{n} \times (t+1)\right] \end{aligned}$$

Bien qu'il soit possible de construire un ensemble de tâches indépendantes et de définir la fonction f_d, cette application de calcul d'une intégrale simple n'est pas FTII. En effet, le temps de calcul d'une tâche est constant. Il est donc inutile de proposer une résolution dynamique du problème de l'équilibre de charge. Pour obtenir une parallélisation efficace de ce type de problème, il suffit de confier le calcul de $\frac{n}{p}$ tâches à chaque processeur.

Nous présenterons au chapitre suivant, en plus du problème de lancer de rayons étudié au chapitre 5, le problème de l'ensemble de Mandelbrot et deux exemples concrets d'applications FTII issues de la CAO.

2. représentée par un identifiant unique de type entier.

2.2 Accélération et équilibre de charge dynamique

2.2.1 Accélération théorique

À l'aide de la définition des applications FTII, nous pouvons exprimer l'accélération théorique obtenue par une parallélisation de l'application FTII $\mathcal{A}$ pour p processeurs. Avant d'étudier ce point, nous introduisons la notion de travail pour une application.

Définition 3.3 *(travail d'une application)*
Le travail fourni par une application $\mathcal{A}$ (noté $W_{\mathcal{A}}$) est égal à la somme de toutes les opérations élémentaires exécutées par $\mathcal{A}$. Cette mesure peut être exprimée en nombre d'opérations ou bien en unités de temps.

Proposition 3.1 *Si une application $\mathcal{A}$ de type FTII (composée de n tâches) est parallélisée et la répartition de charge a un coût nul, le temps d'exécution parallèle pour p processeurs est proche de l'optimal :*

$$Tpar_{\mathcal{A}}(p) = \max\left(\frac{t_{seq}}{p}, \max_{t=0}^{n-1} t_{cal}(t)\right).$$

Preuve : pour montrer ce résultat, nous exprimons le travail réalisé par l'algorithme séquentiel et par l'algorithme parallèle quand la répartition de la charge a un coût négligeable.

Comme les domaines des tâches sont disjoints deux à deux, il n'y a pas de calculs redondants, c'est à dire que chaque opération élémentaire n'est réalisée qu'une seule fois par l'algorithme. Donc, le travail de l'algorithme séquentiel est égal à :

$$W_{seq} = \sum_{t=0}^{n-1} t_{cal}(t) = t_{seq}.$$

D'après la définition 3.2, le calcul de chaque tâche est indépendant (il n'y a pas de communications entre les tâches) et les domaines des tâches sont disjoints. Quand la répartition de la charge a un coût nul, le travail fourni par l'algorithme parallèle est :

$$W_{par} = \sum_{t=0}^{n-1} t_{cal}(t).$$

Comme $Tpar_{\mathcal{A}}(p) = \max\left(\frac{W_{par}}{p}, \max_{t=0}^{n-1} t_{cal}(t)\right)$ et $W_{par} = W_{seq}$, le temps d'exécution parallèle est égale à $\max\left(\frac{t_{seq}}{p}, \max_{t=0}^{n-1} t_{cal}(t)\right)$. Ainsi, le temps d'exécution parallèle théorique miminum est, soit égal à la partie entière du temps d'exécution parallèle optimal $\left(\frac{t_{seq}}{p}\right)$, soit au temps d'exécution de la tâche la plus coûteuse $\left(\max_{t=0}^{n-1} t_{cal}(t)\right)$. □

D'après la proposition précédente, nous pouvons espérer une parallélisation efficace des applications FTII. L'accélération théorique maximale est proche de l'accélération optimale (p pour p processeurs). L'écart entre l'accélération théorique de l'application FTII et l'accélération optimale est déterminé par le temps de calcul de la tâche la plus coûteuse.

Ainsi, la granularité de l'application est un facteur important qui influe directement sur l'efficacité de la solution parallèle. En pratique, si la granularité n'est pas adaptée, il peut en résulter une inefficacité de l'application parallèle. Par exemple, supposons qu'une application composée de 4 tâches (dont les temps d'exécution sont 7, 16, 12 et 8 secondes) est exécutée sur une machine comportant 2 processeurs. D'après le théorème précédent, nous pouvons espérer un temps d'exécution de 20 secondes, or en pratique, le temps d'exécution ne peut être inférieur à 23 secondes car la migration de tâche n'est pas autorisée. Si l'utilisateur souhaite une accélération supérieure, il devra proposer une granularité plus fine pour l'atteindre.

Pour obtenir une exécution parallèle efficace des applications FTII, il est non seulement nécessaire de choisir un grain en adéquation avec l'application cible, mais aussi de proposer des algorithmes d'équilibre de charge adaptés. Ils doivent répartir efficacement les tâches entre les processeurs et éviter de surcharger le réseau de communication car ce sont les communications induites par les stratégies d'équilibre de charge dynamique qui détermineront en grande partie l'accélération pratique obtenue pour les applications FTII.

2.2.2 Équilibre de charge dynamique

Généralement, l'objectif de l'algorithme d'équilibre de charge est de répartir les différentes tâches sur l'ensemble des processeurs de manière à ce que tous les processeurs soient constamment actifs durant l'exécution de l'application. C'est donc une fonction d'optimisation de la répartition de la charge. Il existe dans la littérature plusieurs approches à ce problème. Ainsi, il est possible de considérer ce problème comme un problème de placement (satisfaction de contraintes, recherche opérationnelle). Cependant, compte tenu des hypothèses de FTII, l'équilibre de charge ne peut être traité de manière statique. En effet, le temps de chaque tâche est imprévisible, il est donc difficile de construire le graphe des tâches nécessaire à une solution statique. Ce qui nous motive donc à proposer des algorithmes d'équilibre de charge dynamique.

Définition 3.4 *(équilibre de charge FTII)*

Soit une application $\mathcal{A}$ de type FTII composée de n tâches. L'algorithme d'équilibre de charge dynamique appliqué par $\mathcal{A}$ est une fonction vérifiant à tout instant :

$$\forall i \,|\, 0 \leq i \leq p-1,\ w_i > 0,\ \textit{si}\ W \geq p.$$

Où w_i est la charge du processeur P_i définie par le nombre de tâches que P_i doit calculer. La charge globale du système W est égale à la somme des charges de tous les processeurs :
$W = \sum_{i=0}^{p-1} w_i$ *où p est le nombre de processeurs.*

Remarque : d'après cette définition, la stratégie d'équilibre de charge doit s'assurer que tous les processeurs sont actifs ($w_i > 0, \forall i,\ 0 \leq i \leq p-1$) tant que la charge globale du système est supérieure à p. En effet, si $W < p$ au plus W processeurs sont actifs, car le nombre de tâches est insuffisant et une tâche est indivisible.

Nous supposons que la migration de tâche n'est pas autorisée ; si une tâche est en cours d'exécution sur P_i, nous nous interdisons de déplacer cette tâche en direction d'un

processeur P_j ($i \neq j$). Si la charge d'un processeur est égale à 0, ce processeur est considéré comme *déchargé*[3] . De même, si la charge d'un processeur est supérieure ou égale à 2, ce processeur est susceptible de transmettre au moins une tâche en direction d'un processeur déchargé, il est donc considéré comme *surchargé*.

Si le processeur P_i est déchargé, la stratégie d'équilibre de charge doit déplacer au minimum une tâche t provenant d'un processeur surchargé P_j vers P_i. Ainsi, l'identifiant de t est transmis à P_i et la charge de P_j est modifiée en conséquence. Le processeur P_i est capable de réaliser la correspondance entre le numéro d'une tâche et son domaine (l'ensemble des données qu'elle représente) grâce à la fonction f_d définie pour toute application FTII. C'est pourquoi, chaque processeur doit connaître la fonction f_d pour permettre une gestion dynamique de la charge.

2.2.3 Gestion des accès aux données

Dans le cadre des applications FTII, nous supposons qu'il est uniquement nécessaire de transmettre l'identifiant d'une tâche t quand elle est déplacée du processeur P_i vers le processeur P_j ($i \neq j$). À l'aide de la fonction injective f_d, le processeur P_j est en mesure d'effectuer les calculs associés à cette tâche. Cependant, si les données sont distribuées, le calcul de la tâche t peut nécessiter des données directement accessibles à P_i mais dont P_j ne dispose pas.

Exemple 3.3 : calcul d'une intégrale simple (suite et fin).

Si l'application $\mathcal{A}$ détermine une valeur approchée de l'intégrale $\int_a^b f(x).dx$ où $f(x)$ est une fonction continue définie par morceau sur l'intervalle $[a,\ b]$, il est inutile au processeur P_i chargé du calcul de la tâche t de connaître la définition de la fonction $f(x)$ sur tout l'intervalle $[a,\ b]$. En effet, il suffit que P_i soit en mesure d'évaluer la fonction $f(x)$ sur l'intervalle $\left[a + \frac{b-a}{n} \times t, a + \frac{b-a}{n} \times (t+1)\right]$ où n est le nombre de tâches.

Cependant, si P_j envoie une tâche u à P_i ($u \neq t$ et $i \neq j$), il est nécessaire que P_i connaisse aussi la définition de $f(x)$ sur l'intervalle $\left[a + \frac{b-a}{n} \times u, a + \frac{b-a}{n} \times (u+1)\right]$. C'est pourquoi, il est souhaitable de mettre en place un mécanisme adapté à l'application pour permettre à tout processeur d'accéder à toutes les données de l'application.

Si la machine parallèle possède à la fois une mémoire partagée accessible à l'ensemble des processeurs et une mémoire locale pour chaque processeur, nous pouvons placer les données de l'application FTII à paralléliser au niveau de la mémoire partagée. Comme toutes les données du problème sont accessibles à tous les processeurs, aucune communication ne sera nécessaire quand une tâche est déplacée. Les calculs induits par une tâche sont alors effectués en utilisant la mémoire locale du processeur responsable de l'exécution de cette tâche.

Quand la machine ne dispose pas d'une mémoire partagée mais que la mémoire locale de chaque processeur est importante, une solution simple qui peut être adoptée est de dupliquer toutes les données dans les mémoires locales des processeurs. Comme dans le cas d'une mémoire partagée, le déplacement d'une tâche ne posera plus de problème d'accès aux données.

3. nous employons aussi indifféremment les termes inactif ou oisif comme synonymes de déchargé.

Par contre, s'il n'est pas réaliste d'envisager la duplication des données, deux solutions sont envisageables :

- accès aux données par échange de messages,
- utilisation d'une mémoire virtuelle partagée.

Dans les deux cas, des communications supplémentaires sont nécessaires quand une tâche est déplacée. L'utilisation de la mémoire virtuelle partagée cache ces communications à l'utilisateur qui n'est alors pas responsable de la gestion de l'accès aux données. Cependant, ces communications pénalisent la stratégie d'équilibre de charge car le déplacement d'une tâche entraîne un surcoût en communication pour recopier les données nécessaires à l'exécution d'une tâche. Si les données nécessaires à une tâche peuvent être déterminées avant son exécution, il est néanmoins possible d'estimer *a priori* ce surcoût. Cette information est importante, en particulier pour permettre une analyse des performances d'une stratégie d'équilibre de charge dynamique à l'aide d'un modèle matriciel.

Quand il est impossible de prévoir les données nécessaires à une tâche, le problème est plus délicat. En effet, il est difficile d'estimer le surcoût du déplacement d'une tâche. Dans certains cas il sera très lourd, dans un autre il pourra même être négatif : le processeur destinataire d'une tâche déplacée a des informations nécessaires à cette tâche que le processeur expéditeur ne possède pas.

2.2.4 Indépendances des tâches et efficacité des applications FTII

Un autre aspect important des applications FTII et l'hypothèse d'indépendance des tâches et plus particulièrement le fait qu'il n'y a pas recouvrement des calculs au niveau de deux tâches distinctes : $\mathcal{D}_t \cap \mathcal{D}_u = \emptyset$ $(t \neq u)$.

Si cette hypothèse n'est pas vérifiée, l'utilisateur a le choix entre deux solutions : il choisit de ne pas effectuer de calculs redondants ou bien il autorise cette redondance.

Si les calculs redondants ne sont pas autorisés bien que les domaines des tâches ne soient pas disjoints, nous pouvons envisager plusieurs techniques pour résoudre ce problème. La plus simple est de disposer d'une mémoire globale où les résultats communs à plusieurs tâches sont déposés par la première tâche qui a fait les calculs (principe de la *boîte aux lettres*). Une tâche qui a besoin d'un résultat partagé vérifie si ce calcul a déjà été effectué au niveau de la mémoire globale. Si le résultat est présent, elle le récupère, dans le cas contraire elle effectue le travail et dépose le résultat dans la mémoire partagée.

Dans la mesure où il n'est pas possible d'utiliser une mémoire partagée (même virtuelle), il est possible de communiquer les résultats communs à plusieurs tâches aux différents processeurs responsables de ces tâches. Par exemple, si la tâche t est calculée avant la tâche u par le processeur P_i et P_j est responsable de la tâche u ($t \neq u$ et $i \neq j$), quand P_i termine l'exécution de la tâche t, il transmet les calculs communs à la tâche u à P_j. Quand P_j calcule la tâche u, il dispose déjà des résultats partagés avec la tâche t et il évite donc de les calculer à nouveau. Cette solution suppose qu'à tout instant P_i connaît le processeur responsable de la tâche u, ce qui peut être difficile à réaliser quand une tâche est déplacée du fait de la stratégie d'équilibre de charge dynamique.

Quand nous choisissons d'autoriser les calculs redondants, l'estimation de l'accélération théorique que nous avons menée au paragraphe 2.2.1 n'est plus satisfaisante. En effet, pour montrer ce résultat, nous supposons que l'hypothèse $\mathcal{D}_t \cap \mathcal{D}_u = \emptyset$ $(t \neq u)$ est

vérifiée. Dans le cas contraire, nous pouvons néanmoins estimer l'accélération théorique de la solution parallèle si l'utilisateur est en mesure d'estimer l'importance des calculs redondants. En effet, si la travail de la solution parallèle est, par exemple, 10% supérieur au travail fourni par la solution séquentielle, nous pouvons estimer que l'accélération obtenue sera majorée par $\frac{9}{10}p$. Cependant, en pratique cette solution peut être plus performante que la précédente s'il est plus rapide d'effectuer les calculs redondants plutôt que de les communiquer.

3 Un modèle matriciel pour les algorithmes MIMD

Afin de comparer les différentes stratégies que nous allons proposer pour les applications FTII, nous développons une méthode d'analyse matricielle qui s'applique au modèle MIMD. Après avoir présenté formellement ce modèle, nous l'appliquerons à tous nos algorithmes d'équilibre de charge.

Nous exprimerons à l'aide de ce modèle les propriétés essentielles permettant de caractériser une application parallèle: convergence, temps d'exécution, coût d'une communication, nombre de phases d'équilibre de charge déclenchées et nombre de messages échangés...

À l'aide de ce modèle matriciel, nous pouvons évaluer les performances de chaque stratégie suivant le caractère irrégulier de l'application qui est simulée à l'aide de lois de probabilité (normale et log-normale). Ainsi, il est possible d'étudier chaque stratégie dans plusieurs cas: application d'irrégularité faible, application d'irrégularité moyenne et application très irrégulière .

3.1 Présentation de la méthode matricielle

Nous supposons que l'application étudiée est une application FTII, donc elle peut être divisée en un certain nombre de tâches indépendantes et élémentaires. Le nombre de tâches est donc connu au début de l'exécution, par contre, le temps d'exécution de chaque tâche est imprévisible. Ce modèle est suffisamment général pour permettre une large représentation des algorithmes d'équilibre de charge. En particulier, nous appliquons ce modèle aux stratégies client-serveur, source et serveur initiative, hybride et semi-distribuée présentées dans ce mémoire.

Un processeur peut envoyer un message en direction d'un processeur quelconque de façon asynchrone. Dans une application MIMD, les processeurs peuvent être soit en train de communiquer, soit en phase de calcul. De plus, il est possible que plusieurs paires de processeurs échangent des messages simultanément. Le coût d'une communication entre deux processeurs est fixe, il est nécessaire d'évaluer cette valeur en fonction de la machine cible, car elle intervient dans l'évaluation de performance de chaque algorithme d'équilibre de charge. En effet, le coût d'une stratégie dynamique est largement déterminé par les communications induites pour équilibrer la charge.

Pour permettre la modélisation du comportement d'une stratégie d'équilibre de charge, nous introduisons les notations suivantes:

- le système est composé de p processeurs numérotés de 0 à $p-1$;
- le temps de communication d'un message de longueur 1 mot est égale à t_{com} ;

- n est le nombre de tâches qui composent l'application ;
- chaque tâche est caractérisée par son temps de calcul $t_{cal}(i)$;
- $w_i^{(k)}$ représente la charge du processeur i à l'étape k (à l'instant $t^{(k)}$);
- la charge du système à l'instant $t^{(k)}$ est représentée par le vecteur $w^{(k)} = \begin{pmatrix} w_0^{(k)} \\ \vdots \\ w_i^{(k)} \\ \vdots \\ w_{p-1}^{(k)} \end{pmatrix}$ et la charge globale du système est $W^{(k)} = \sum_{i=0}^{p-1} w_i^{(k)}$;
- le vecteur $A^{(k)} = \begin{pmatrix} a_0^{(k)} \\ \vdots \\ a_i^{(k)} \\ \vdots \\ a_{p-1}^{(k)} \end{pmatrix}$ modélise le comportement irrégulier de l'application. Ainsi, si le processeur P_i a consommé une tâche entre les étapes k et $k+1$, la valeur $a_i^{(k)}$ est égale à 1 ;
- À chaque instant, la matrice $M^{(k)} = \begin{pmatrix} \alpha_{00} & \alpha_{01} & \cdots \\ \alpha_{10} & \alpha_{11} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$ de dimensions $p \times p$ modélise le comportement de la stratégie d'équilibre de charge dynamique ;
- α_{ij} représente l'échange de charge entre les processeurs i et j.

Le but de la méthode matricielle proposée est de déterminer à chaque instant le vecteur de charge $(w^{(k)})$ qui représente la répartition de la charge dans le système. Le vecteur de charge à l'étape $k+1$ est déterminé à l'aide de $w^{(k)}$:

$$w^{(k+1)} = M^{(k)} w^{(k)} - A^{(k)}. \tag{3.1}$$

Le passage de l'étape k à l'étape $k+1$ peut intervenir dans deux cas :

1. une tâche est consommée par un processeur. Le vecteur de charge est alors modifié.
2. Une phase d'équilibre de charge débute. Il est alors nécessaire de déterminer la matrice $M^{(k)}$ en fonction de la stratégie d'équilibre de charge appliquée.

Supposons qu'à l'étape k ($t^{(k)} = 5$), le processeur P_i débute le calcul d'une tâche dont le temps d'exécution est de 3 secondes. Si aucun autre processeur n'achève le calcul d'une tâche avant 3 secondes, l'étape $k+1$ débute à l'instant $t^{(k+1)} = 8$ et le vecteur de charge $w^{(k+1)}$ est identique au vecteur $w^{(k)}$ sauf pour la ligne i :

$$w_j^{(k+1)} = w_j^{(k)}, \ \forall j \neq i \text{ et } w_i^{(k+1)} = w_i^{(k)} - 1.$$

Le vecteur $A^{(k)}$ est défini par :

$$a_j^{(k)} = \begin{cases} 0 & \text{si } j \neq i \\ 1 & \text{sinon.} \end{cases}$$

Donc, le comportement irrégulier de l'application est représenté par l'équation :

$$w^{(k+1)} = w^{(k)} - A^{(k)}.$$

S'il n'y a aucune redistribution des tâches due à la stratégie d'équilibre de charge, on peut en déduire que la matrice $M^{(k)}$ est la matrice identité et le vecteur de charge $w^{(k+1)}$ est bien déterminé par l'équation 3.1.

Si une phase d'équilibre de charge est déclenchée à l'étape k, une nouvelle redistribution des tâches est calculée. Il est nécessaire de déterminer la matrice $M^{(k)}$ en fonction de la stratégie d'équilibre de charge appliquée. En supposant qu'aucune tâche n'est consommée pendant cette phase d'équilibre de charge, le vecteur de charge à l'instant $k+1$ est égal à :

$$w^{(k+1)} = M^{(k)} w^{(k)}.$$

Dans ce cas, le vecteur $A^{(k)}$ est le vecteur nul et le comportement de l'application distribuée est donc modélisé dans les deux cas par l'équation 3.1.

Si pendant une phase d'équilibre de charge un processeur consomme une tâche, le vecteur qui représente le comportement irrégulier de l'application est différent du vecteur nul et il faut donc considérer à la fois les modifications apportées par $M^{(k)}$ et par $A^{(k)}$.

Pour déterminer complètement le comportement de l'application, il faut :

- connaître la répartition initiale de toutes les tâches et leur temps d'exécution (pour avoir la possibilité de construire $A^{(k)}$) ;
- être capable d'estimer le coût d'une phase d'équilibre de charge.

3.2 Indices de performance

Nous définissons plusieurs indices de performance pour analyser le comportement de chaque stratégie. Ainsi, nous définissons les valeurs suivantes :

t_{seq} : le temps d'exécution séquentiel de l'application, il est défini par la somme des temps de calcul associés à chaque tâche.

$$t_{seq} = \sum_{i=0}^{n} t_{cal}(i).$$

t_p : le temps d'exécution parallèle sur p processeurs. Cette mesure correspond à l'instant où le dernier processeur termine.

$$t_p = \max_{i=0}^{p-1} \left(\sum_{j=0}^{n_i} t_{cal}(j) + t_{lb}^i \right)$$

où n_i est le nombre de tâches calculées par le processeur P_i et t_{lb}^i est le temps que P_i a passé à appliquer la stratégie d'équilibre de charge.

n_{lb} : est le nombre de phases d'équilibre de charge réalisées. n_{lb} est le nombre d'étapes où la matrice $M^{(k)}$ est différente de la matrice identité.

n_{req} : le nombre de requêtes émises par les processeurs en appliquant la stratégie d'équilibre de charge.

$$n_{req} = \sum_{i=0}^{p-1} n_{req}^{i}$$

où n_{req}^{i} est le nombre de requêtes émises par P_i.

n_{tac} : le nombre de tâches transmises entre les processeurs pour équilibrer la charge du système distribué.

$$n_{tac} = \sum_{i=0}^{p-1} n_{tac}^{i}$$

où n_{tac}^{i} est le nombre de tâches déplacées par P_i.

acc : est l'accélération obtenue par la parallélisation, $acc = \frac{t_{seq}}{t_p}$.

eff : est l'efficacité de l'application parallèle, $eff = \frac{t_{seq}}{t_p \times p}$.

Chacun des paramètres définis est à évaluer en fonction de l'irrégularité de l'application, mais aussi en fonction de la stratégie d'équilibre de charge mise en œuvre.

3.3 Qualité de la répartition de charge

Afin d'estimer la qualité de la répartition de charge à une étape quelconque, nous définissons les notions de *vecteur de distribution uniforme* et d'*indicateur de qualité.*

Définition 3.5 *(vecteur de distribution uniforme)*

Le vecteur de distribution uniforme est égal à $\bar{w}^{(k)} = \begin{pmatrix} \bar{w}_0^{(k)} \\ \vdots \\ \bar{w}_{p-1}^{(k)} \end{pmatrix}$ *où*

$$\bar{w}_i^{(k)} = \begin{cases} \sum_j w_j^{(k)}/p + 1 & si\ i < \sum_j w_j^{(k)} \% p \\ \sum_j w_j^{(k)}/p & sinon. \end{cases}$$

Où / est la division entière et % le reste de la division entière.

L'indicateur de qualité mesure la distance entre la situation courante et la situation idéale représentée par le vecteur de distribution uniforme.

Définition 3.6 *(indicateur de qualité)*
La qualité de la répartition de la charge à l'étape k peut être estimée par :

$$q(k) = ||\bar{w}^{(k)} - w^{(k)}|| = \sqrt{\sum_{k=0}^{p-1} (\bar{w}_k^{(k)} - w_k^{(k)})^2}.$$

Plus la valeur de $q(k)$ est petite, meilleure est la répartition de la charge car la distance qui sépare la répartition des tâches à l'étape k de la répartition idéale, est faible.

Exemple 3.4 :

Supposons qu'à l'instant $t^{(k)}$, la charge d'un système parallèle composé de 4 processeurs est exprimée par le vecteur de charge $w^{(k)} = \begin{pmatrix} 5 \\ 0 \\ 2 \\ 8 \end{pmatrix}$. Nous pouvons calculer la distribution uniforme correspondant à cette situation : $\bar{w}^{(k)} = \begin{pmatrix} 4 \\ 4 \\ 4 \\ 3 \end{pmatrix}$.

Dans ces conditions, la qualité de la répartition de charge à l'étape k est $q(k) = \sqrt{46}$.
Si une phase d'équilibre de charge est exécutée et que le comportement de la stratégie d'équilibre de charge dynamique est modélisé par la matrice $M^{(k)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, nous en déduisons la répartition des tâches à l'instant $t^{(k+1)}$, $w^{(k+1)} = \begin{pmatrix} 5 \\ 1 \\ 1 \\ 8 \end{pmatrix}$ et $q(k+1) = \sqrt{44}$.

Nous pouvons noter que dans cette situation la stratégie d'équilibre de charge a effectivement amélioré la répartition des tâches sur l'ensemble des processeurs.

Remarque : $q(k)$ ne garantit pas que la répartition du travail restant à accomplir est parfaite. En effet, comme le temps de chaque tâche est imprévisible, même si tous les processeurs ont le même nombre de tâches à accomplir, il n'est pas possible de s'assurer que tous les processeurs termineront au même instant les calculs de leurs tâches.

De plus, si nous comparons les deux situations suivantes $w^{(k)} = \begin{pmatrix} 2 \\ 4 \\ 5 \\ 0 \end{pmatrix}$ et $w^{(l)} = \begin{pmatrix} 1 \\ 4 \\ 5 \\ 1 \end{pmatrix}$ $(k \neq l)$, intuitivement nous considérons que la situation l est préférable à la situation k. Cependant, d'après la définition du vecteur de distribution uniforme et de l'indicateur de qualité, nous constatons que $q(k) = q(l)$. En effet, d'après la définition du vecteur de distribution uniforme, $\bar{w}^{(k)} = \bar{w}^{(l)} = \begin{pmatrix} 3 \\ 3 \\ 3 \\ 2 \end{pmatrix}$ et nous vérifions que $q(k) = q(l)$.

Si nous considérons la permutation du vecteur $\bar{w}^{(k)}$, $\bar{w}'^{(k)} = \bar{w}'^{(l)} = \begin{pmatrix} 2 \\ 3 \\ 3 \\ 3 \end{pmatrix}$, l'indicateur

de qualité $q(k)$ est supérieur à $q(l)$. L'indicateur de qualité défini ne tient donc pas compte des différentes permutations du vecteur de distribution uniforme. C'est pourquoi, dans les cas où une tâche est déplacée d'un processeur dont la charge est 2 vers un processeur oisif, l'indicateur de qualité ne met pas toujours en évidence que la répartition de charge a été améliorée et il considère que les deux situations sont équivalentes.

4 Équilibre de charge dynamique pour une application FTII

Dans cette partie, nous présentons un ensemble de stratégies d'équilibre de charge dynamique pour les applications FTII. Ainsi, nous développons un algorithme centralisé de type client-serveur adapté à nos hypothèses. Trois nouvelles stratégies distribuées sont proposées : une stratégie serveur initiative, une stratégie source initiative et une stratégie hybride qui tire parti des avantages des deux stratégies distribuées précédentes. Enfin, un algorithme semi-distribué qui réalise un partitionnement logique du système parallèle est développé. Pour chaque algorithme, une analyse de la complexité théorique est détaillée.

De plus, nous modélisons le comportement de chaque stratégie d'équilibre de charge à l'aide du modèle matriciel que nous avons développé pour les applications FTII. Grâce à ce modèle, nous pouvons exprimer la qualité de la répartition de charge obtenue ainsi que le coût associé à chaque algorithme.

Pour conclure ce chapitre, nous montrons que les méthodes hybride, semi-distribuée et client-serveur sont très proches mais que la stratégie source initiative est mal adaptée aux algorithmes FTII. Enfin, nous remarquons que malgré la réduction du nombre de messages de contrôle induits par la stratégie serveur initiative, la stratégie hybride n'améliore pas de façon significative les performances obtenues.

4.1 Caractère original des algorithmes proposés

Les cinq algorithmes d'équilibre de charge dynamique développés pour les applications FTII ont plusieurs caractéristiques communes, en particulier :

- ils prennent en compte les principales propriétés des applications FTII : nombre de tâches fini et temps d'exécution de chaque tâche imprévisible.
- leur objectif est de limiter le nombre de messages de contrôle tout en maintenant une répartition des tâches satisfaisante (au sens de la définition 3.4).
- une analyse menée à l'aide du modèle matriciel apporte une preuve de convergence pour chaque algorithme.

La principale propriété des algorithmes serveur et source initiative est la méthode employée pour réduire le nombres de messages de contrôle. Chaque processeur gère une table dans laquelle il marque les processeurs qui sont dans le même état que lui (oisif ou actif). Cette optimisation permet d'éviter un nombre de messages important de type processeur inactif vers processeur inactif dans le cas de l'algorithme serveur initiative et de type processeur surchargé vers processeur surchargé dans la cas source initiative. L'algorithme hybride possède une originalité supplémentaire : il adapte sa stratégie en

fonction de l'état du système, c'est pourquoi il peut adopter un comportement de type source initiative ou serveur initiative.

La stratégie semi-distribuée hérite des propriétés des algorithmes distribuées que nous avons développés (méthode d'appariement, définition d'un marquage, choix du nombre de tâches à déplacer,...) et met en œuvre une optimisation supplémentaire. En proposant une partition du système parallèle et une gestion adaptée des phases d'équilibre de charge, cette stratégie permet une réduction sensible du nombre de messages induits tout en conservant le caractère extensible des stratégies distribuées.

4.2 Un algorithme centralisé

Afin de pouvoir réaliser une étude comparative complète des différentes techniques d'équilibre de charge dynamiques, nous avons proposé une stratégie centralisée très simple qui répartit la charge au niveau global. Notre algorithme est très proche de celui proposé par J. W. Kho [Kho 92]. Toutes les tâches sont centralisées dans une file d'attente, accessible uniquement au serveur de l'application. À chaque fois qu'un client est oisif, il demande au serveur de lui fournir une tâche. Quand le serveur reçoit une demande, il retire une tâche de sa file et la transmet au client qui a émis la requête. La détection de la terminaison de l'application est réalisée par le serveur. Quand sa file d'attente est vide, il attend que tous les processeurs aient calculé leur dernière tâche et il peut alors mettre fin à l'application.

Les performances obtenues par cet algorithme sont fonction de la granularité de l'application et du nombre de processeurs disponibles. Il est nécessaire de trouver un compromis entre la taille d'une tâche et le nombre de messages induits par leur distribution. En effet, il faut éviter que le serveur ne devienne un goulot d'étranglement (voir le paragraphe 3.2.1 du chapitre 2).

4.2.1 Analyse de la stratégie client-serveur

Afin d'étudier la convergence de la stratégie client-serveur, ainsi que son influence sur le comportement global de l'application FTII, nous modélisons cette stratégie à l'aide du modèle matriciel que nous avons proposé.

4.2.1.1 Construction de la matrice $M^{(k)}$ à l'instant $t^{(k)}$. La matrice $M^{(k)}$ détermine complètement le comportement de la stratégie étudiée. En fonction de la situation courante (représentée par le vecteur $w^{(k)}$), nous pouvons exprimer $M^{(k)}$ pour la stratégie client-serveur. Afin d'obtenir l'expression de $M^{(k)}$, nous supposons que le serveur est le processeur P_0. De plus, nous distinguons deux cas :

- cas 1, tous les processeurs sont chargés ;
- cas 2, un processeur est inactif.

Cas 1 : quand tous les processeurs sont actifs, la stratégie client-serveur ne modifie pas la répartition de la charge, donc la matrice $M^{(k)}$ est la matrice identité car chaque processeur conserve sa charge locale. Nous pouvons exprimer ce premier cas de la façon suivante :
Si $\forall i \neq 0,\ w_i^{(k)} = 1$, alors $M^{(k)} = Id$.
Nous pouvons noter que la charge du processeur P_i $(i \neq 0)$ ne peut prendre que deux

valeurs, soit $w_i^{(k)} = 0$ (le processeur est déchargé) ou bien $w_i^{(k)} = 1$ (P_i est actif).

Cas 2 : si un processeur est oisif, il demande une nouvelle tâche au serveur. Nous exprimons cet état ainsi :

$$\exists i \neq 0, w_i^{(k)} = 0 \text{ et } w_0^{(k)} > 0.$$

Comme P_i reçoit une tâche du serveur (P_0), nous pouvons en déduire :

$$\alpha_{i0} = \alpha_{0i} = \frac{1}{w_o^{(k)}}.$$

P_i et P_0 ne modifient pas le comportement des autres processeurs, donc toutes les autres entrées des lignes 0 et i sont nulles (à l'exception de α_{ii} et α_{00}) :

$$\forall l \neq i \text{ et } l \neq 0, \alpha_{il} = \alpha_{0l} = 0.$$

Aucun échange n'est autorisé par la stratégie client-serveur entre les autres processeurs clients donc toutes les autres entrées de la matrice sont nulles (à l'exception des éléments de la diagonale) :

$$\forall l \neq i \text{ et } l \neq j, \forall m \neq l, \alpha_{lm} = 0.$$

La charge globale du système est conservée, donc :

$$\forall l, \alpha_{ll} = \sum_{m \neq l} \alpha_{lm}.$$

Exemple 3.5 :

Supposons qu'un système distribué composé de 4 processeurs applique la stratégie client-serveur, et que la charge soit définie par le vecteur $w^{(k)} = \begin{pmatrix} 10 \\ 1 \\ 0 \\ 1 \end{pmatrix}$. Nous déduisons la charge du système à l'étape suivante à l'aide de l'expression de $M^{(k)}$:

$$M^{(k)} = \begin{pmatrix} 1 - \frac{1}{w_0^{(k)}} & 0 & \frac{1}{w_0^{(k)}} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{w_0^{(k)}} & 0 & 1 - \frac{1}{w_0^{(k)}} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{9}{10} & 0 & \frac{1}{10} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{10} & 0 & \frac{9}{10} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ et } w^{(k+1)} = M^{(k)} w^{(k)} = \begin{pmatrix} 9 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Une tâche a été transmise par le serveur (P_0) en direction de P_2 qui était oisif.

4.2.1.2 Qualité de la répartition de charge obtenue.

Afin de montrer la convergence de la stratégie client-serveur, nous étudions comment est modifiée la répartition de la charge par la stratégie d'équilibre de charge.

Proposition 3.2 *La stratégie client-serveur améliore la répartition des tâches.*

Preuve : il suffit de montrer que $q(k+1) - q(k) < 0$ quand la stratégie client-serveur déplace une tâche. En effet, si cette condition est vérifiée, la répartition de la charge à l'étape $k+1$ est plus proche de la distribution uniforme qu'à l'étape k.

Il est évident que lorsque la stratégie ne modifie pas la répartition de la charge ($M^{(k)} = Id$), $q(k+1) = q(k)$.

Si le serveur n'a plus qu'une tâche ($w_0^{(k)} = 1$), il transmet cette tâche au premier processeur qui en fait la demande. Dans ce cas, la répartition des tâches n'est pas améliorée, mais le serveur ne participe pas à la résolution du problème et il n'a donc pas d'autre choix.

Supposons qu'une tâche soit transmise par P_0 (le serveur) à P_i et $w_0^{(k)} > 1$. Afin de vérifier que $q(k+1) - q(k) < 0$, nous étudions le signe de $q(k+1) - q(k)$. Or, par définition, $q(k+1) > 0$ et $q(k) > 0$, donc le signe de $q(k+1) - q(k)$ est déterminé par le signe de $q^2(k+1) - q^2(k)$.

En substituant $q(k)$ et $q(k+1)$ par leur définition :

$$q^2(k+1) - q^2(k) = \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k+1)} - w_l^{(k+1)}\right)^2 - \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2$$

il n'y pas de consommation de tâche entre les étapes k et $k+1$, donc, $\bar{w}^{(k+1)} = \bar{w}^{(k)}$ et :

$$q^2(k+1) - q^2(k) = \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k+1)}\right)^2 - \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2$$

or, seules les charges de P_0 et de P_i sont modifiées, donc :

$$q^2(k+1) - q^2(k) = \sum_{l\neq 0, l\neq i} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2 + \left(1 - \bar{w}_i^{(k)}\right)^2 + \left(w_0^{(k)} - 1 - \bar{w}_o^{(k)}\right)^2 - \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2$$

$$q^2(k+1) - q^2(k) = \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2 - \left(\bar{w}_i^{(k)} - w_i^{(k)}\right)^2 - \left(\bar{w}_0^{(k)} - w_0^{(k)}\right)^2 + \left(1 - \bar{w}_i^{(k)}\right)^2 + \left(w_0^{(k)} - 1 - \bar{w}_o^{(k)}\right)^2 - \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2$$

après simplification, nous obtenons l'équation suivante :

$$q^2(k+1) - q^2(k) = 1 + \bar{w}_0^{(k)} - \bar{w}_i^{(k)} - w_0^{(k)}$$

Donc, pour déterminer le signe de $q(k+1) - q(k)$, il faut étudier deux cas :

1. La i^{e} composante du vecteur de distribution uniforme ($\bar{w}^{(k)}$) est égale à la 0^{e} composante : $\bar{w}_i^{(k)} = \bar{w}_0^{(k)}$. Nous pouvons en déduire que $q(k+1) - q(k) < 0$ car $w_0^{(k)} > 1$.

2. La i^{e} composante du vecteur de distribution uniforme ($\bar{w}^{(k)}$) est égale à la 0^{e} composante -1 : $\bar{w}_i^{(k)} = \bar{w}_0^{(k)} - 1$. Il faut dans ce cas particulier, considérer deux situations : $w_0^{(k)} > 2$ et $w_0^{(k)} = 2$. Si la charge du serveur est strictement supérieure à 2 ($w_0^{(k)} > 2$), $q(k+1) - q(k) < 0$. Par contre, si $w_0^{(k)} = 2$, nous rencontrons la situation décrite dans la remarque du paragraphe 3.3 où l'indicateur de qualité considère les deux situations comme identiques. Cependant, la situation a été améliorée car le processeur P_i est à nouveau actif.

Nous pouvons noter que le cas où $\bar{w}_i^{(k)} = \bar{w}_0^{(k)} + 1$ n'est pas possible d'après la définition de $\bar{w}^{(k)}$ car $i > 0$ (voir définition 3.6).

Nous avons bien montré que dans tous les cas où il est possible de rééquilibrer la distribution des tâches, $q(k+1)$ est inférieur à $q(k)$. Nous pouvons en conclure que la stratégie client-serveur améliore la répartition des tâches. □

4.2.1.3 Coût d'une phase d'équilibre de charge. Quand la charge est équilibrée par la stratégie client-serveur, nous devons estimer le coût de ce rééquilibrage. L'étude du nombre de messages de contrôle permet d'avoir une estimation satisfaisante du coût de cette phase. Supposons que le serveur ait transmis une tâche à P_i ($i \neq 0$), dans ce cas une requête a été émise par P_i en direction de P_0 et une tâche a été déplacée. Donc, l'étape suivante débute à l'instant :

$$t^{(k+1)} = t^{(k)} + t_{com} + t_{com} l.$$

Où l est la longueur du message qui décrit une tâche. Rappelons que pour les applications FTII, l'identifiant de la tâche suffit et donc $l = 1$. Si, en plus de l'identifiant, il est nécessaire de transférer des données (voir paragraphe 2.2.3), la valeur de l doit être déterminée pour prendre en compte ce surcoût en communication.

Seuls le serveur et P_i sont pénalisés par cette phase d'équilibre de charge, les autres clients continuent à travailler normalement.

Remarque : quand le processeur P_i demande du travail au serveur et qu'il n'y a plus de tâche en attente, P_i détecte la terminaison. Même si aucune tâche n'est déplacée, il faut considérer cette étape car les processeurs P_0 et P_i sont pénalisés par l'émission de la requête infructueuse et $t^{(k+1)} = t^{(k)} + t_{com}$. Ce cas se présente exactement $p - 1$ fois, car c'est le nombre d'échecs nécessaires pour que tous les clients détectent la terminaison de l'application.

4.3 Les algorithmes distribués

Nous avons vu qu'il est possible de proposer des algorithmes distribués simples qui permettent d'améliorer de façon sensible le temps de réponse du système. Pour qu'un algorithme distribué soit efficace, il doit réaliser un compromis entre le nombre de messages de contrôle engendrés et la qualité de l'équilibre de charge obtenu. Nous proposons trois stratégies distribuées qui modifient la charge au niveau global [KHHG 96a]. Ces trois méthodes sont appliquées de manière asynchrone par tous les processeurs. Au départ, l'ensemble des tâches est réparti équitablement entre tous les processeurs. Nous supposons qu'il n'y a pas de notion de voisinage. Ainsi, un processeur peut communiquer avec tout processeur qui constitue le système parallèle.

Pour les trois stratégies distribuées, chaque processeur possède une variable locale *cible* qui désigne, pour la prochaine phase d'équilibre de charge, le premier processeur interrogé. Il existe plusieurs méthodes pour initialiser cette variable parmi lesquelles nous pouvons distinguer les trois solutions suivantes :

1. Chaque processeur choisit au hasard un processeur du système. Cette méthode présente deux inconvénients principaux : rien ne garantit que le même processeur ne recevra pas plusieurs demandes simultanément et de plus, certains processeurs risquent de n'être jamais interrogés.

2. Chaque processeur initialise la variable *cible* avec une valeur fixe et identique pour tous (par exemple $cible = 0$). Cette valeur est ensuite incrémentée à chaque nouvelle requête. Dans ce cas, tous les processeurs du système participeront à l'équilibre de charge. Cependant, il est possible qu'un même processeur soit surchargé par le nombre de requêtes reçues simultanément.

3. Chaque processeur initialise la variable *cible* avec une valeur différente des autres processeurs. Une solution simple est d'affecter à la variable *cible* la valeur $(i+1)\%p$ pour le processeur P_i ($0 \leq i \leq p-1$ et % est le modulo). Ensuite la variable *cible* est incrémentée à chaque requête. Ainsi on limite le risque qu'un processeur soit inondé de requêtes et tous les processeurs participent à la résolution du problème de l'équilibre de charge. C'est cette méthode que nous avons choisi d'appliquer pour les trois stratégies présentées.

4.3.1 Un algorithme serveur initiative

Dans cet algorithme, le rééquilibrage de la charge est à l'initiative des processeurs déchargés qui recherchent du travail auprès des processeurs surchargés (voir algorithme 3.1).

4.3.1.1 Principe de l'algorithme. Nous présentons les idées principales ainsi que les hypothèses de notre algorithme :

- comment différencier les processeurs surchargés des processeurs déchargés ? Dans [ELZ 86c], un processeur dont la charge est inférieure à T tâches est défini comme un processeur déchargé. À l'opposé, un processeur est considéré comme surchargé s'il a plus de T tâches à traiter. Nous choisissons pour notre stratégie la valeur $T = 1$ proposée par Livny et Melman [LM 82]. Nous montrerons que modifier ce seuil ne permet pas d'améliorer les performances de cette stratégie car un processeur ne peut aider efficacement un nœud distant tant qu'il n'est pas totalement déchargé.

- Comment déterminer l'état courant du système distribué ? Chaque processeur conserve dans sa mémoire un tableau d'indicateurs de charge sur l'ensemble du système. Ces indicateurs sont mis à jour à chaque tentative de transfert d'une tâche (mise à jour à la demande, voir paragraphe 3.6.1.2 du chapitre 2). À chaque fois qu'un processeur P_j refuse de partager sa charge avec un processeur déchargé P_i, P_i déduit que P_j est aussi déchargé et le mémorise dans sa table. De même, le processeur P_j mémorise dans sa propre table que P_i est déchargé. Cette proposition permet d'éviter des communications inutiles entre deux processeurs déchargés. En effet, lors d'une prochaine phase d'équilibre de charge, P_i et P_j éviteront dans un premier temps de s'interroger car chacun connaît l'état de l'autre.

- Quelle est la méthode d'appariement ? La méthode d'appariement détermine avec quel nœud un processeur donné doit partager sa charge. Le processeur déchargé P_i demande des tâches à P_{i+1}. Si la requête échoue, *i. e.* P_{i+1} est aussi déchargé, alors P_i demande du travail au prochain processeur non marqué dans sa table. Ce principe est appliqué jusqu'à ce que P_i trouve du travail ou que tous les processeurs aient été interrogés. Dans ce dernier cas, l'équilibre de charge est différé à une étape ultérieure.

- Quelle est la méthode de transfert? La méthode de transfert détermine le nombre de tâches à déplacer. Quand un processeur P_i a trouvé un processeur P_j surchargé, alors P_j transmet S tâches à P_i. S est une valeur à déterminer. Dans [ELZ 86c], S est choisi égal à 1. Il est à noter que si la valeur de S est supérieure à 1, un processeur déchargé peut devenir surchargé car il peut avoir reçu plusieurs tâches depuis la requête précédente. Ainsi, si la stratégie d'équilibre de charge échoue avec les processeurs non marqués, il est nécessaire d'interroger les processeurs marqués. Nous expérimentons la valeur de $S = 1$, mais nous proposons aussi une gestion dynamique de S: quand P_i a trouvé un processeur P_j surchargé, P_j transmet la moitié de sa charge au processeur P_i. Dans ce cas, un équilibre de charge local parfait est obtenu entre P_i et P_j. Cependant, le fait qu'ils possèdent le même nombre de tâches ne garantit pas qu'ils seront actifs pendant le même intervalle de temps (voir la remarque du paragraphe 3.3).

Remarque : Tous les processeurs travaillent dans un système coopératif. Ainsi, avant de réaliser un transfert de tâches, la phase de négociation suivante est appliquée en trois étapes :

1. Le processeur déchargé P_i envoie $V(i)$, la valeur de sa charge ($V(i) = 0$), au processeur P_j : il effectue une requête de transfert de tâches ;

2. Le processeur P_j met à jour $V(i)$ dans sa table des indicateurs et envoie à P_i la valeur de sa charge $V(j)$.

3. P_i reporte la charge de P_j dans la table d'indicateurs qu'il gère. Si $V(j)$ est supérieure à T, l'appariement entre P_i et P_j est décidé, P_j envoie alors S tâches à P_i. Dans le cas contraire, la requête échoue et P_i réalise une nouvelle phase de négociation avec un autre processeur.

Algorithme 3.1 *Algorithme serveur initiative*

```
fonct Serveur (i, n, T) : Boolean
                                  /* i : numéro du processeur n : nombre de processeurs, T : seuil */
  Cible ⟵ (i + 1) % n
  Succès ⟵ faux
  Fin ⟵ faux
  tant que (¬Succès) ∧ (¬Fin) faire
          si ¬Marqué (Cible) alors
            si DemandeCharge (Cible) > T alors
              Succès ⟵ vrai                    /* Le processeur Cible est surchargé */
            sinon
               Marque (Cible)
               Cible ⟵ (Cible + 1) % n
               si Cible = i alors Fin ⟵ vrai finsi
            finsi
          finsi
  fintant que
```

si Succès **alors**
  RecevoirCharge (Cible) /* Le processeur Cible peut partager sa charge */
  retourner vrai /* avec le processeur i */
sinon retourner faux /* toutes les tâches ont été calculées */
finsi.

Un exemple d'exécution de l'algorithme serveur initiative est présenté figure 3.2.

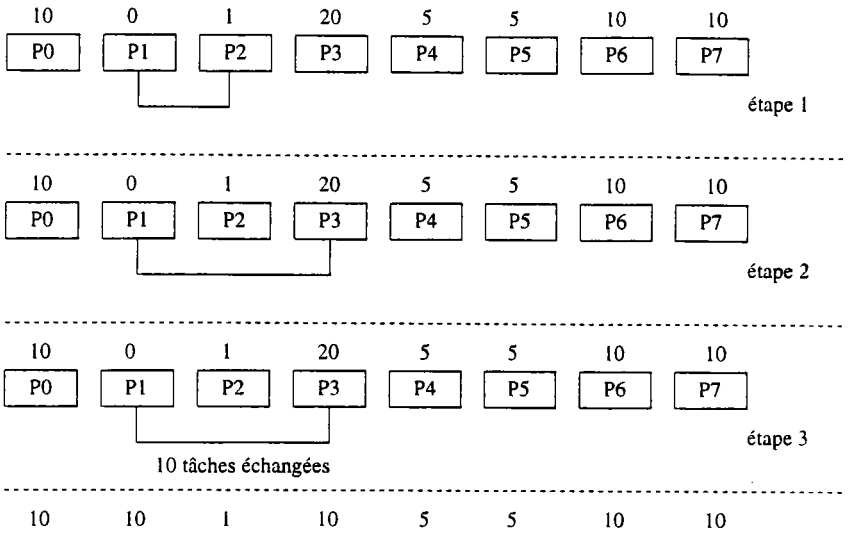


FIG. 3.2 –*exemple d'exécution de l'algorithme serveur initiative, 8 processeurs*

4.3.1.2 Propriétés de l'algorithme serveur initiative.

Proposition 3.3 *(Terminaison)*
L'algorithme serveur initiative termine si et seulement si au moins un processeur a émis $p-1$ *requêtes sans succès, où p est le nombre de processeurs du système.*

Preuve : un processeur déchargé P_i a émis $p-1$ requêtes si et seulement si, il a interrogé, sans succès, les $p-1$ processeurs distincts qui constituent le système. Ce cas n'est possible que si l'ensemble des processeurs est déchargé. En d'autres termes, tous les processeurs ont une charge inférieure ou égale à T : ils ne peuvent donc plus la partager. □

Proposition 3.4 *(Complexité en communication)*
Le nombre total de communications de l'algorithme serveur initiative est en $O(n \times p)$ *où p est le nombre de processeurs et n le nombre de tâches à calculer.*

Preuve : dans le pire des cas, toute la charge du système est concentrée sur un seul processeur et le seuil S est égal à 1. Ainsi, à chaque nouvelle étape de redistribution des tâches, un processeur déchargé interroge au plus $p-1$ processeurs. Donc la distribution des n tâches nécessite au plus $n \times (p-1)$ communications. Pour détecter la terminaison de l'algorithme, $p-1$ communications sont nécessaires (voir la proposition précédente). Ainsi, le nombre total de communications est $(n+1) \times (p-1)$. □

Proposition 3.5 *(Comparaison des stratégies serveur initiative et client-serveur)*
La complexité en temps du cas le plus défavorable de l'algorithme serveur initiative est égale à la complexité en temps de l'algorithme centralisé dans une architecture client-serveur.

Preuve :

- soit t_1, la complexité en temps du cas le plus défavorable de l'algorithme client-serveur ;
- soit t_2, la complexité en temps du cas le plus défavorable de l'algorithme serveur initiative ;
- soit t_{seq} le temps d'exécution séquentiel de l'application $\mathcal{A}$ de type FTII ;
- soit n le nombre de tâches de $\mathcal{A}$;
- soit t_c le temps de communication d'une tâche ;
- soit t_n le temps d'une phase de négociation ;
- soit t_{max} le temps de calcul de la tâche la plus coûteuse ;
- soit p le nombre de processeurs.

Le pire des cas de l'algorithme client-serveur se produit quand tous les processeurs terminent leur travail à l'exception d'un processeur qui calcule la tâche la plus coûteuse. Ainsi, on peut estimer t_1 de la façon suivante :

$$t_1 \simeq \frac{t_{seq} - t_{max}}{p-1} + t_{max} + n \times t_c$$

Le pire des cas de l'algorithme serveur initiative se produit aussi quand tous les processeurs terminent leur travail à l'exception d'un processeur qui calcule la tâche la plus coûteuse. Ainsi, on peut estimer t_2 de la façon suivante :

$$t_2 \simeq \frac{t_{seq} - t_{max}}{p} + t_{max} + n \times t_c + (p-1) \times t_n$$

En effet, n tâches doivent être déplacées, ce qui coûte au plus $n \times t_c$ unités de temps. Le facteur $(p-1)$ représente les $(p-1)$ requêtes infructueuses avant de trouver l'unique processeur surchargé. Grâce à l'utilisation de la table d'indicateurs de charge, il n'y a plus aucune requête rejetée quand le processeur surchargé a été trouvé. t_n représente le temps de communication de 1 octet qui est très petit et donc t_1 et t_2 sont pratiquement égaux. □

4.3.1.3 Analyse de la stratégie serveur initiative.

Nous nous proposons de modéliser et d'analyser le comportement dynamique de la stratégie serveur initiative proposée à l'aide de la méthode matricielle pour les applications FTII. Le rééquilibrage de la charge est à l'initiative des processeurs déchargés qui recherchent du travail auprès des processeurs surchargés. Quand un processeur est oisif, il déclenche une nouvelle phase d'équilibre de charge. Afin de trouver des tâches disponibles, il interroge les processeurs distants jusqu'à trouver un processeur dont la charge est supérieure ou égale à 2. Si le processeur P_j dont la charge est $w_j \geq 2$ est interrogé par le processeur P_i, P_j envoie $\frac{w_j}{2}$ à P_i.

Afin d'étudier le comportement de cette stratégie dynamique, il est nécessaire de construire $M^{(k)}$. Ainsi, nous aurons la possibilité de prouver la convergence de la méthode serveur initiative et nous exprimerons le coût d'une phase d'équilibre de charge.

Construction de la matrice $M^{(k)}$ à l'instant $t^{(k)}$. Pour construire $M^{(k)}$ à chaque instant, il est nécessaire de distinguer deux cas :

- cas 1, tous les processeurs sont chargés ;
- cas 2, un processeur est oisif.

Cas 1 : si $\forall i,\ w_i^{(k)} \neq 0$ alors $M^{(k)} = Id$.
En effet, tous les processeurs sont actifs et il n'y a donc pas d'échange entre les processeurs. La stratégie serveur initiative ne modifie pas le comportement de l'application.

Cas 2 : $w_i^{(k)} = 0,\ \exists\ !j \mid w_j^{(k)} > 1$ et $d(i,j) = (j-i)\%p$ est minimum.
Le processeur P_i est inactif et P_j est le premier processeur interrogé par P_i susceptible de partager sa charge avec lui.
P_i reçoit la moitié des tâches du processeurs P_j. Donc :

$$\alpha_{ij} = \alpha_{ji} = \frac{1}{2}.$$

Toutes les autres entrées des lignes i et j sont nulles (à l'exception de α_{ii} et α_{jj}) :

$$\forall l \neq i \text{ et } l \neq j,\ \alpha_{il} = \alpha_{jl} = 0.$$

Il n'y a pas d'échange entre les autres processeurs, donc toutes les entrées de la matrice (à l'exception des éléments de la diagonale) sont nulles :

$$\forall l \neq i \text{ et } l \neq j,\ \forall m \neq l,\ \alpha_{lm} = 0.$$

La charge globale du système est conservée :

$$\forall l,\ \alpha_{ll} = 1 - \sum_{m \neq l} \alpha_{lm}.$$

Remarque : la somme de chaque ligne et de chaque colonne est égale à 1 et tous les éléments de la matrice sont positifs. La matrice est donc doublement stochastique.

Exemple 3.6 :

La matrice associée au vecteur de charge $w^{(k)} = \begin{pmatrix} 5 \\ 0 \\ 2 \\ 8 \end{pmatrix}$ est $M^{(k)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

Le vecteur de charge à l'instant $t^{(k+1)}$ est $w^{(k+1)} = M^{(k)} \times w^{(k)}$, pour l'exemple précédent on obtient $w^{(k+1)} = \begin{pmatrix} 5 \\ 1 \\ 1 \\ 8 \end{pmatrix}$.

Qualité de la répartition de charge obtenue.

Proposition 3.6 *La stratégie serveur initiative améliore la répartition des tâches.*

Preuve : Il suffit de montrer que $q(k)$ est décroissante. Si $M^{(k)} = Id$, alors $q(k) = q(k+1)$. Quand la matrice $M^{(k)}$ est différente de la matrice identité :

$$q(k) = \sqrt{\sum_{l=0}^{p-1} (\bar{w}_l^{(k)} - w_l^{(k)})^2} \text{ et}$$

$$q(k+1) = \sqrt{\sum_{l\neq i, l\neq j} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2 + \left(\frac{w_j^{(k)}}{2} - \bar{w}_i^{(k)}\right)^2 + \left(\frac{w_j^{(k)}}{2} - \bar{w}_j^{(k)}\right)^2}$$

car P_i reçoit la moitié des tâches de P_j.

$$= \sqrt{\sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2 - \left(\bar{w}_i^{(k)} - w_i^{(k)}\right)^2 - \left(\bar{w}_j^{(k)} - w_j^{(k)}\right)^2 + \left(\frac{w_j^{(k)}}{2} - \bar{w}_i^{(k)}\right)^2 + \left(\frac{w_j^{(k)}}{2} - \bar{w}_j^{(k)}\right)^2}$$

Afin de montrer que q(k) est décroissante, étudions le signe de $q(k+1) - q(k)$. Or, par définition $q(k)$ et $q(k+1)$ sont positifs, donc le signe de la différence $q(k+1) - q(k)$ est aussi le signe de $q^2(k+1) - q^2(k)$.

$$\begin{aligned}
q^2(k+1) - q^2(k) &= \sum_{l=0}^{p-1} \left(\bar{w}_l^{(k)} - w_l^{(k)}\right)^2 - \left(\bar{w}_i^{(k)} - w_i^{(k)}\right)^2 - \left(\bar{w}_j^{(k)} - w_j^{(k)}\right)^2 \\
&+ \left(\frac{w_j^{(k)}}{2} - \bar{w}_i^{(k)}\right)^2 + \left(\frac{w_j^{(k)}}{2} - \bar{w}_j^{(k)}\right)^2 - \sum_{l=0}^{p-1} (\bar{w}_l^{(k)} - w_l^{(k)})^2 \\
q^2(k+1) - q^2(k) &= \left(\frac{w_j^{(k)}}{2} - \bar{w}_i^{(k)} - w_i^{(k)} + \bar{w}_i^{(k)}\right)\left(\frac{w_j^{(k)}}{2} - \bar{w}_i^{(k)} + w_i^{(k)} - \bar{w}_i^{(k)}\right) \\
&+ \left(\frac{w_j^{(k)}}{2} - \bar{w}_j^{(k)} - w_j^{(k)} + \bar{w}_j^{(k)}\right)\left(\frac{w_j^{(k)}}{2} - \bar{w}_j^{(k)} + w_j^{(k)} - \bar{w}_j^{(k)}\right) \\
q^2(k+1) - q^2(k) &= \frac{w_j^{(k)}}{2}\left(\frac{w_j^{(k)}}{2} - 2\bar{w}_i^{(k)}\right) + \frac{-w_j^{(k)}}{2}\left(\frac{3}{2}w_j^{(k)} - 2\bar{w}_j^{(k)}\right) \\
&\text{car } w_i^{(k)} = 0. \\
q^2(k+1) - q^2(k) &= \frac{w_j^{(k)}}{2}\left(-w_j^{(k)} - 2\bar{w}_i^{(k)} + 2\bar{w}_j^{(k)}\right) \\
q^2(k+1) - q^2(k) &= \frac{-w_j^{(k)}}{2}\left(w_j^{(k)} + 2\left(\bar{w}_i^{(k)} - \bar{w}_j^{(k)}\right)\right)
\end{aligned}$$

Donc la différence $q^2(k+1) - q^2(k)$ est strictement négative si $w_j^{(k)} + 2\left(\bar{w}_i^{(k)} - \bar{w}_j^{(k)}\right) > 0$ car $w_j^{(k)} > 1$. D'après la définition de $\bar{w}^k$, Il faut étudier trois cas :

1. La i^{e} composante du vecteur de distribution uniforme $(\bar{w}^{(k)})$ est égale à la j^{e} composante : $\bar{w}_i^{(k)} = \bar{w}_j^{(k)}$. Nous pouvons en déduire que $q(k+1) - q(k) < 0$ car $w_j^{(k)} > 1$.

En effet la charge de P_j est forcément strictement supérieure à 1 car P_j a accepté de partager sa charge avec P_i.

2. La i^e composante du vecteur de distribution uniforme $(\bar{w}^{(k)})$ est égale à la j^e composante $+1$: $\bar{w}_i^{(k)} = \bar{w}_j^{(k)} + 1$. Donc $q(k+1) - q(k) < 0$ car $w_j^{(k)} > 1$ et $2\left(\bar{w}_i^{(k)} - \bar{w}_j^{(k)}\right) = 2$.

3. La j^e composante du vecteur de distribution uniforme $(\bar{w}^{(k)})$ est égale à la i^e composante $+1$: $\bar{w}_j^{(k)} = \bar{w}_i^{(k)} + 1$. Si $w_j^{(k)} > 2$, $q(k+1) - q(k) < 0$ car $\left(\bar{w}_i^{(k)} - \bar{w}_j^{(k)}\right) = -2$. Par contre, si $w_j^{(k)} = 2$, nous rencontrons la situation très particulière présentée dans la remarque du paragraphe 3.3. Dans ce cas, nous avons expliqué pourquoi l'indicateur de qualité considère les deux situations équivalentes bien que la situation à l'étape $q+1$ soit préférable car le processeur P_i est à nouveau actif.

□

Proposition 3.7 *La stratégie serveur initiative est plus efficace lorsque le processeur déchargé P_i s'apparie avec le processeur P_j le plus surchargé.*

Preuve : On peut exprimer cette proposition ainsi :
$q(k+1)$ est minimale si et seulement si la charge de P_j est maximale ou bien encore : $q(k+1)$ est minimale ssi $w_j^{(k)} = \max_{l=0}^{p-1}\left(w_l^{(k)}\right)$.
Or, d'après la démonstration de la proposition précédente,

$$q^2(k+1) - q^2(k) = \frac{w_j^{(k)}}{2}\left(-w_j^{(k)} - 2\bar{w}_i^{(k)} + 2\bar{w}_j^{(k)}\right).$$

Comme la différence $q^2(k+1) - q^2(k)$ est négative et que $q(k)$ est une constante, on en déduit que $q(k+1)$ est minimum si et seulement si $|q^2(k+1) - q^2(k)|$ est maximum. Or, $|q^2(k+1) - q^2(k)|$ est maximum quand $w_j^{(k)} = \max_{l=0}^{p-1}\left(w_l^{(k)}\right)$. □

Coût d'une phase d'équilibre de charge. Quand la charge est équilibrée par la stratégie dynamique $(M^{(k)} \neq Id)$, il est nécessaire d'estimer le coût de cette phase. L'étude du nombre de messages de contrôle induits permet de mesurer la durée de cette phase. En effet, si le processeur P_i s'apparie avec P_j, nous pouvons déduire que P_i a émis $(j-i)\%p$ requêtes. De plus $\frac{w_j^{(k)}}{2}$ tâches sont envoyées par P_j en direction du processeur oisif. Donc, la valeur de $t^{(k+1)}$ est déterminée par l'équation :

$$t^{(k+1)} = t^{(k)} + 2\left((j-i)\%p\right)t_{com} + \frac{w_j^{(k)}}{2} t_{com} l.$$

Où l est la longueur du message qui décrit une tâche. Comme l'ensemble des tâches est homogène, la valeur de l est fixe quelle que soit la tâche transmise.

Chaque processeur n'est pas pénalisé de la même façon par cette phase d'équilibre de charge :

- les processeurs qui n'ont pas été interrogés par P_i, ne sont pas pénalisés par cette phase d'équilibre de charge. Ils continuent à travailler normalement.

- Les processeurs interrogés par P_i qui n'ont pu répondre favorablement à sa requête sont tous pénalisés. En effet, ils ont interrompu leur travail pendant $2t_{com}$ secondes pour lire et répondre à la requête de P_i.
- Le processeur P_j est lui pénalisé plus largement. Il a non seulement répondu à la requête de P_i, mais lui a aussi transmis la moitié de ses tâches. Ainsi P_j a arrêté de travailler pendant un intervalle de temps égal à $2t_{com} + \frac{w_j^{(k)}}{2} t_{com} l$.
- Enfin le processeur P_i n'a pas travaillé pendant toute cette période. Il est donc resté inactif pendant $2((j-i)\%p)t_{com} + \frac{w_j^{(k)}}{2} t_{com} l$ secondes.

Remarque : Quand P_i initie une phase d'équilibre de charge et qu'il ne peut trouver un processeur avec qui s'appareiller ($M^{(k)} = Id$), il faut néanmoins considérer cette étape. En effet, P_i détecte la terminaison de l'algorithme et il émet $p-1$ requêtes infructueuses qui ont perturbé le comportement global du système. Cependant, ce cas ne se présente que p fois exactement, car tous les processeurs auront détecté la terminaison après le p^e cas.

Exemple 3.7 :

On dispose d'une machine parallèle composée de 4 processeurs (numérotés de 0 à 3). Une application parallèle composée de 8 tâches doit être exécutée sur cette machine. Le temps d'exécution de chaque tâche ainsi que leur répartition initiale sont donnés dans le tableau suivant :

Numéro de la tâche	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Temps d'exécution	2	1	4	8	5	2	8	7
Localisation initiale	0	0	1	1	2	2	3	3

Le temps de communication d'un message est égal à 0,25 seconde ($t_{com} = 0,25$). La stratégie appliquée est la stratégie serveur initiative proposée au chapitre précédent. Quand P_i émet une requête en direction de P_j, deux messages sont nécessaires : la demande de P_i et la réponse de P_j. C'est pourquoi, le temps d'exécution d'un requête est égal à 0,5 seconde. Pour cet exemple, on suppose que le transfert d'une tâche a un coût nul. En utilisant la méthode matricielle présentée, le comportement suivant est déduit (voir figure 3.3). Nous pouvons remarquer que ce système est pénalisé par les communications.

Il est possible d'évaluer chacun des indices de performance présentés :

t_{seq} : le temps d'exécution séquentiel, $t_{seq} = \sum_{i=0}^{7} t_{cal}(i) = 37$ secondes.

t_4 : le temps d'exécution parallèle sur 4 processeurs, $t_4 = \max_{i=0}^{3} \left(\sum_{j=0}^{nb_i} t_{cal}(j) + t_{lb}^i \right) = 16,5$ secondes car c'est le processeur P_2 qui détecte le dernier la terminaison.

n_{lb} : le nombre de phases d'équilibre de charge, $n_{lb} = 7$. En effet, P_0 initie deux phases d'équilibre de charge, de même P_1 et P_2 sont responsables de deux phases chacun et P_3 n'est responsable que d'une seule tentative.

n_{req} : le nombre de requêtes émises par les processeurs, $n_{req} = \sum_{i=0}^{3} n_{req}^i = 15$. En effet, P_0, P_1 et P_2 ont émis 4 requêtes chacun et P_3 a émis seulement 3 requêtes.

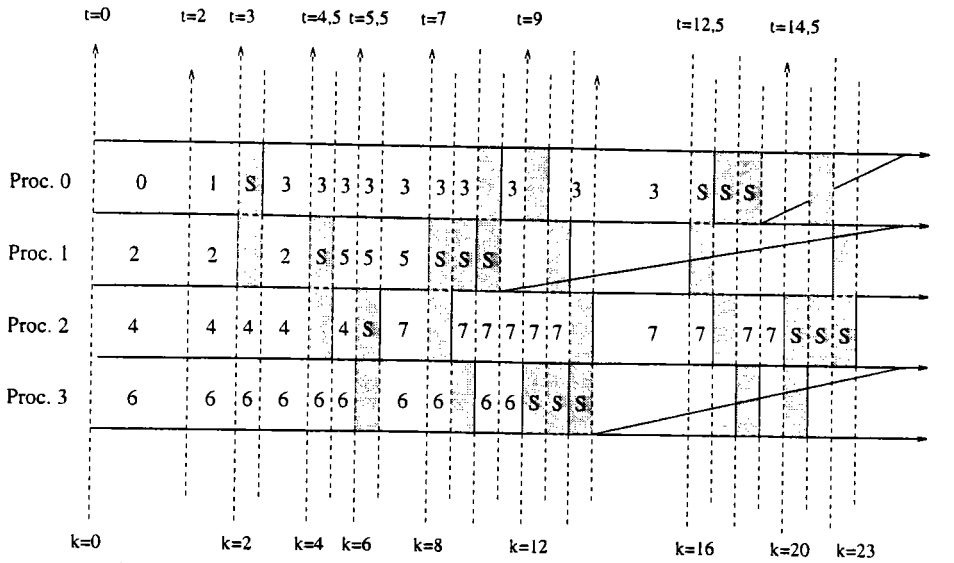


FIG. 3.3 –*exemple de comportement dynamique de la stratégie serveur initiative.*

n_{tac} : le nombre de tâches transmises, $n_{tac} = \sum_{i=0}^{3} n^i_{tac} = 3$. Car P_1 a envoyé la tâche 3 à P_0, P_2 a transmis la tâche 5 à P_1 et P_3 a confié la tâche 7 à P_2.

acc : l'accélération, $acc = \frac{t_{seq}}{t_4} = 2,24.$

eff : l'efficacité, $eff = \frac{t_{seq}}{t_4 \times 4} = 0,56.$

Nous pouvons conclure que sur cet exemple, la stratégie a sensiblement modifié le comportement de l'application car trois tâches ont été déplacées (sur un total de huit) et que la stratégie a été particulièrement pénalisée par le coût des communications.

4.3.2 Un algorithme source initiative

L'algorithme de type source initiative est une stratégie duale de la méthode serveur initiative. À l'opposé de la méthode serveur initiative, le déclenchement d'une nouvelle phase d'équilibre de charge est décidé par les processeurs surchargés (voir algorithme 3.2).

4.3.2.1 Principe de l'algorithme. La stratégie source initiative que nous avons développée est basée sur les hypothèses suivantes :

- comment différencier les processeurs surchargés des processeurs déchargés ? Symétriquement à l'algorithme précédent, nous considérons qu'un processeur dont la charge est supérieure au seuil $T = 1$ est surchargé.

- Comment déterminer l'état courant du système distribué ? Comme pour l'algorithme serveur initiative, afin d'éviter des communications inutiles, chaque processeur gère en mémoire locale une table d'indicateurs de charge. La table est mise à jour à l'aide la méthode proposée pour la stratégie serveur initiative.

- Quelle est la méthode d'appariement? Un processeur surchargé P_i propose de partager sa charge avec le processeur P_{i+1}. Si la proposition échoue car P_{i+1} est aussi surchargé, alors le processeur P_i interroge le prochain processeur non marqué dans sa table d'indicateurs de charge. Ce principe est appliqué jusqu'à ce que tous les processeurs aient été interrogés ou que le processeur P_i ait réussi à partager sa charge. À chaque interrogation, la table est mise à jour. Quand le processeur P_i parvient à partager sa charge, il a momentanément terminé sa phase d'équilibre de charge.
- Quelle est la méthode de transfert? Quand un processeur surchargé a trouvé un processeur oisif, il lui transmet S tâches. S est un seuil à déterminer expérimentalement.

Algorithme 3.2 *algorithme source initiative*

```
fonct Source (i, n, T) : Boolean
                              /* i : numéro du processeur, n : nombre de processeurs, T : seuil */
  Cible ⟵ (i + 1) % n
  Succès ⟵ faux
  Fin ⟵ faux
  tant que (¬Succès) ∧ (¬Fin) faire
          si ¬Marqué (Cible) alors
            si DemandeCharge (Cible) < T alors
              Succès ⟵ vrai                    /* le processeur Cible est oisif */
            sinon
               Marque (Cible)
               Cible ⟵ (Cible + 1) % n
               si Cible = i alors Fin ⟵ vrai finsi
            finsi
          finsi
  fintant que
  si Succès alors
    EnvoyerCharge (Cible) /* le processeur i peut partager sa charge avec le processeur Cible */
    retourner vrai
  sinon retourner faux
  finsi.
```

Un exemple d'exécution de l'algorithme source initiative est présenté à la figure 3.4.

4.3.2.2 Propriétés de l'algorithme source initiative.

Proposition 3.8 *(Complexité en communication)*
Le nombre de messages générés par l'algorithme source initiative est en $O(n \times p)$ où p est le nombre de processeurs et n le nombre de tâches à calculer.

Preuve : le nombre maximal de communications est généré quand tous les processeurs sont surchargés. Alors, à chaque phase d'équilibre de charge, *i. e.* avant de calculer une tâche, chaque processeur surchargé engendre exactement $p - 1$ messages car il doit interroger chacun des processeurs distants. Comme il y a n tâches à calculer, le nombre total de messages est égal à $(p - 1) \times n$. □

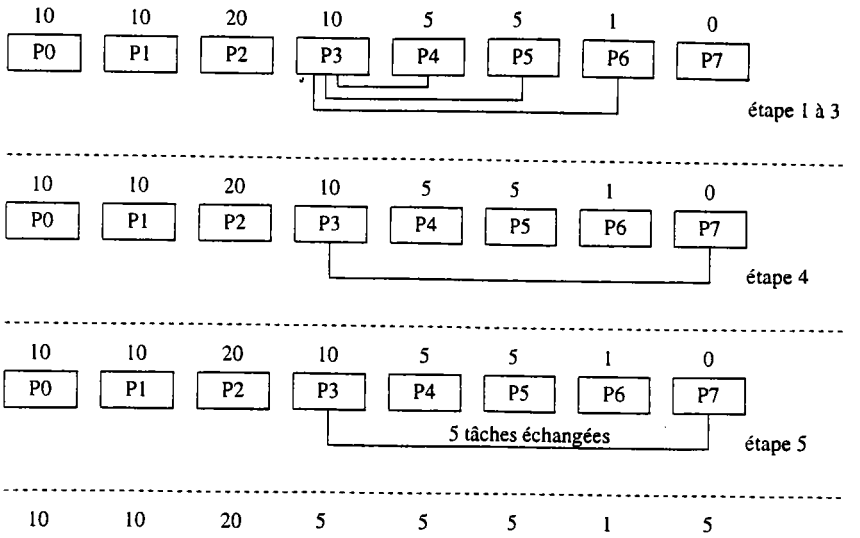


FIG. 3.4 –*exemple d'exécution de l'algorithme source initiative, 8 processeurs*

Remarque : à la différence de la stratégie serveur initiative, quand les $p-1$ requêtes d'un processeur sont rejetées, il n'est pas possible de détecter la terminaison de l'algorithme.

Proposition 3.9 *(Comparaison des stratégies source initiative et serveur initiative) La complexité en temps dans le meilleur cas de l'algorithme source initiative est supérieure à la complexité en temps dans le meilleur cas de l'algorithme serveur initiative.*

Preuve :

- soit t_1, la complexité en temps du cas le plus favorable de l'algorithme source initiative ;
- soit t_2, la complexité en temps du cas le plus favorable de l'algorithme serveur initiative ;
- soit t_{seq} le temps d'exécution séquentiel de l'application $\mathcal{A}$ de type FTII ;
- soit n le nombre de tâches de A ;
- soit t_n le temps d'une phase de négociation ;
- soit t_{max} le temps de calcul de la tâche la plus coûteuse ;
- soit p le nombre de processeurs.

□

Le cas le plus favorable de l'algorithme source initiative est atteint quand tous les processeurs disposent toujours de tâches à traiter : cela veut dire que l'équilibre de charge est *a priori* parfait :

$$t_1 \simeq \frac{t_{seq}}{p} + \frac{n}{p} \times (p-1) \times t_n.$$

Chaque processeur essaie inutilement de partager sa charge avant de débuter le calcul d'une nouvelle tâche. Le coût de chaque tentative infructueuse est de $(n-1)$ requêtes.

Le meilleur cas de l'algorithme serveur initiative est identique au précédent : cela veut dire qu'il n'y a aucun processeur déchargé et il n'y aura donc aucune phase d'équilibre de charge :

$$t_2 \simeq \frac{t_{seq}}{p}. \square$$

4.3.2.3 Analyse de la stratégie source initiative. La stratégie source initiative a un comportement opposé à celui de la stratégie serveur initiative étudiée précédemment. Le rééquilibrage de la charge est à l'initiative des processeurs surchargés qui cherchent à partager leur charge avec des processeurs oisifs. Quand un processeur est surchargé, il déclenche une nouvelle phase d'équilibre de charge. Afin de trouver un processeur avec qui s'appareiller, il interroge les processeurs distants jusqu'à trouver un processeur dont la charge est nulle. Si le processeur P_j dont la charge est $w_j = 0$ est interrogé par le processeur P_i, il envoie $\frac{w_i}{2}$ à P_j.

Construction de la matrice $M^{(k)}$ à l'instant $t^{(k)}$. Une phase d'équilibre de charge est déclenchée si un processeur P_i a consommé une tâche à l'étape $k-1$ ($a_i^{k-1} = 1$) et sa charge est supérieure ou égale à 2 ($w_i^{k-1} > 1$), cette condition peut s'exprimer ainsi :

$$\exists i |\ a_i^{k-1} = 1 \text{ et } w_i^{k-1} > 1. \tag{3.2}$$

Si P_i initie une nouvelle phase d'équilibre de charge à l'étape k (la condition 3.2 est vérifiée), soit P_i ne peut partager sa charge, soit il trouve un processeur oisif.

Cas 1 : si $a_i^{k-1} = 1$, $w_i^{k-1} > 1$ et $\forall j$, $w_j^{(k)} \neq 0$ alors $M^{(k)} = Id$.
En effet, tous les processeurs sont chargés, P_i ne peut partager sa charge, mais il perturbe néanmoins le comportement de l'application car il émet $p-1$ requêtes qui sont toutes rejetées.

Cas 2 : $a_i^{k-1} = 1$, $w_i^{k-1} > 1$ et $\exists\, !j \mid w_j^{(k)} = 0$, $d(i,j) = (j-i)\%p$ est minimum.
P_i a trouvé le processeur déchargé P_j le plus proche de lui. La matrice $M^{(k)}$ est construite de la même manière que pour la stratégie serveur initiative.
P_j reçoit la moitié des tâches du processeur P_i. Donc :

$$\alpha_{ij} = \alpha_{ji} = \frac{1}{2}.$$

Toutes les autres entrées des lignes i et j sont nulles (à l'exception de α_{ii} et α_{jj}) :

$$\forall l \neq i \text{ et } l \neq j,\ \alpha_{il} = \alpha_{jl} = 0.$$

Il n'y a pas d'échange entre les autres processeurs, donc toutes les entrées de la matrice (à l'exception des éléments de la diagonale) sont nulles :

$$\forall l \neq i \text{ et } l \neq j,\ \forall m \neq l,\ \alpha_{lm} = 0.$$

La charge globale du système est conservée :

$$\forall l,\ \alpha_{ll} = 1 - \sum_{m \neq l} \alpha_{lm}.$$

Qualité de la répartition de charge obtenue. Le comportement du système distribué est complètement déterminé par l'équation matricielle :

$$w^{(k+1)} = M^{(k)} w^{(k)} - A^{(k)}.$$

En particulier, la stratégie source initiative est modélisée par la matrice $M^{(k)}$. Nous pouvons étudier son influence sur le système distribué, et en déduire quelques propriétés.

Proposition 3.10 *La stratégie source initiative améliore la répartition des tâches.*

Preuve : Il suffit de montrer que $q(k)$ est décroissante. La technique de preuve utilisée pour démontrer la proposition 3.6 concernant la stratégie serveur initiative est appliquée pour démontrer ce résultat. □

Proposition 3.11 *La stratégie source initiative est plus efficace si le processeur surchargé P_i est le processeur dont la charge est maximale.*

Preuve : La démonstration est analogue à la démonstration de la proposition 3.7. □

Coût d'une phase d'équilibre de charge. Quand la charge est équilibrée par la stratégie source initiative ($M^{(k)} \neq Id$), il est nécessaire d'estimer le coût de cette phase. L'étude du nombre de messages de contrôle induits permet de mesurer la durée de cette phase. En effet, comme dans le cas de la stratégie serveur initiative, si le processeur P_i s'apparie avec P_j, nous pouvons déduire que P_i a émis $(j-i)\%p$ requêtes. De plus $\frac{w_i^{(k)}}{2}$ tâches sont envoyées par P_i en direction du processeur oisif. Donc, la valeur de $t^{(k+1)}$ est déterminée par l'équation :

$$t^{(k+1)} = t^{(k)} + 2\left((j-i)\%p\right) t_{com} + \frac{w_i^{(k)}}{2} t_{com} l.$$

Comme pour la stratégie serveur initiative, chaque processeur n'est pas pénalisé de la même façon.

Remarque : Quand P_i initie une phase d'équilibre de charge et qu'il ne peut trouver un processeur avec qui s'appareiller ($M^{(k)} = Id$), il faut tout de même considérer cette étape. En effet, P_i émet $p-1$ requêtes infructueuses qui ont perturbé le comportement global du système. À la différence de la stratégie serveur initiative, ce cas se produit un nombre important de fois (au plus $n-p$), le système est donc fortement pénalisé par ces échecs successifs.

4.3.3 Un nouvel algorithme hybride

Nous avons présenté dans les paragraphes précédents un algorithme source initiative et un algorithme serveur initiative ainsi que leurs propriétés théoriques. Il a été montré que les algorithmes source initiative sont très efficaces quand la charge du système est faible à moyenne et que les algorithmes serveur initiative sont particulièrement adaptés quand la charge est élevée [ELZ 86c]. Quand l'ensemble des tâches est connu au début de

l'exécution, comme dans notre étude, les algorithmes source initiative sont très efficaces en fin d'exécution alors que les algorithmes serveur initiative le sont en début d'exécution. Nous proposons alors de concevoir un nouvel algorithme hybride qui conserve les avantages des stratégies source et serveur initiative. Cet algorithme est basé sur une variation temporelle de sa stratégie: au début de l'exécution, il se comporte comme l'algorithme serveur initiative et modifie sa stratégie pour se comporter comme l'algorithme source initiative en fin d'exécution (voir algorithme 3.3). La décision de modifier la stratégie est prise dynamiquement en fonction de l'état global du système.

Les principes mis en œuvre par cet algorithme sont les suivants:

- tous les processeurs appliquent la stratégie serveur initiative au début de l'exécution.
- Dès qu'un processeur détecte que plus de k % des processeurs sont déchargés, il modifie sa stratégie. k est un seuil qu'il est nécessaire de déterminer expérimentalement.
- Tous les processeurs suivent la stratégie source initiative afin de terminer le traitement de l'ensemble des tâches.

4.3.3.1 Propriétés de l'algorithme hybride.

La complexité de l'algorithme hybride est directement héritée de celle des stratégies source et serveur initiative.

Proposition 3.12 *(Complexité en communication)*
la complexité en communication de l'algorithme hybride est en $O(n \times p)$ où p est le nombre de processeurs et n le nombre de tâches à calculer.

Preuve : Comme l'algorithme hybride met en œuvre les stratégies source et serveur initiative, il ne peut avoir une complexité pire que celle de ces deux méthodes. □

Algorithme 3.3 *Algorithme hybride*

```
proc Hybrid (i, n, T)
                                  /* i : numéro du processeur, n : nombre de processeurs, T : seuil */
  si (Strategie = ServeurInitiative) ∧ (Charge(i) < T) alors
    Serveur(i, n, T)
    si NbMessages > k.n alors
      Strategie ⟵ SourceInitiative /* La stratégie est modifiée pour tous les processeurs */
    finsi
  sinon
    si (Strategie = SourceInitiative) ∧ (Charge(i) > T) alors
      Source(i, n, T)
    finsi
  finsi.
```

4.3.3.2 Analyse de la stratégie hybride.

La stratégie hybride se comporte au début de l'exécution comme la stratégie serveur initiative, puis elle se modifie pour adopter le comportement de la stratégie source initiative en fin d'exécution. Le but de cette méthode est de réduire le nombre de phases d'équilibre de charge initiées inutilement.

Construction de la matrice $M^{(k)}$ à l'instant $t^{(k)}$. Tant que la stratégie d'équilibre de charge mise en œuvre est de type serveur initiative, la matrice $M^{(k)}$ est construite en appliquant la méthode proposée au paragraphe 4.3.1.3. Quand la stratégie est modifiée, la construction de $M^{(k)}$ est déduite en utilisant la matrice $M^{(k)}$ construite pour la stratégie source initiative.

Remarque : quand la stratégie hybride est appliquée, pratiquement toutes les phases d'équilibre de charge sont utiles, car il est presque toujours possible de former une paire pour transférer des tâches. En effet, quand la stratégie serveur initiative est appliquée, la moitié des processeurs est interrogée dans le cas le plus défavorable, ensuite la stratégie est modifiée. Dans cette situation, un grand nombre de processeurs sont déchargés et donc la stratégie source initiative est très efficace.

Qualité de la répartition de charge obtenue. Comme la stratégie hybride est un compromis entre les stratégies serveur et source initiative, elle hérite des propriétés des méthodes qui la composent.

Proposition 3.13 *La stratégie hybride améliore la répartition des tâches.*

Preuve : nous savons que la stratégie serveur initiative améliore la répartition des tâches. Ainsi, quand la stratégie appliquée par la méthode hybride est de type serveur initiative, la répartition des tâches est plus équitable. De même, la stratégie source initiative améliore cette répartition. C'est pourquoi, l'algorithme hybride améliore aussi la répartition quand il se comporte comme la stratégie source initiative. Donc, dans tous les cas, la stratégie hybride améliore la répartition des tâches dans le système distribué. □

Proposition 3.14 *La stratégie hybride est plus efficace quand la paire de processeurs qui s'apparient comporte le processeur dont la charge est la plus importante du système parallèle.*

Preuve : en début d'exécution la stratégie appliquée est une méthode serveur initiative. Quand le processeur déterminé par la stratégie serveur initiative pour partager sa charge avec le processeur à l'initiative de cette phase est le plus chargé, la répartition obtenue est la meilleure possible pour cet algorithme (d'après la proposition 3.7).

De même si le processeur à l'origine d'une phase de type source initiative est le plus chargé, la répartition obtenue est la meilleure possible (d'après la proposition 3.11). □

Coût d'une phase d'équilibre de charge. Comme pour les algorithmes serveur et source initiative, le coût de chaque phase d'équilibre de charge est déterminé en fonction du nombre de messages émis. Les processeurs les plus pénalisés sont les processeurs à l'origine de ces phases, alors que les processeurs qui n'y participent pas ne voient pas leur comportement modifié. Comme pour déterminer $M^{(k)}$, le coût de chaque étape est déduit des analyses des méthodes source et serveur initiative.

4.4 Un algorithme semi-distribué

Les stratégies centralisées de type client-serveur sont très efficaces car la charge est distribuée au niveau global. Cependant, quand le nombre de processeurs est important, le serveur peut devenir un goulot d'étranglement. Inversement, les stratégies distribuées évitent cet inconvénient, mais le nombre de messages de contrôle induits par ces stratégies est plus important.

C'est pourquoi, nous proposons un nouvel algorithme semi-distribué [Kra 98a]. Cet algorithme conservera les avantages des stratégies centralisées et distribuées : il évitera le problème du goulot d'étranglement tout en maintenant à un niveau raisonnable le nombre de messages de contrôle.

I. Ahmad and A. Ghafoor ont proposé un algorithme semi-distribué [AG 91]. Dans cette approche, la stratégie utilise un contrôle à deux niveaux afin de distribuer équitablement la charge entre les processeurs. Le système distribué est partitionné en un ensemble de *sphères*. Ainsi, chaque processeur appartient à une unique sphère indépendante des autres. Le processeur central de chaque sphère applique une stratégie de type client-serveur afin d'équilibrer la charge entre les processeurs dont il est responsable. Il maintient un niveau d'information satisfaisant avec un surcoût en communication raisonnable car le nombre de processeurs dans une sphère est réduit. Si la charge doit être équilibrée à un niveau global, un algorithme source initiative est appliqué par chaque processeur responsable d'une sphère. Comme des tâches sont engendrées dynamiquement par l'application, le choix d'une stratégie source initiative est adapté pour résoudre le problème auquel sont confrontés les auteurs. En effet, dans ce contexte, Eager *et al.* ont montré que les stratégies source initiative sont plus efficaces que les algorithmes serveur initiative quand la charge globale du système est faible à moyenne [ELZ 86c]. Quand des tâches sont engendrées dynamiquement, les algorithmes serveur initiative doivent autoriser la migration de tâches pour être aussi efficaces que les stratégies source initiative. Cependant, le coût d'une migration est en général supérieur au coût d'un transfert d'une tâche dont l'exécution n'a pas encore débutée.

Dans notre cas, l'équilibre de charge ne concerne que la répartition d'une seule application parallèle dont toutes les tâches sont connues en début d'exécution. Comme nous avons montré que les stratégies serveur initiative sont plus performantes que les stratégies source initiative et aussi efficaces que les stratégies centralisées, nous proposons un nouvel algorithme semi-distribué de type serveur initiative afin de réduire le surcoût en communication induit par la gestion dynamique de la répartition de la charge.

4.4.1 Principe

Le système est divisé en un ensemble fixe de régions. Les régions sont construites de la façon suivante : supposons que le système distribué soit composé de p processeurs notés P_0 à P_{p-1}. Le système est composé de r régions nommées R_0 à R_{r-1}. Toutes les régions ont la même taille, elles sont composées de $m = \frac{p}{r}$ processeurs (p doit être un multiple de r). Ainsi, le processeur P_i appartient à la région $R_{i\ div\ m}$ où *div* est la division entière. P_i est le $(i\ \%\ m)$ ème processeur de sa région, où % est le reste de la division entière (*modulo*). La stratégie semi-distribuée équilibre la charge à deux niveaux : à un niveau local et à un niveau global.

À un niveau local, la charge est équitablement distribuée entre les processeurs qui

appartiennent à la même région. Nous proposons d'appliquer la stratégie serveur initiative présentée au paragraphe 4.3.1. Ainsi, quand un processeur est déchargé, il essaie de trouver un processeur surchargé dans sa région. Tant qu'il y aura des tâches à calculer dans une région, la charge sera uniquement équilibrée au niveau local.

Quand un processeur ne peut trouver aucune tâche à calculer dans sa région, une phase d'équilibre de charge globale est nécessaire (voir l'algorithme 3.4). Pour chaque région, un processeur est responsable de l'équilibre de charge au niveau global. Au départ, le premier processeur de chaque sphère est désigné comme responsable, par exemple le processeur P_0 est responsable de la région R_0 et le processeur $P_{k\times m}$ est responsable de la $k$ème région, $(k < r)$. Quand une phase globale est requise, le processeur responsable de la région déchargée applique une stratégie serveur initiative. Supposons que le processeur $P^{(i)} = P_{i\times m}$, responsable de la région R_i cherche du travail pour sa région. $P^{(i)}$ émet une requête en direction du processeur $P^{(i+1)}$, responsable de la région R_{i+1}. Si la charge de $P^{(i+1)}$ est supérieure à $T = 1$ tâches, la requête est acceptée. En effet, si $P^{(i+1)}$ n'est pas oisif, les autres processeurs de la région R_{i+1} sont aussi chargés, car ils appliquent tous la stratégie locale de type serveur initiative. Si cet essai ne peut aboutir, le processeur $P^{(i)}$ essaie de trouver des tâches disponibles au niveau de la région R_{i+2}. Ce principe est appliqué jusqu'à ce que une demande soit acceptée, ou bien que toutes les régions soient déchargées. Si la phase d'équilibre de charge globale échoue, la fin de l'exécution de l'application parallèle est détectée.

Si la phase d'équilibre de charge globale appliquée par la région R_i détermine que la région R_j est chargée, alors chaque processeur de R_j envoie la moitié de sa charge en direction d'un processeur de la région R_i. Ainsi, le processeur $P_{j\times m+k}$ partage sa charge avec le processeur $P_{i\times m+k}$, $0 \leq k < m$. De cette façon, la charge est uniformément distribuée entre tous les processeurs des régions R_i et R_j.

Algorithme 3.4 *Algorithme serveur initiative global*

```
fonct Global (i, r, T) : Boolean
                                  /* i : numéro du processeur, r : nombre de régions, T : seuil */
  Cible ⟵ (i + 1) % r
  Succès ⟵ faux
  Fin ⟵ faux
  tant que (¬Succès) ∧ (¬Fin) faire
          si ¬Marqué (Cible) alors
            si DemandeGlobalTravail (Cible) > T alors
              Succès ⟵ vrai                    /* la région Cible est surchargée */
            sinon
               Marque (Cible)
               Cible ⟵ (Cible + 1) % r
               si Cible = i alors Fin ⟵ vrai finsi
            finsi
          finsi
  fintant que
  si Succès alors
    EnvoyerGlobalCharge (Cible)   /* la région Cible peut partager sa charge avec la région i */
    retourner vrai
  sinon retourner faux                     /* toutes les tâches ont été calculées */
  finsi.
```

4.4.2 Deux optimisations de la stratégie semi-distribuée

La stratégie présentée peut avoir un comportement non souhaitable dans deux cas :

1. un processeur responsable d'une région manque d'information pour répondre à une requête globale ;
2. un processeur responsable d'une région calcule sa dernière tâche pendant que les autres processeurs sont oisifs ;

cas 1 : supposons que le processeur $P^{(i)}$, responsable de la région R_i, calcule sa dernière tâche pendant que le processeur $P^{(j)}$ émet une requête en sa direction. $P^{(i)}$ ne peut partager sa charge puisqu'il n'a plus qu'une seule tâche et $P^{(j)}$ considère donc que la région R_i est oisive. Cependant, $P^{(i)}$ n'a pas encore appliqué la stratégie au niveau local et il est donc possible que tous les autres processeurs de la région R_i soient chargés. C'est pourquoi le processeur $P^{(i)}$ doit répondre à $P^{(j)}$ qu'il ne connaît pas l'état de sa région. Ainsi, $P^{(j)}$ pourra envoyer une nouvelle requête un peu plus tard.

cas 2 : si $P^{(i)}$ calcule la dernière tâche de la région R_i, pendant que les autres processeurs sont déchargés, aucune phase d'équilibre de charge globale ne peut être initiée bien que ce soit nécessaire. Pour éviter cet inconvénient, nous proposons d'utiliser un jeton dans le but de désigner dynamiquement le processeur responsable de chaque région. Au début de l'exécution, le premier processeur de chaque région possède le jeton. Ainsi, il y a exactement r jetons à chaque instant dans le système. Quand le processeur P_i, reçoit une requête au niveau local, il transmet le jeton au processeur P_j dont il ne peut satisfaire la demande s'il a le jeton de sa région et qu'il calcule sa dernière région. En effet, P_i ne peut partager sa charge avec P_j, mais il n'est pas encore complètement déchargé et il ne peut donc initier une phase d'équilibre de charge globale. Grâce à cette optimisation, P_j pourra appliquer la stratégie au niveau global s'il ne trouve pas de tâches disponibles dans sa région.

Grâce à ces deux optimisations, nous sommes sûrs que notre algorithme est capable de trouver n'importe quelle tâche qui peut être redistribuée que ce soit au niveau local comme au niveau global.

4.4.3 Une structuration optimale

Dans la stratégie semi-distribuée, il est important de choisir judicieusement la façon dont le système de processeurs est structuré. Formellement, il s'agit de choisir le nombre de régions r de manière à minimiser le coût en communication de notre méthode. Nous allons donc exprimer la complexité en communication en fonction de r de la stratégie semi-distribuée.

4.4.3.1 Complexité de la stratégie semi-distribuée. Nous supposons que le système comporte p processeurs. Chaque processeur appartient à une des r régions. Toutes les régions sont identiques : chacune comporte m processeurs. Par hypothèse, on obtient :

$$p = r \times m.$$

Proposition 3.15 *La complexité en communication de la stratégie locale de l'algorithme semi-distribué est en* $O(m^2)$.

Preuve : En effet, le cas le plus défavorable intervient quand il n'y a plus de tâche à distribuer dans la région. Comme la stratégie serveur initiative est appliquée, chaque processeur émet $m - 1$ requêtes en direction de ses voisins. Donc le nombre total de messages échangés est $m \times (m - 1)$. □

Proposition 3.16 *La complexité en communication de la stratégie globale de l'algorithme semi-distribué est en* $O(r^2)$.

Preuve : Le même raisonnement que pour la démonstration au niveau local peut être tenu. Ainsi, chaque processeur responsable d'une des r régions émet $r - 1$ requêtes en direction des autres régions. Donc le nombre total de messages induits par la stratégie globale est $r \times (r - 1)$. □

Proposition 3.17 *La complexité en communication de l'algorithme semi-distribué est en* $O(\frac{p^2}{r} + r^2)$.

Preuve : Quand toutes les tâches du système ont été consommées, chaque processeur essaie de trouver du travail dans sa sphère. Donc il y a $m \times (m - 1)$ messages échangés dans chaque région (voir théorème 3.15). Pour les r régions, la stratégie locale a engendré $r \times m \times (m - 1)$ messages.

Pour chaque région, un processeur applique la stratégie serveur initiative au niveau global. Comme exprimé dans le théorème 3.16, le nombre total de messages induits est égal à $r \times (r - 1)$.

C'est pourquoi, le nombre de messages induits par la stratégie semi-distribuée est :

$$r \times m \times (m - 1) + r \times (r - 1).$$

Comme $m = \frac{p}{r}$, nous pouvons déduire le surcoût de la stratégie dynamique en fonction du nombre de processeurs (p) et du nombre de régions (r) :

$$f(p,r) = \frac{p^2}{r} - p + r^2 - r.$$

□

4.4.3.2 Minimiser la complexité en communication. La stratégie semi-distribuée sera aussi efficace que possible si le nombre de messages échangés est minimum. Comme le nombre de messages induits est exprimé par la fonction $f(p,r) = \frac{p^2}{r} - p + r^2 - r$, il est nécessaire de dériver $f(p,r)$ afin de déterminer son minimum.

$$\frac{\partial f}{\partial r} = -\frac{p^2}{r^2} + 2r - 1 \text{ car } p \text{ est une constante.}$$

TAB. 3.1 –*Structuration optimale du système distribué*

Nombre de processeurs (p)	32	64	128
Nombre de régions , $k = 1$	8.17	12.87	20.33
Nombre de régions , $k = 2$	6.52	10.25	16.17
Nombre de régions , $k = 3$	5.72	8.97	14.15

En résolvant l'équation $\frac{\partial f}{\partial r} = 0$, nous déduisons une subdivision optimale du système distribué. Les résultats obtenus sont présentés dans le tableau 3.1.

Si un message échangé au niveau global est plus coûteux qu'un message local par un facteur k, le coût en communication de la stratégie semi-distribuée peut être exprimé ainsi :

$$f(p,r) = \frac{p^2}{r} - p + r.k\,(r-1) \text{ et}$$

$$\frac{\partial f}{\partial r} = -\frac{p^2}{r^2} + 2kr - k.$$

4.4.4 Analyse de la méthode semi-distribuée

La stratégie semi-distribuée applique une stratégie de type serveur initiative à deux niveaux : local et global. Quand la charge est répartie au niveau global, plusieurs paires de processeurs échangent des tâches simultanément. Pour cette étude, nous ignorons les optimisations proposées pour cet algorithme. En particulier, le processeur responsable d'une région est toujours le premier processeur de la région considérée. De plus, on suppose que ce processeur est toujours capable de répondre à une requête globale.

4.4.4.1 Construction de la matrice $M^{(k)}$ à l'instant $t^{(k)}$. Nous nous intéressons à la construction de la matrice $M^{(k)}$ quand une phase d'équilibre de charge est nécessaire (un processeur est oisif) et cette phase induit une redistribution effective des tâches (il existe un processeur surchargé). Dans tous les autres cas, $M^{(k)}$ est la matrice identité.

Cas 1 : un processeur déchargé trouve du travail en local. On peut exprimer cet état par la condition suivante :

$$w_i^{(k)} = 0,\ \exists\, !j \mid w_j^{(k)} > 1,\ (i/r)p \leq j < (i/r+1)p \text{ et } (j-i)\%p \text{ est minimum.}$$

Dans cette situation, le processeur P_i reçoit des tâches de P_j. La matrice est identique à la matrice de la méthode serveur initiative.

Cas 2 : un processeur déchargé ne trouve pas de travail en local, une phase globale est requise. Seul le processeur responsable de sa région peut initier une phase au niveau global. Cet état peut s'exprimer à l'aide de l'équation :

$$w_i^{(k)} = 0,\ i\%r = 0 \text{ et } \forall l \mid (i/r)p \leq l < (i/r+1)p,\ w_l^{(k)} = 0.$$

La phase d'équilibre de charge global conduit à une redistribution des charges entre la région $R_{i/r}$ et la région $R_{j/r}$ telle que :

$$\exists\, !j,\ j\%r = 0,\ w_j^{(k)} > 1 \text{ et}(j - i/r)\%p \text{ est minimum.}$$

$M^{(k)}$ est construit de la façon suivante :
Les processeurs des régions $R_{i/r}$ et $R_{j/r}$ échangent des tâches. Le processeur $P_{(i/r)p+l}$ reçoit la moitié des tâches de $P_{(j/r)p+l}$:

$$\forall l \in [0,\ n/r],\ \alpha_{(i/r+l)(j/r+l)} = \alpha_{(j/r+l)(i/r+l)} = \frac{1}{2}.$$

où p/r est le nombre de processeurs par région.
Tous les processeurs qui ne sont pas dans les régions $R_{i/r}$ ou $R_{j/r}$ conservent leur charge initiale, toutes les entrées sont nulles (à l'exception des éléments de la diagonale) :

$$\forall l \mid l \notin [(i/r)p,\ (i/r+1)p[\cup [(j/r)p,\ (j/r+1)p[,\ \text{et } \forall m \neq l,\ \alpha_{lm} = 0.$$

La charge globale du système est conservée :

$$\forall l,\ \alpha_{ll} = 1 - \sum_{m \neq l} \alpha_{lm}.$$

4.4.4.2 Qualité de la répartition de charge obtenue. Comme pour les stratégies que nous avons présentées précédemment, l'algorithme semi-distribuée permet d'obtenir une répartition plus équitable des charges entre les processeurs.

Proposition 3.18 *La stratégie semi-distribuée améliore la répartition des tâches.*

Preuve : si la charge a été modifiée localement par la stratégie semi-distribuée, elle se comporte comme la méthode serveur initiative présentée au paragraphe 4.3.1. Donc d'après la proposition 3.6, la répartition de la charge est plus proche de la distribution uniforme.

Si la charge a été modifiée au niveau global, nous pouvons considérer que $\frac{p}{r}$ processeurs ont appliqué la stratégie serveur initiative. Comme chacune des étapes améliore la répartition des charges, la stratégie semi-distribuée améliore la répartition des tâches aussi au niveau global. Nous pouvons en déduire que la stratégie semi-distribuée améliore la répartition des tâches dans tous les cas de figure. □

4.4.4.3 Coût d'une phase d'équilibre de charge. Si la phase d'équilibre à l'étape k s'est déroulée au niveau local, nous pouvons estimer le coût de cette phase grâce à l'équation introduite pour la stratégie serveur initiative. Ainsi, si le processeur P_i s'est appareillé avec le processeur P_j (P_i et P_j sont dans la même région), le coût de cette étape est :

$$t^{(k+1)} = t^{(k)} + 2\left((j-i)\%p\right)t_{com} + \frac{w_j^{(k)}}{2}t_{com}l, \text{ car } P_j \text{ a transmis } \frac{w_j^{(k)}}{2} \text{ tâches à } P_i.$$

Si P_i n'a pas trouvé de tâche disponible au niveau de sa région et s'il n'est pas le responsable de cette région, alors cette phase d'équilibre de charge est un échec ($M^{(k)} = Id$) et :

$$t^{(k+1)} = t^{(k)} + 2\frac{p}{r}t_{com}.$$

En effet, P_i a émis $\frac{p}{r}$ requêtes infructueuses en direction de ses voisins. Seuls les processeurs de la région de P_i sont pénalisés.

Si un processeur P_i initie une phase d'équilibre de charge au niveau global, nous pouvons estimer son coût de la façon suivante (la charge est équilibrée entre les régions $R_{i/r}$ et $R_{j/r}$) :

$$\begin{aligned} t^{(k+1)} = t^{(k)} \quad & + \quad temps(\text{requêtes locales}) \\ & + \quad temps(\text{requêtes globales}) \\ & + \quad temps(\text{échanges entre les 2 régions}) \end{aligned}$$

$$t^{(k+1)} = t^{(k)} + 2\frac{p}{r}t_{com} + 2\left(\left(\frac{j}{r} - \frac{i}{r}\right)\%r\right)t_{com} + \sum_{m=0}^{p/r-1}\frac{w_{j+m}^{(k)}}{2}t_{com}l.$$

En effet le processeur $P_{i+m}, (0 \leq m < \frac{p}{r})$ reçoit $\frac{w_{j+m}^{(k)}}{2}$ tâches du processeur P_{j+m}.
Si P_i ne peut trouve de région surchargée ($M^{(k)} = Id$), alors le coût est déterminé par :

$$t^{(k+1)} = t^{(k)} + 2\frac{p}{r}t_{com} + 2rt_{com}. = t^{(k)} + 2t_{com}\left(\frac{p}{r} + r\right).$$

4.5 Comparaison des performances théoriques des différentes stratégies

Dans la première partie de ce chapitre, nous avons présenté un modèle matriciel qui permet de représenter le comportement de stratégies d'équilibre de charge. En particulier, nous avons exprimé les propriétés de nos algorithmes dans ce modèle. Grâce à cette analyse, nous pouvons comparer ces différentes stratégies dans le cadre d'un ensemble homogène d'hypothèses.

4.5.1 Évaluation des perfomances à l'aide du modèle matriciel

Pour évaluer les différentes variables du modèle matriciel (temps séquentiel, temps parallèle, nombre de tâches déplacées,...), nous avons développé une application itérative qui peut être comparée à un simulateur. L'application mettant en œuvre le modèle matriciel réalise une simulation par événements discrets.

Nous avons vu que le modèle matriciel permet de discrétiser le temps en un ensemble d'étapes qui interviennent à des instants ponctuels et précis. Une étape (ou *état*) est caractérisée par une date et un ensemble de variables. Pour le modèle matriciel considéré, les variables définissant le k^e état sont :

1. Le vecteur de charge ($w^{(k)}$).
2. Le vecteur décrivant le caractère irrégulier de l'application ($A^{(k)}$).
3. La matrice $M^{(k)}$ qui caractérise la stratégie d'équilibre de charge appliquée.

Nous avons vu que l'équation matricielle $w^{(k+1)} = M^{(k)}w^{(k)} - A^{(k)}$ permet de déterminer le vecteur de charge du prochain état en fonction des variables qui caractérisent l'état courant. la matrice $M^{(k)}$ est déterminée par la méthode matricielle en fonction de la stratégie appliquée. Pour construire le vecteur $A^{(k)}$, il est nécessaire de connaître la répartition initiale de chaque tâche, ainsi que le temps de calcul qui leur est associé. Un processeur est caractérisé par trois variables :

1. le numéro de la tâche qu'il calcule actuellement ;
2. le temps encore nécessaire pour achever cette tâche ;
3. la liste des tâches en attente ;

L'état de chaque processeur est réévalué à chaque étape en tenant compte du coût de la phase d'équilibre (quand il y en a une) et du temps écoulé depuis la dernière étape. Si le processeur P_i a calculé une tâche à l'étape k, nous pouvons en déduire que $a_i^{(k)} = 1$. Ainsi le vecteur $A^{(k)}$ est construit à l'aide de l'état de chaque processeur à l'étape k.

Grâce à cette étude, nous pouvons construire la trace de l'application parallèle en fonction du dynamisme de l'application (déterminé par la répartition et le temps de calcul des tâches) et de la stratégie d'équilibre de charge. Quand cette trace est construite, nous pouvons exprimer les différents indices de performance présentés au paragraphe 3.2.

4.5.2 Étude statistique

Afin de comparer les différents algorithmes d'équilibre de charge dynamique présentés, nous étudions leur comportement dans les trois situations différentes possibles :

1. les tâches sont homogènes c'est à dire qu'elles ont toutes un temps de calcul très proche ;
2. les tâches confiées initialement à un processeur sont homogènes, mais un déséquilibre global existe ;
3. le temps de calcul de chaque tâche est irrégulier, il existe alors un déséquilibre à la fois au niveau global et au niveau local.

4.5.2.1 Présentation des lois de probabilité appliquées. Afin de comparer les différentes stratégies, nous avons choisi d'utiliser des lois de probabilité. Ainsi, le comportement dynamique de l'application parallèle est déterminé par ces lois. Une variable aléatoire X est caractérisée par sa *fonction de répartition.*

Définition 3.7 *(fonction de répartition)*
La fonction de répartition d'une variable aléatoire X est l'application F de $\mathbb{R}$ dans $[0,1]$ définie par :

$$F(x) = P(X < x).$$

Remarque : F est une fonction monotone croissante continue à gauche.

La notion de variable continue ou plus exactement absolument continue, se confond avec celle de variable admettant une densité de probabilité.

Définition 3.8 *Une loi de probabilité P_X admet une densité f si pour tout intervalle I de $\mathbb{R}$ on a :*

$$P_X(I) = \int_I f(x)dx = \int_{\mathbb{R}} \mathbb{1}_I(x)f(x)dx.$$

($\mathbb{1}_I$ est la fonction indicatrice de I).

F est alors dérivable et admet f pour dérivée. On a donc :

$$P(a < X < b) = \int_a^b f(x)dx = F(b) - F(a).$$

Le choix de la loi de *Laplace-Gauss* (ou loi *normale*) nous a permis d'étudier le comportement des stratégies quand la charge globale est régulière. De même la loi *log-normale* engendre des tâches dont les temps de calcul sont très irréguliers. Pour obtenir des variables aléatoires qui répondent précisément à ces lois, nous avons utilisé la méthode de *Monte-Carlo*. Une étude détaillée de lois de probabilités est présentée dans [Sap 90].

Loi de Laplace-Gauss. X suit une loi normale LG(m ; σ) (une loi de Laplace-Gauss de paramètres m et σ) si sa densité est :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-m}{\sigma}\right)^2}.$$

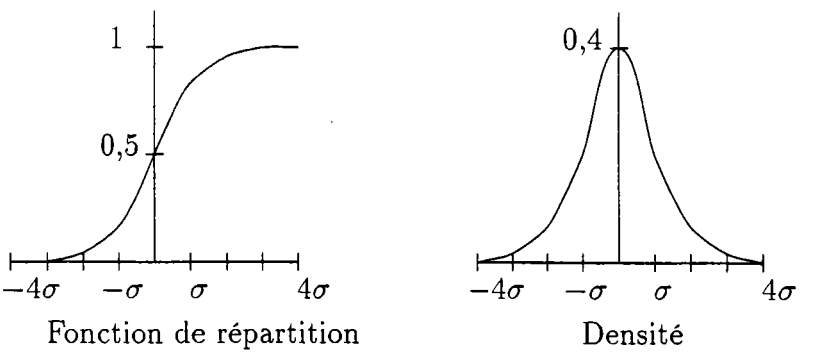
Où la moyenne de X est égale à m :

$$E(X) = m.$$

Et σ est l'écart type de X :

$$V(X) = E(X^2) - E^2(X) = \sigma^2.$$

La fonction de répartition et la densité de X sont représentées en figure 3.5.



Fonction de répartition Densité

FIG. 3.5 –*La fonction de répartition et la densité pour la loi LG.*

Loi log-normale. C'est la loi d'une variable positive X telle que son logarithme népérien suive une loi de Laplace-Gauss :

$$\ln X \sim LG(m\,;\ \sigma).$$

Sa densité s'obtient par un simple changement de variable et on trouve :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi x}} e^{-\frac{1}{2}\left(\frac{\ln x - m}{\sigma}\right)^2}.$$

De même, on détermine $E(X)$ et$V(X)$:

$$E(X) = e^{m+\frac{\sigma^2}{2}} \text{ et } V(X) = e^{2m+\sigma^2}e^{\sigma^2-1}.$$

La figure 3.6 représente la densité de la loi log-normale d'espérance 2 et d'écart-type 1 ($m \simeq -0,35$ et $\sigma \simeq 0,83$).

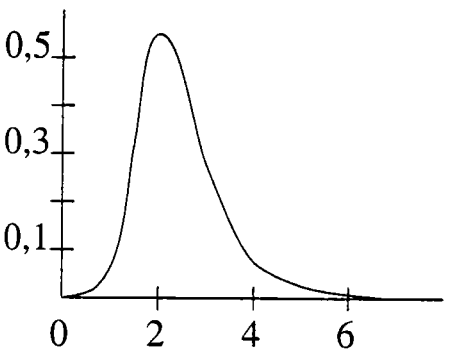


FIG. 3.6 –*La densité pour la loi log-normale.*

4.5.2.2 Comparaison des cinq stratégies. L'ensemble des jeux de données que nous avons considéré présente quelques caractéristiques communes. L'application est divisée en 1024 tâches et le système est composé de 32 processeurs, ainsi chaque processeur reçoit initialement 32 tâches à calculer. De même, le temps d'une tâche est toujours compris entre 0 et 20 secondes et il est déterminé par une variable aléatoire.

Application régulière. Nous avons construit un ensemble de jeux de données à l'aide de la loi de Laplace-Gauss. L'ensemble des tâches est dans ce cas très homogène. Quelques tâches ont un temps d'exécution très court, une grande majorité des tâches a un temps très proche de l'espérance et quelques tâches ont un temps d'exécution très long.

Nous avons fixé le paramètre moyenne $m = 10$ et le paramètre σ varie de $0,25$ à $2,75$ pour la loi $LG(m;\ \sigma)$, c'est pourquoi le temps moyen d'une tâche est 10 secondes et l'écart type varie entre 0,25 et 2,75 avec un pas de 0,25. Afin d'éviter d'analyser un cas particulier, nous avons créé 10 jeux de données pour chaque valeur de σ. C'est pourquoi 110 jeux de données ont été utilisés pour comparer les stratégies quand la charge est régulière.

Pour illustrer la régularité de l'application, nous présentons à la figure 3.7 un exemple d'exécution sans équilibre de charge dynamique. Le système est composé de 32 processeurs,

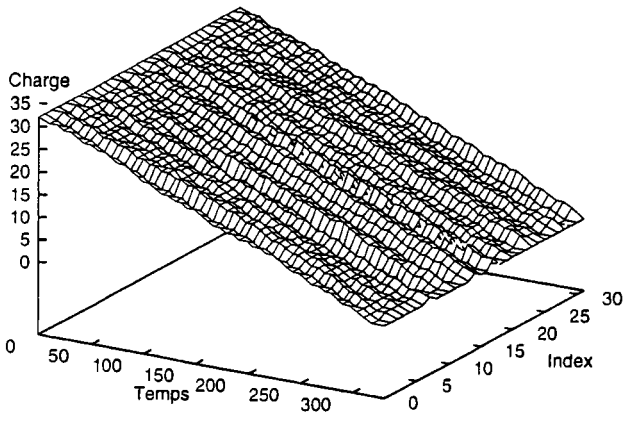


FIG. 3.7 –*Évolution de la charge sans équilibre de charge dynamique.*

les temps des tâches suivent la loi $LG(10,\ 2)$. Chaque processeur doit calculer 32 tâches. L'axe des abscisses représente le temps, l'axe des ordonnées les différents processeurs et la hauteur exprime la charge associée à chaque processeur (nombre de tâches en attente). Nous pouvons observer que tous les processeurs terminent pratiquement en même temps. Dans une telle situation, il n'est donc pas indispensable de modifier dynamiquement la répartition des charges. Si un algorithme d'équilibre de charge est appliqué, il doit être peu pénalisant pour le système, car le gain que nous pouvons espérer est très faible.

Dans un premier temps, nous avons comparé les trois stratégies distribuées serveur initiative, source initiative et hybride. Pour chaque valeur de σ, nous avons observé les mêmes éléments :

- les trois stratégies distribuées sont efficaces : elle obtiennent dans tous les cas une efficacité supérieure à la répartition statique.

- la stratégie serveur initiative déplace plus de tâches que les stratégies source et serveur initiative. La stratégie source initiative est la stratégie qui déplace le moins de tâches.

- La stratégie source initiative est la stratégie distribuée la moins performante. Nous pouvons expliquer ce résultat par le grand nombre de phases d'équilibre de charge qui échoue (voir paragraphe 4.3.2.2).

- La stratégie hybride obtient des résultats très proches de la stratégie serveur initiative. Bien que la stratégie engendre moins de messages de contrôle que les stratégies source et serveur initiative, elle ne permet pas de réel gain de performance.

Pour illustrer ces résultats, le tableau 3.2 présente les indices de performance définis par le modèle matriciel obtenu par chaque stratégie distribuée[4].

Nous pouvons aussi noter que le nombre moyen de tâches déplacées par les stratégies distribuées est faible (moins de 3 %). Ce résultat est satisfaisant, car il est inutile de

4. résultats moyens obtenus pour 10 jeux de données régis par la loi $LG(10;\ 1,75)$

TAB. 3.2 –*Comparaison des stratégies distribuées quand la charge est régulière.*

Indices de performance	serveur initiative	source initiative	hybride
temps d'exécution (t_{32}) (en s.)	326	335	332
accélération (acc)	31,3	30.5	30,8
efficacité (eff)	97,8 %	95,5 %	96,4 %
Nbre de phases d'équilibre de charge (n_{lb})	49,8	958	16,9
Nbre de proc. interrogés (n_{req})	1083	29513	103
Nbre de tâches déplacées (n_{tac})	19	9,2	17

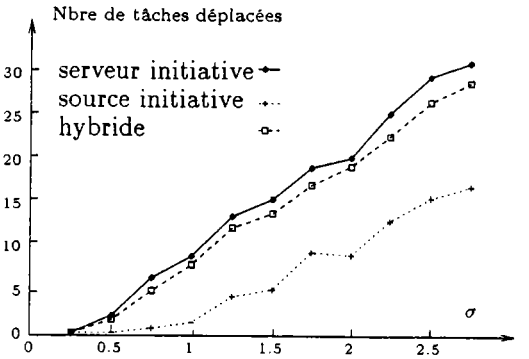


FIG. 3.8 –*évolution du nombre de tâches déplacées en fonction de la régularité.*

perturber le comportement de l'application quand la répartition des tâches est équitable. La figure 3.8 représente le nombre de tâches déplacées en fonction de σ. Plus l'application est irrégulière[5], plus le nombre de tâches déplacées est important. Ce résultat montre bien le caractère adaptatif des stratégies distribuées.

TAB. 3.3 –*Comparaison des stratégies quand la charge est régulière.*

Indices de performance	statique	serveur initiative	semi-distribuée	client-serveur
temps d'exécution (t_{32}) (en s.)	344	326	328	337
accélération (acc)	29,8	31,3	31,2	30,4
efficacité (eff)	93,0 %	97,8 %	97,5 %	95,0 %
Nbre de phases d'équilibre de charge (n_{lb})		49,8	48,22	1055
Nbre de proc. interrogés (n_{req})		1083	270	1055
Nbre de tâches déplacées (n_{tac})		19	18	1024

Pour achever l'étude du comportement des stratégies d'équilibre de charge quand la charge est régulière, nous comparons les stratégies client-serveur, semi-distribuée et serveur initiative. Nous pouvons exprimer les faits suivants :

- la stratégie client-serveur obtient dans les cas les plus réguliers une efficacité inférieure à la répartition statique.

- Les stratégies serveur initiative et semi-distribuée obtiennent des résultats très

5. cette propriété est exprimée par la valeur de σ

proches (efficacité et nombre de tâches déplacées). La stratégie semi-distribuée réduit de façon importante le nombre de messages de contrôle induits.

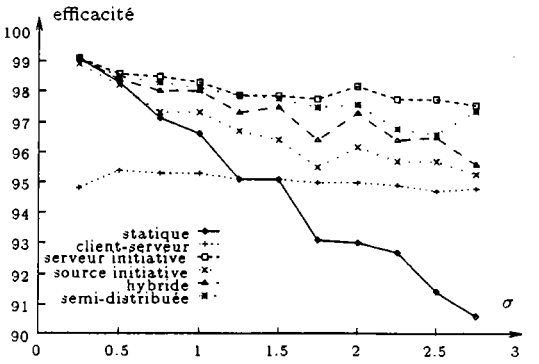


FIG. 3.9 –*efficiacité*

La table 3.3 résume les résultats obtenus pour des jeux de données de loi de Laplace-Gauss $LG(10;\ 1,75)$. Enfin la figure 3.9 montre l'évolution de l'efficacité de chaque stratégie en fonction de σ. Pour conclure, nous pouvons noter que les stratégies distribuées (serveur initiative et hybride) ainsi que la stratégie semi-distribuée sont les solutions à privilégier. Par contre les stratégies source initiative et client-serveur sont mal adaptées. En particulier, il est inutile d'avoir un serveur pour réguler la charge, alors que l'application est très régulière.

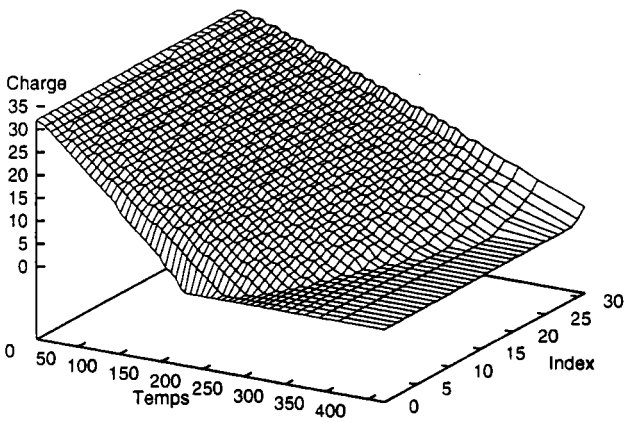


FIG. 3.10 –*Évolution de la charge sans équilibre de charge dynamique.*

Application d'irrégularité moyenne. Pour simuler le comportement dynamique d'une application irrégulière, nous avons utilisé des jeux de données dont le temps d'exécution des tâches est aussi déterminé à l'aide de la loi de Laplace-Gauss. Cependant, chaque processeur a la responsabilité de tâches dont les temps de calcul sont très proches. Ainsi le processeur P_0 est responsable des $\frac{n}{p}$ tâches les moins coûteuses et le processeur P_{p-1}

hérite des $\frac{n}{p}$ tâches les plus longues. Ainsi deux processeurs dont le numéro d'index est proche ont une charge localement équilibrée. Par contre, les processeurs P_0 et P_p ont une charge très différente.

La figure 3.10 illustre ce cas d'étude (il n'y a pas d'équilibre de charge dynamique et chaque processeur calcule les tâches dont il a la responsabilité). On observe en effet que le processeur P_0 est le processeur qui termine en premier et que le processeur P_p achèvent son travail le dernier.

TAB. 3.4 –*Comparaison des stratégies distribuées quand la charge est moyennement régulière.*

Indices de performance	**serveur initiative**	**source initiative**	**hybride**
temps d'exécution (t_{32}) **(en s.)**	327	344	337
accélération (acc)	31,3	29,8	30,3
efficacité (eff)	97,9 %	93,0 %	94,8 %
Nbre de phases d'équilibre de charge (n_{lb})	176,3	908,5	113,2
Nbre de proc. interrogés (n_{req})	1922	27396	1129
Nbre de tâches déplacées (n_{tac})	213,3	91,3	159

Nous avons à nouveau commencé par comparer les 3 stratégies distribuées entre elles. Ainsi, nous avons pu constater que la stratégie source initiative est moins efficace que les méthodes serveur initiative et hybride. Comme dans le cas où l'application est régulière, nous pouvons constater que la stratégie hybride réduit le nombre de messages de contrôle, mais elle n'est pas plus efficace que la stratégie serveur initiative. Le tableau 3.4 résume l'ensemble des résultats obtenus($\sigma = 1,75$). Enfin, la figure 3.11 met en évidence l'évolution du nombre de tâches déplacées en fonction de σ.

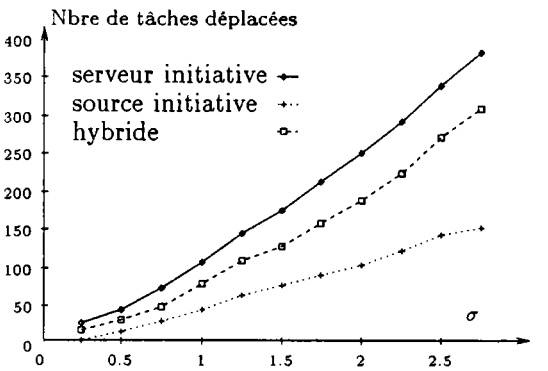


FIG. 3.11 –*évolution du nombre de tâches déplacées en fonction de la régularité.*

L'étude comparative des stratégies serveur initiative, semi-distribuée et client-serveur, nous a permis de montrer que (voir tableau 3.5) :

- les trois stratégies sont efficaces. En effet, elles permettent d'obtenir des accélérations sensiblement plus importantes que dans le cas où il n'y a pas d'équilibre de charge dynamique.

- La stratégie la plus efficace est la méthode serveur-initiative. Cependant, les stratégies client-serveur et semi-distribuée sont très proches. Pour chaque stratégie, l'efficacité obtenue est toujours supérieure à 90%.

- la stratégie semi-distribuée induit beaucoup moins de messages de contrôle que la stratégie serveur initiative.

TAB. 3.5 –*Comparaison des stratégies quand la charge est moyennement régulière.*

Indices de performance	statique	serveur initiative	semi-distribuée	client-serveur
temps d'exécution (t_{32}) **(en s.)**	446	327	333	333
accélération (acc)	22,9	31,3	30,7	30,8
efficacité (eff)	71,7 %	97,9 %	95,9	96,1 %
Nbre de phases d'équilibre de charge (n_{lb})		176,3	110,8	1055
Nbre de proc. interrogés (n_{req})		1922	579	1055
Nbre de tâches déplacées (n_{tac})		213,3	172,4	1024

Pour conclure, nous pouvons affirmer que l'utilisation de stratégies d'équilibre de charge dynamique permet d'obtenir une parallélisation efficace quand l'application est caractérisé par une irrégularité faible. En effet, nous pouvons constater que toutes les stratégies améliorent de façon importante l'efficacité de l'application par rapport au cas où il n'y a pas d'équilibrage dynamique de la charge (voir figure 3.12).

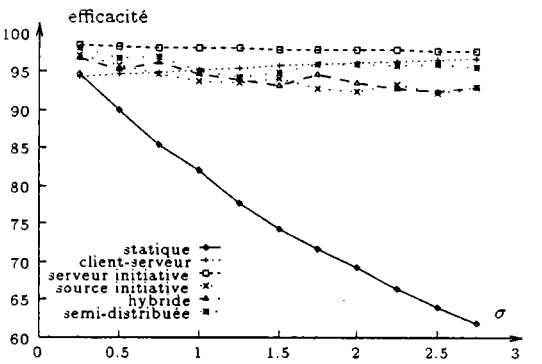


FIG. 3.12 –*efficiacité*

Application irrégulière. Afin de simuler le comportement d'une application fortement irrégulière, nous avons créé des jeux de données dont le temps de chaque tâche a été déterminé à l'aide de la loi log-normale. Pour obtenir des jeux de données d'irrégularité variable, nous avons fait varier le paramètre σ de 0,1 à 0,9 avec un pas de 0,1. Pour chaque valeur de σ, nous avons utilisé 10 jeux de données, ainsi 90 jeux de tests ont été utilisés.

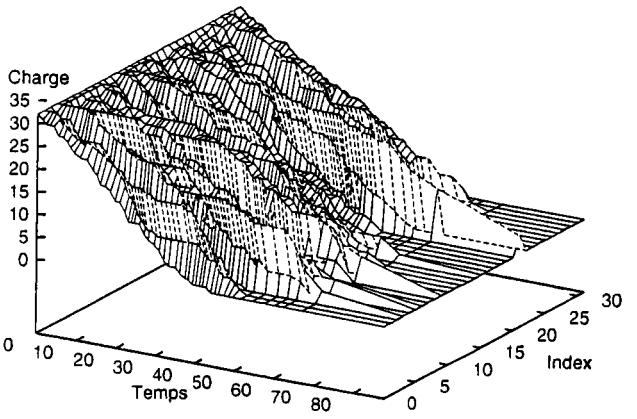


FIG. 3.13 –*Évolution de la charge sans équilibre de charge dynamique.*

Le caractère irrégulier des tâches obtenues est illustré à la figure 3.13. Chaque processeur reçoit le même nombre de tâches mais une irrégularité au niveau local existe. Ainsi, deux tâches que doit calculer un même processeur peuvent avoir des temps d'exécution très différents. De plus, une irrégularité au niveau global existe, c'est pourquoi deux processeurs différents ont très certainement une charge très variable.

TAB. 3.6 –*Comparaison des stratégies distribuées quand la charge est irrégulière.*

Indices de performance	**serveur initiative**	**source initiative**	**hybride**
temps d'exécution (t_{32}) **(en s.)**	64	69	65
accélération (acc)	27,0	25,0	26,4
efficacité (eff)	84,4 %	78,1 %	82,6 %
Nbre de phases d'équilibre de charge (n_{lb})	107,8	896,9	68,4
Nbre de proc. interrogés (n_{req})	1325	26829	245
Nbre de tâches déplacées (n_{tac})	146,3	117.2	136,7

Quand nous comparons les performances des stratégies distribuées, nous pouvons présenter les mêmes conclusions que dans les deux cas précédents. Ainsi, la stratégie source initiative est mal adaptée aux applications FTII. En effet, elle est fortement pénalisée par le nombre de messages induits. À l'opposé, la stratégie hybride est la méthode qui engendre le moins de messages de contrôle. Cependant, malgré cet avantage, quand le réseau de communication est rapide, la stratégie serveur initiative conserve une efficacité légèrement supérieure. Le tableau 3.6 présenté l'ensemble des résultats obtenus quand $\sigma = 0,9$.

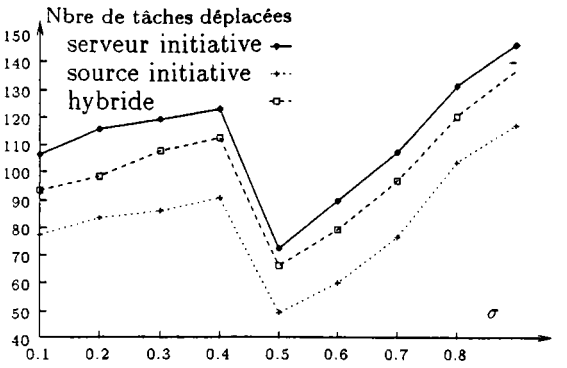


FIG. 3.14 –*évolution du nombre de tâches déplacées en fonction de la régularité.*

Le comportement adaptatif des stratégies distribuées est satisfaisant. Ainsi nous pouvons constater que, plus l'application est irrégulière, plus les stratégies distribuées déplacent de tâches pour compenser cette irrégularité (voir figure 3.14).

TAB. 3.7 –*Comparaison des stratégies quand la charge est irrégulière.*

Indices de performance	**statique**	**serveur initiative**	**semi-distribuée**	**client-serveur**
temps d'exécution (t_{32}) **(en s.)**	79	64	63	65
accélération (acc)	22,0	27,0	27,2	26,4
efficacité (eff)	68,7 %	84,4 %	85,0 %	82,5 %
Nbre de phases d'équilibre de charge (n_{lb})		107,8	99,2	1055
Nbre de proc. interrogés (n_{req})		1325	395	1055
Nbre de tâches déplacées (n_{tac})		146,3	139,3	1024

Enfin, nous pouvons constater que toutes les stratégies améliorent les résultats obtenus par la répartition statique. Cependant, la stratégie source initiative est plus en retrait des autres méthodes d'équilibre de charge dynamique. Nous pouvons aussi constater que quel que soit le caractère irrégulier, les accélérations obtenues sont très satisfaisantes (voir tableau 3.7 et figure 3.15).

5 Conclusion

Dans un premier temps, nous avons apporté une définition de la famille d'applications que nous voulons paralléliser. De plus, nous avons exprimé pourquoi une solution parallèle peut être envisagée pour les applications FTII dans le but d'améliorer leurs performances.

La seconde partie de ce chapitre présente un nouveau modèle matriciel qui permet une représentation formelle des stratégies d'équilibre de charge dans le cadre MIMD. Ce modèle prend en compte le caractère irrégulier de l'application. Dans ce cadre formel, nous avons défini plusieurs indices de performance classiques afin de comparer les différentes stratégies d'équilibre de charge dynamique.

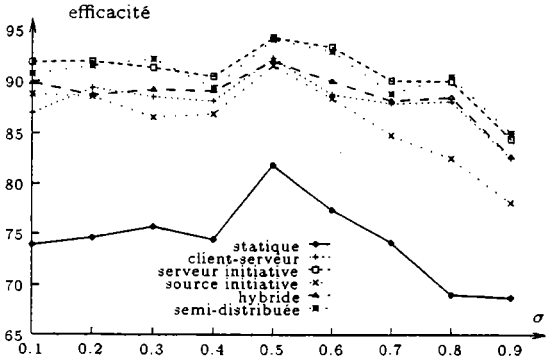


FIG. 3.15 –*évolution de l'efficacité*

Nous avons ensuite proposé cinq stratégies d'équilibre de charge qui prennent en compte les principales caractéristiques des applications FTII (nombre fini de tâches indépendantes et temps d'exécution d'une tâche imprévisible). Nous avons d'abord étudié une solution centralisée dans une architecture client-serveur. À l'aide du modèle matriciel, nous avons analysé cette première solution et nous avons estimé le coût d'une phase d'équilibre de charge. De même, nous avons proposé trois solutions distribuées : source initiative, serveur initiative et hybride. La stratégie source initiative permet aux processeurs surchargés de chercher un processeur déchargé susceptible de recevoir une ou plusieurs de leurs tâches. À l'opposé, la stratégie serveur initiative autorise les processeurs déchargés à chercher du travail. Enfin, nous avons proposé une stratégie hybride qui est basée sur une variation temporelle du comportement : dans un premier temps, la stratégie est de type serveur initiative, puis, en fin d'exécution, une stratégie source initiative est appliquée. Une analyse formelle de chaque stratégie a été menée et, pour chaque méthode, nous avons proposé une modélisation de son comportement à l'aide du modèle matriciel. Une dernière stratégie semi-distribuée a été développée afin de conserver les avantages des méthodes distribuées et des méthodes centralisées. Comme les stratégies distribuées sont facilement extensibles et que les algorithmes centralisés créent un nombre réduit de messages de contrôle, nous avons proposé un algorithme qui réalise un partitionnement logique du système parallèle ou distribué. Ainsi une stratégie de type serveur initiative est mise en œuvre à deux niveaux : au niveau des processeurs appartenant à une même région et entre les différentes régions.

Pour conclure ce chapitre, nous avons mené une étude statistique afin d'évaluer théoriquement les mérites des différentes solutions proposées pour la parallélisation des applications FTII. Nous avons constaté que les stratégies distribuées sont très efficaces, même si la stratégie source initiative obtient de moins bons résultats. La stratégie centralisée est très efficace quand le nombre de processeurs est raisonnable et que l'application est particulièrement irrégulière. Enfin la stratégie semi-distribuée semble réaliser un très bon compromis entre qualité de la répartition de la charge et nombre de messages induits.

Nous présentons dans le chapitre suivant, un environnement d'aide à la parallélisation des applications FTII. Le but de cet environnement est de proposer une première implémentation parallèle pour l'application FTII de façon systématique.

Chapitre 4

Un environnement parallèle pour les applications FTII

1 Introduction

Dans ce chapitre, nous développons un environnement parallèle pour les applications FTII [Kra 98b]. Cet environnement apporte une aide à un utilisateur désireux de paralléliser une application FTII. Il est composé de cinq parties principales (voir figure 4.1) :

1. La première partie (notée *FTII*) représente la famille des applications FTII.
2. La deuxième partie (notée *MP* pour *Machine Parallèle*) représente les caractéristiques de la machine parallèle de l'utilisateur.
3. La partie *LB* (pour *Load Balancing*) définit les différents algorithmes d'équilibre de charge dynamique qui peuvent être appliqués par l'utilisateur.
4. La partie nommée *MM* (pour *Modèle Matriciel*) représente le modèle matriciel MIMD développé dans le chapitre précédent.
5. La partie *AD* (pour *Algorithme de Décision*) aide l'utilisateur à choisir l'algorithme d'équilibre de charge dynamique le plus adapté à son problème.

2 Les applications FTII

Nous avons présenté les différentes caractéristiques des applications FTII au paragraphe 2 du chapitre 3. L'ensemble des caractéristiques des applications FTII est pris en compte au niveau de l'environnement parallèle par l'intermédiaire de la partie *FTII*.

2.1 Présentation algorithmique des applications FTII

Une application FTII est caractérisée par plusieurs propriétés que nous résumons ainsi :

- l'application est composée d'un nombre fini de tâches ;
- chaque tâche est indépendante et irrégulière.

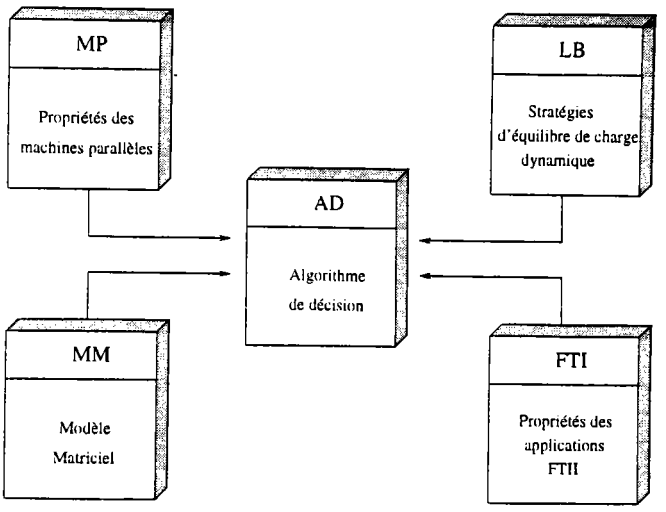


FIG. 4.1 –*Architecture de l'environnement parallèle pour les applications FTII*

Donc, il est toujours possible de proposer une algorithme séquentiel qui résout un problème FTII sous la forme de l'algorithme 4.1.

Algorithme 4.1 *Algorithme séquentiel pour une application FTII*

```
proc Resolution(n)                          /* n est le nombre de tâches */
  pour i = 1 à n faire                      /* la résolution du problème n'est pas achevée */
      TraiterTache(i)                       /* je résous le sous problème i */
  finpour.
```

Ainsi, pour que l'environnement apporte une aide efficace à la parallélisation d'une application FTII, l'utilisateur doit répondre à deux questions :

1. Découper son problème en un nombre fini de tâches.

2. Fournir une fonction permettant de résoudre une tâche identifiée par son numéro.

Les choix effectués par l'utilisateur sont importants. Il doit choisir une granularité (un nombre de tâches) adaptée à la machine parallèle qu'il veut exploiter. En effet, nous avons vu que la granularité influe sur la qualité de la répartition de la charge mais aussi sur le niveau d'utilisation du réseau d'interconnexion (paragraphe 3.2.1 du chapitre 2). Par conséquent, si le choix effectué par l'utilisateur n'est pas judicieux, l'efficacité de la solution parallèle n'est pas assurée car l'accélération théorique pour p processeurs est dépendante du temps d'exécution de la tâche la plus coûteuse (voir proposition 3.1, chapitre 3). Donc, si le temps maximal d'une tâche $\left(\max_{t=0}^{n-1} t_{cal}(t)\right)$ est largement supérieur à $\frac{t_{seq}}{p}$, l'efficacité de la solution parallèle sera faible. Par contre, si la granularité choisie par l'utilisateur est trop fine, il peut en résulter une surcharge du réseau de communication ce qui peut conduire à une dégradation des performances. Pour adapter la granularité de son problème en fonction des différents critères à considérer, l'utilisateur peut s'aider de l'analyse matricielle proposée par l'environnement.

2.2 Présentation de quelques problèmes FTII

Dans cette partie nous illustrons la famille FTII par le problème de l'ensemble de Mandelbrot et par deux exemples d'applications particulières de CAO : la *facetisation de carreaux restreints* et les *opérations booléennes*. Ces exemples nous permettent d'expliquer comment utiliser l'environnement proposé pour obtenir une solution parallèle à un problème donné. Un dernier exemple est étudié plus précisément au chapitre suivant : il s'agit du lancer de rayons.

2.2.1 L'ensemble de Mandelbrot

Lorsque on applique une opération simple aux points du plan complexe, certains points partent à l'infini, alors que les points dans l'ensemble de Mandelbrot ne se «déplace» pratiquement pas. B. Mandelbrot est à l'origine de la théorie sur les fractales et a étudié cet ensemble qui porte son nom dont la frontière est une fractale [Dew 85].

L'opération répétée indéfiniment consiste à appliquer à un nombre complexe la modification suivante : $z = z^2 + c$. La valeur initiale de z est égale à c et $c = x + yi$, x étant l'abscisse du point étudié et y son ordonnée.

Pour obtenir une représentation de cet ensemble, il suffit d'appliquer l'algorithme suivant à chaque point du plan complexe :

Algorithme 4.2 *Algorithme pour déterminer si un point fait partie de l'ensemble de Mandelbrot*

```
proc Mandelbrot(c, mi)                                  /* c = x + yi */
  z = c
  n = 0
  tant que module(z) < 2 ∧ n < mi faire                 /* si n=mi, c est dans l'ensemble */
          z = z * z + c
          n = n + 1
  fintant que.
```

Si le module de z est toujours inférieur à 2 après mi itérations, nous considérons que c est un point de l'ensemble de Mandelbrot. Dans le cas contraire, le point est extérieur à l'ensemble. Cette technique permet d'obtenir des images d'une étonnante complexité.

Cette solution au problème de l'ensemble de Mandelbrot est une application FTII, car elle vérifie l'ensemble des hypothèses de la définition 3.2 du chapitre 3. En effet, la notion de tâche peut être définie comme la vérification de l'appartenance à l'ensemble de Mandelbrot d'un point du plan complexe. Chaque tâche est indépendante et les domaines des tâches ainsi définis sont disjoints. De plus, il est possible de proposer une fonction f_d qui associe à chaque tâche un identifiant unique en considérant la fonction qui associe au point (x, y) de l'écran l'entier $x + y \times largeur$ où $largeur$ est la largeur de l'écran. Enfin, le temps de chaque tâche est imprévisible car un point peut diverger rapidement alors qu'un autre ne divergera pas ou seulement après un grand nombre d'itérations. C'est pourquoi, l'environnement pour les applications FTII permet d'obtenir une parallélisation efficace de l'application Mandelbrot.

2.2.2 Facetisation de carreaux restreints

Afin de représenter au mieux des objets du monde réel, il est parfois nécessaire d'utiliser des surfaces gauches (non planes). Les *carreaux de Bézier* permettent une représentation

de ce type de surface. Cependant, dans le cadre d'intersection entre deux objets, il est possible d'obtenir des surfaces gauches *restreintes*, c'est à dire dont le domaine paramétrique n'est pas rectangulaire. La facetisation de ce type de carreau est alors délicate. C. Nicolas a proposé un algorithme basé sur la subdivison de De Casteljau pour résoudre ce problème [Nic 95]. La méthode proposée peut être résumée par l'algorithme 4.3.

Algorithme 4.3 *Facetisation de carreaux restreints*

```
proc Facetisation(carreau)                          /* le carreau à facetiser */
  CalculProfondeurSubdivision(carreau, prof) /* l'algo détermine la profondeur de subdivision */
  pour i = 1 à 4^P faire                    /* il y a encore des sous-carreaux à facetiser */
      SubdiviserEtConstruire(carreau, i)              /* On construit le sous-carreau i */
      FacetiseSousCarreau(i)                        /* projection de la facette i en 3D */
  finpour.
```

L'algorithme calcule en premier la profondeur de subdivision à atteindre pour garantir que les facettes approcheront de façon satisfaisante la surface représentée par le carreau de Bézier restreint. Cette profondeur est fonction de la courbure de la surface. En pratique, l'auteur précise que p varie entre 5 et 9. Ainsi, pour $p = 6$, $4^p = 4096$ sous-carreaux doivent être construits et facetisés. Il est donc possible de définir la notion de tâche pour l'algorithme de facetisation comme le calcul d'un sous-carreau et de la facette associée. Si l'utilisateur trouve que le nombre de tâches est trop important pour la machine parallèle cible (au minimum $p = 4$, soit 1024 tâches) il peut choisir de traiter les sous-carreaux par petits groupes. Par exemple si $p = 4$ et un groupe est constitué de 4 sous-carreaux, l'utilisateur a défini 256 tâches. De plus, il est aisé de proposer une numérotation (basée par exemple sur le parcours en profondeur de l'arbre 4-aire de profondeur p) qui permet d'attribuer un numéro unique à chaque sous-carreau.

Ensuite, pour chaque sous-carreau de Bézier, les courbes de restriction sont subdivisées à l'aide de l'algorithme de De Casteljau. Cette phase peut avoir un temps très variable car un sous carreau peut n'avoir aucun contour de restriction. Dans ce cas, la subdivision est triviale. Par contre, si le sous-carreau considéré est fortement restreint, la phase de subdivision peut être particulièrement longue. Quand un sous-carreau a été calculé, il est facétisé. Encore une fois, suivant la complexité du sous-carreau traité, le temps nécessaire à la facetisation peut être très variable. Puis, pour chaque sous-carreau ainsi défini, une facette 3D est engendrée. Remarquons que si le sous carreau i est complètement intérieur à un contour de restriction, il n'engendrera pas de facette et il est donc inutile de le traiter. Dans ce cas, le temps d'exécution de la tâche i est pratiquement nul.

La méthode de facetisation de C. Nicolas est donc bien une application FTII. En effet, nous avons vérifié que :

- le nombre de tâches est fini ;
- une tâche est définie comme le traitement d'un sous-carreau ;
- il est possible de numéroter de façon unique chaque tâche ;
- les tâches peuvent être considérées indépendantes ;
- le temps de chaque tâche est imprévisible.

Donc, l'environnement parallèle proposé pour les applications FTII permet d'obtenir un algorithme parallèle efficace pour la facetisation des carreaux restreints.

2.2.3 Opérations booléennes

La forme d'un objet dans un système de CAO peut être conservée à l'aide d'un modèle B-rep (*Boundary Representation*) qui représente un objet à l'aide de ses frontières. Quand un objet est modélisé par le modèle B-rep, le système ne conserve que les faces qui constituent l'enveloppe de l'objet. E. Perrin a proposé un algorithme qui permet d'évaluer les opérations booléennes (union, intersection, différence) pour des objets en trois dimensions représentés à l'aide d'un modèle B-rep [Per 95]. La solution proposée réalise un traitement face par face. C'est pourquoi la résolution du problème est simplifiée car elle est ramenée dans un espace 2D. Quand toutes les faces sont planes, le principe général de l'algorithme est le suivant :

Algorithme 4.4 *Opérations booléennes*

proc *Traitement*(O1, O2, *op*) /* O1et O2 sont les deux objets et op l'opération booléenne */
pour chacune des faces de O1 et de O2 **faire**
déterminer la section de second objet par le plan de la face
calculer l'opération booléenne 2D op appliquée à la face et à la section
finpour
construire l'objet résultat avec l'ensemble des faces générés.

De façon naturelle, nous pouvons proposer de définir la notion de tâche comme le traitement d'une face par rapport au second objet considéré. Ce traitement consiste en la détermination de la section, puis le calcul de l'opération booléenne 2D. Ainsi, le nombre de tâches est fini : si $O1$ est composé de $n1$ faces et $O2$ de $n2$ faces, le nombre de tâches est égal à $n = n1 + n2$. Nous pouvons numéroter chaque tâche, par exemple, de la façon suivante : traiter la tâche i revient à considérer la face i de l'objet $O1$ par rapport à l'objet $O2$ si $i \leq n1$, traiter la face $i - n1$ de l'objet $O2$ par rapport à l'objet $O1$ sinon. Grâce à cette définition, nous pouvons reformuler la solution proposée par E. Perrin sous la forme de l'algorithme général pour une application FTII (voir l'algorithme 4.1). De plus, le temps de traitement de chaque face peut être très variable en fonction de la complexité de la face à étudier. Comme la solution de facetisation des carreaux restreints proposé par C. Nicolas, cette application est une application FTII. C'est pourquoi, l'environnement parallèle FTII peut apporter une aide à l'utilisateur désireux de proposer une solution parallèle à ce problème.

Le lecteur peut remarquer que pour ces deux derniers exemples, des calculs redondants existent. Ce problème a déjà été abordé au paragraphe 2.2.4 du chapitre 3 ; les approches générales proposées sont applicables pour les applications de facetisation et de calcul des opérations booléennes.

Dans le cas de la facetisation de carreaux restreints, l'algorithme réalise plusieurs fois les mêmes calculs quand les points situés sur une frontière délimitant deux sous-carreaux sont traités. L'utilisateur doit choisir s'il estime plus judicieux de ne pas reproduire deux fois ou non le même calcul. S'il choisit d'éviter la redondance, des communications seront nécessaires pour pouvoir complètement traiter deux sous-carreaux voisins mais calculés par deux processeurs différents. Ces communications ne sont pas prises en compte au niveau de la modélisation des applications FTII et c'est donc à l'utilisateur d'évaluer le surcoût induit pas ces communications. S'il dispose d'une mémoire partagée, il peut choisir d'y placer les résultats concernant les points frontières. Quand un processeur aura

besoin des calculs associés à un point frontière, il lui suffira de consulter l'information stockée au niveau de la mémoire partagée.

L'algorithme des opérations booléennes effectue lui aussi des calculs redondants. En effet, quand l'algorithme traite la face f_{1i} de l'objet O_1 par rapport à l'objet O_2, l'algorithme calcule une première fois l'intersection entre f_{1i} et une face f_{2j} de O_2. Quand la face f_{2j} est traitée par rapport à O_1, le calcul d'intersection entre f_{1i} et f_{2j} est réitéré. Si l'utilisateur ne souhaite pas modifier son algorithme, mais aimerait cependant éviter ces calculs, il peut choisir une solution induisant de nouvelles communications qui ne sont pas prise en compte au niveau du modèle développé pour les applications FTII. Il peut aussi, comme dans le cas de la facetisation de surface gauche, choisir d'utiliser une mémoire partagée.

2.3 Parallélisation de programmes versus parallélisation FTII

D'autres auteurs ont défini une famille d'algorithmes pour lesquels ils proposent une méthode de parallélisation. Nous distinguons les *algorithmes à pile* et *les algorithmes SDI* pour lesquels une étude à été menée. Cependant, nous constatons que dans chaque étude, les propriétés de la famille d'algorithmes sont différentes. Ainsi SDI est plutôt orienté vers les méthodes de recherche opérationnelle alors que les algorithmes à pile sont prévus pour une machine SIMD.

2.3.1 Algorithmes à pile et Structures de Données Irrégulières

C. Fonlupt a introduit brièvement dans sa thèse la notion d'*algorithme à pile* [Fon 94]. Dans le contexte SIMD, un algorithme à pile permet le traitement de chaque donnée (*objet*) indépendamment des autres et sur n'importe quel processeur. Le temps d'exécution est identique pour toutes les données, mais chaque objet peut engendrer une nouvelle donnée. Tous les objets sont regroupés dans un pile qui est distribuée. Le but de l'équilibre de charge dynamique est de distribuer équitablement cette pile sur les processeurs de la machine SIMD.

Dans [AGF+ 95, chap. 4], G. Authié *et al.* apportent une définition de l'irrégularité d'un algorithme. Intuitivement, les auteurs expliquent que l'irrégularité d'un algorithme doit être liée à la difficulté d'ordonnancement des tâches. Plus il est difficile d'ordonnancer les tâches pour obtenir une efficacité pratique satisfaisante, plus l'algorithme est irrégulier.

Le problème *TSP*[1] consiste à déterminer un ordonnancement de toutes les tâches issues d'un même ensemble afin d'obtenir une efficacité pratique optimale. Si le nombre de tâches est n et p le nombre de processeurs, les auteurs supposent que le problème *TSP* est résolu avec un coût de $\frac{n}{p}$.

L'irrégularité d'un algorithme (dans le pire des cas) est donc la plus grande des complexités du problème TSP pour toutes les exécutions possibles. Les auteurs apportent alors la définition de 3 classes d'algorithmes : les algorithmes *réguliers*, *log-réguliers* et *irréguliers*.

Enfin, les auteurs montrent qu'un algorithme peut être considéré comme irrégulier pour une granularité donnée et qu'il est possible (en utilisant un plus gros grain) de rendre l'algorithme régulier. Cependant, ils remarquent que modifier la granularité afin

1. Task Scheduling Problem.

de réduire le caractère irrégulier de l'application, peut en pratique diminuer l'efficacité de l'algorithme et qu'il est donc nécessaire de trouver un compromis.

Quand les algorithmes manipulent des structures de données irrégulières (*SDI*), il est difficile de proposer un algorithme parallèle efficace en pratique. En effet, les approches de parallélisation automatique ne sont pas en mesure de considérer cette irrégularité [AGF+ 95, chap 5.]. C. Roucairol considère les SDI dans le cadre du *branch et bound* [Rou 95]. Une structure de données irrégulières pour l'algorithme A^* est développée. M. Benaïchouche et S. Dowaji proposent un algorithme d'équilibre de charge de type semi-distribué pour le *Branch and bound* distribué. Les processeurs sont répartis dans différents groupes et chaque groupe régule sa charge en appliquant une stratégie de type client-serveur. Cette étude a conduit les auteurs à développer une bibliothèque pour la résolution de problèmes *Branch and Bound* appelée BOB^2. Ce développement permet d'étudier pratiquement différentes optimisations sur les structures de données utilisées ou bien sur les algorithmes d'équilibre de charge mis en œuvre.

2.3.2 Parallélisation de programmes

Il n'est pas possible de présenter tous les algorithmes sous la forme d'algorithmes FTII. Ainsi, si nous considérons l'algorithme 4.5, l'environnement parallèle proposé est *a priori* mal adapté. En effet, il n'est pas possible de présenter cet algorithme sous la forme générale proposée pour les applications FTII (voir algorithme 4.1).

Algorithme 4.5 *Algorithme séquentiel d'une application non FTII*

```
proc Resolution(n1, n2)                    /* n1 et n2 sont les nombres de tâches */
  pour i = 1 à n1 faire                    /* la résolution du problème n'est pas achevée */
      TraiterTache1(i)                     /* je résous le sous problème i */
  finpour
  pour i = 1 à n2 faire                    /* la résolution du problème n'est pas achevée */
      TraiterTache2(i)                     /* je résous le sous problème i */
  finpour.
```

Pour savoir s'il est possible de paralléliser facilement ce type de problème à l'aide de l'environnement, nous devons distinguer deux cas :

1. Les deux boucles sont indépendantes.

2. Il faut connaître les résultats de la première boucle pour calculer la deuxième.

Si les deux boucles sont indépendantes, nous pouvons considérer que c'est une application FTII. En effet, il suffit de définir la notion de tâche comme le regroupement des calculs associés à une tâche de la première boucle et à une tâche de la seconde boucle. Ainsi, l'algorithme 4.5 peut être présenté ainsi :

Algorithme 4.6 *Un nouvel algorithme FTII*

```
proc Resolution(n1, n2)                    /* n1 et n2 sont les nombres de tâches */
  pour i = 1 à Max(n1, n2) faire           /* la résolution du problème n'est pas achevée */
      TraiterTache1(i)                     /* je résous la première partie du sous problème i */
```

2. Bibliothèque pour la résolutions des problèmes d'Optimisation avec B&B.

TraiterTache2(i) /* je résous la seconde partie du sous problème i */
/* si n1<n2 et i>n1, TraiterTache1 est vide */

finpour.

Par contre, s'il n'est pas possible de traiter la seconde boucle indépendamment de la première, le problème est plus difficile. Dans ce cas, nous pouvons considérer chaque boucle comme deux algorithmes FTII indépendants. Ainsi la solution parallèle que nous pouvons proposer est d'exécuter en parallèle la première boucle puis de synchroniser l'ensemble des processeurs et d'exécuter en parallèle la deuxième boucle. Cependant, il n'est pas possible de garantir l'efficacité de cette parallélisation car il est possible que la synchronisation des processeurs soit très pénalisante.

De manière générale, il peut être délicat de montrer qu'une application n'est pas FTII. Une application n'est pas FTII si elle ne vérifie pas l'une des 3 propriétés suivantes :

1. Le temps d'exécution de chaque tâche est constant (ou bien prévisible).
2. Le nombre de tâches est infini.
3. Il n'existe pas d'algorithme sous la forme 4.1 qui résout le problème posé.

Si le temps d'exécution de chaque tâche est constant, l'application considérée n'est pas FTII. De plus, il est facile de paralléliser efficacement ce type d'application. En effet, comme chaque tâche est indépendante et le nombre de tâches est connu *a priori*, il suffit de proposer une répartition en modulo pour que la charge soit distribuée de manière égale parmi les processeurs du système parallèle ou distribué.

Quand le nombre de tâches est infini, l'environnement FTII ne peut apporter d'aide à l'utilisateur qui souhaite paralléliser cette application. En effet, la validation théorique des différentes solutions proposées suppose que ce nombre de tâches est fini et la convergence de nos algorithmes n'est donc pas établie dans ce cas de figure. Une extension des applications FTII peut être envisagée mais dans ce cas une nouvelle analyse des algorithmes doit être menée par exemple en utilisant la théorie sur les files d'attente. Ce travail, qui relève de l'algorithmique distribuée, ne rentre pas dans le cadre de notre étude où nous considérons uniquement le caractère irrégulier d'une seule application sur une machine parallèle.

Si l'utilisateur n'est pas en mesure de proposer un algorithme sous la forme 4.1 qui apporte une solution au problème qu'il s'est posé, l'environnement FTII ne peut pas être utilisé afin de proposer une application FTII. Cependant, ce n'est pas parce que l'utilisateur ne peut proposer un algorithme sous la forme FTII, que cette présentation de la solution n'existe pas. En effet, il peut être très difficile de montrer qu'un algorithme ne peut être présenté sous cette forme.

Dans ce cas de figure, les principes appliqués par la parallélisation automatique peuvent aider l'utilisateur à vérifier s'il est possible de présenter ou non un algorithme sous la forme générale d'une application FTII [CT 93, chap. 10], [CNR 92, chap. 17]. Les méthodes de parallélisation automatique cherchent à déterminer le graphe qui représente les dépendances existant entre les différentes tâches. Quand le but de la parallélisation est de vectoriser plusieurs boucles imbriquées (pour obtenir une exécution efficace en mode SIMD), le *vectoriseur* cherche à placer les tâches indépendantes dans les boucles les plus intérieures. Comme la synchronisation entre chaque tour de boucle est peu coûteuse sur

une machine SIMD, il semble judicieux de réaliser un grand nombre de tours de boucle afin d'utiliser un grand nombre de processeurs. Par contre, si la machine parallèle est de type MIMD, la synchronisation est très coûteuse et le *parallélisateur* cherche donc à placer les tâches indépendantes au plus haut niveau afin de réduire le nombre de synchronisations nécessaires.

Quand les tâches définies dans un algorithme sont à l'origine d'un graphe des dépendances très complexe, il est difficile d'exécuter un tel algorithme en parallèle. La parallélisation automatique applique alors des techniques de modifications afin de réduire les dépendances existantes entre les tâches, ce qui permet de dégager un parallélisme plus important de cet algorithme. Ces techniques très générales de transformation de programme peuvent aider l'utilisateur à décider si son algorithme est FTII où non. Nous avons vu au début de ce paragraphe une première technique de transformation qui permet de regrouper deux boucles consécutives en une seule. La transformation inverse est aussi possible et elle permet dans certains cas de dégager une boucle parallèle. Considérons l'exemple suivant :

Algorithme 4.7

pour i = 1 **à** n **faire**
    Proc1
    Proc2
finpour

Si *Proc1* et indépendant de *Proc2*, il est possible d'éclater cette boucle en 2 boucles consécutives. Ainsi, quand *Proc1* est fortement contraint et qu'il n'est pas possible de proposer une exécution parallèle efficace, nous dégageons un parallélisme plus important au niveau de la deuxième boucle qui traite *Proc2*.

De même, l'introduction de nouvelles variables peut conduire à un graphe des dépendances simplifié. Par exemple, il est parfois intéressant d'utiliser des variables temporaires dans une boucle exécutée en séquentiel. Cependant, comme ces variables sont utilisées à chaque tour de boucle, elles induisent des contraintes au niveau du graphe des dépendances qui rendent impossible la parallélisation de cette boucle. Il est alors possible d'utiliser un tableau pour chacune de ces variables afin de supprimer ces dépendances. Dans ce cas, la valeur d'une variable temporaire *temp* au i^e tour de boucle est égale à $temp[i]$.

Les techniques de construction du graphe des dépendances et de transformation de programmes visant à réduire les dépendances existantes entre les tâches apportent une aide à l'utilisateur qui cherche à présenter un algorithme sous la forme FTII. Cette première étape où l'utilisateur dégage le parallélisme que contient son application, lui permettra d'utiliser l'environnement FTII afin d'obtenir une exécution efficace de son application irrégulière. Nous pouvons conclure que l'environnement FTII n'est pas une méthode concurrente des paralléliseurs ou vectoriseurs, mais plutôt une méthode complémentaire qui permet d'utiliser les résultats de l'analyse obtenus grâce à la parallélisation automatique afin de proposer une parallélisation efficace d'un algorithme irrégulier de type FTII.

3 Définition d'une machine parallèle virtuelle

Cette couche logicielle (développée en C++) permet d'obtenir une vue virtuelle du réseau de communication de la machine parallèle. Grâce à cette abstraction, l'environnement

parallèle pour les applications FTII est indépendant de la machine parallèle choisie par l'utilisateur. Le réseau de communication de la machine parallèle est vu par l'utilisateur comme un graphe complet.

Dans la suite de ce chapitre, nous utilisons la méthode de G. Booch pour présenter l'architecture générale des différentes parties logicielles de l'environnement [Boo 94].

Pour que notre application parallèle FTII soit indépendante de la machine cible, nous avons développé une catégorie intitulée *MachineParallèle* qui suppose que des services minimum sont assurés par la machine cible. En contrepartie, cette catégorie fournit une interface cohérente pour accéder aux ressources de la machine (processeurs, réseau d'interconnexion) de façon transparente. Les hypothèses suivantes sont supposées vérifiées par la machine cible :

- la machine parallèle dispose d'une mémoire distribuée et donc les données sont partagées à l'aide de messages explicites ;
- le graphe d'interconnexion est connexe. Il existe toujours un chemin entre deux processeurs de la machine ;
- chaque processeur est accessible par l'intermédiaire d'un identifiant unique. L'ensemble des p processeurs est numéroté de 0 à $p-1$.

L'architecture générale de la catégorie *MachineParallèle* est présentée à l'aide du diagramme des classes de la figure 4.2.

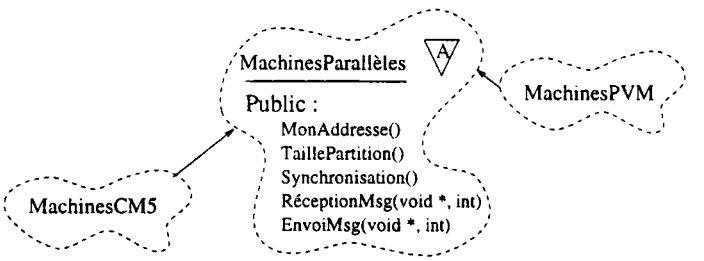


FIG. 4.2 –*Diagramme des classes pour la définition de la machine parallèle*

La classe *MachinesParallèles* est une classe abstraite. En effet, il est impossible d'implanter les différentes méthodes à ce niveau, car il est nécessaire de connaître la machine cible. Son rôle est de fournir une interface unique pour toutes les machines réelles sur lesquelles l'application doit s'exécuter. Ainsi, cette classe définit toutes les méthodes pour utiliser les ressources de la machine et établit les différentes possibilités pour faire communiquer les processeurs. Elle est héritée par les classes *MachinesCm5* et *MachinesPvm* qui permettent une mise en œuvre des services définis au niveau de la classe abstraite pour deux types d'architectures : la CM-5 et un réseau de stations utilisant PVM.

La classe *MachinesParallèles* définit un ensemble de méthodes qui peuvent être appelées par l'utilisateur au niveau de l'application parallèle. Parmi les nombreuses fonctions offertes par cette classe, nous pouvons distinguer :

- la fonction membre *MonAdresse()*: cette fonction retourne l'identifiant unique (*index*) associé à un processeur ;

- la fonction membre *TaillePartition()* : elle retourne le nombre de processeurs disponibles au niveau de la machine parallèle ;

- les fonctions membres *EnvoiMsg()* et *ReceptionMsg()* : l'utilisateur peut utiliser ces fonctions pour envoyer ou recevoir un message en direction d'un processeur quelconque identifié par son index. L'envoi et la réception d'un message peuvent être synchrones ou asynchrones.

Il existe d'autres fonctions que nous ne détaillons pas qui permettent d'obtenir le type d'un message envoyé ou reçu, ou bien de tester si un message attendu est arrivé. Un ensemble particulier de fonctions est très pratique pour déterminer la terminaison de l'application, ce sont les fonctions membres *ProcesseurVrai()*, *ProcesseurFaux()* et *OuGlobal()*. En effet, quand l'application débute, chaque processeur du système distribué appelle la fonction *ProcesseurVrai()* signifiant qu'il est actif. Quand un processeur n'a plus de travail à accomplir, il appelle la fonction *ProcesseurFaux()* pour signaler son état oisif. Pour détecter la terminaison, chaque processeur vérifie la valeur booléenne retournée par la fonction *OuGlobal()*. Quand le résultat retourné est faux, la terminaison de l'application est détectée car la composante de chaque processeur intervenant dans l'évaluation booléenne de la fonction *OuGlobal()* prend la valeur faux.

3.1 Implantation sur un réseau de stations

3.1.1 Présentation de PVM

Pour pouvoir exécuter une application parallèle FTII sur un réseau de stations de travail, nous avons développé la classe *MachinesPVM* à l'aide de la bibliothèque de communications *PVM*[3].

Parallel Virtual Machine est une bibliothèque de communication développée par *ORNL* (*Oak Ridge National Laboratory*) [GBD+ 94]. PVM permet à un réseau de machines hétérogènes de former une machine parallèle virtuelle. Pour obtenir ce résultat, PVM utilise les *sockets de Berkeley* comme moyen de communication. Toute la structure de PVM est basée sur l'envoi et la réception de messages. PVM se compose de deux parties distinctes :

1. La première partie de PVM est une librairie de fonctions accessibles par un programme écrit en C, C++ ou Fortran. Cette librairie contient des fonctions utilisateurs qui permettent d'envoyer ou recevoir des messages, de lancer des processus, de coordonner des tâches, et de modifier dynamiquement la machine virtuelle PVM. Une tâche est au sens PVM, un processus qui s'exécute sur une des stations qui composent la machine parallèle virtuelle.

2. la seconde partie de PVM est un démon qui s'exécute sur chaque station appartenant à la machine parallèle virtuelle. Il est chargé de veiller sur le (ou les ports) de communication de la machine sur laquelle il s'exécute afin d'assurer la réception ou l'envoi des messages demandés par la tâche PVM locale.

Toutes les tâches PVM sont identifiées par un entier (*TID*) qui sert à l'adressage des messages. Ces identifiants ne sont pas consécutifs et le plus petit identifiant n'est pas forcément égal à 0.

3. Ce travail a fait l'objet d'un stage effectué au LRIM par A. Lucchese [Luc 97].

3.1.1.1 Développement de la classe MachinesPVM Le développement de la classe *MachinesPVM* à l'aide de la bibliothèque PVM présente plusieurs difficultés. En effet, la classe *MachinesPVM* hérite de la classe abstraite *MachinesParallèles*, elle doit donc fournir les mêmes services. Le développement des trois services suivants est plus délicat :

1. La numérotation des processeurs : la numérotation proposée par PVM ne correspond pas à la définition proposée. Tous les processeurs ne sont pas accessibles par un numéro compris entre 0 et $p-1$ (où p est le nombre de processeurs).

2. L'envoi de messages synchrones : PVM ne permet pas l'envoi de messages synchrones, seule la réception d'un message peut être bloquante.

3. La gestion de la terminaison de l'application : il n'existe pas au niveau de PVM de fonctions permettant la gestion directe des fonctions membres *ProcesseurVrai()*, *ProcesseurFaux()* et *OuGlobal()*.

Numérotation des processeurs. Avant d'exécuter l'application parallèle sur un réseau de stations, l'utilisateur définit la machine parallèle virtuelle à l'aide d'un fichier qui contient le nom de chacune des machines du réseau qu'il veut utiliser. Quand l'application est lancée par l'utilisateur sur une des stations qui composent la machine parallèle, une phase d'initialisation est nécessaire. L'application se duplique sur chacune des stations (créant ainsi p tâches PVM s'il y a p stations). Elle affecte ensuite à chaque processeur (chaque tâche PVM) un index en commençant à 0 pour elle-même et la dernière tâche créée est le processeur virtuel $p-1$. Après l'initialisation, l'utilisateur à une vue de la machine parallèle conforme à la définition que nous avons donnée.

Envoi de messages synchrones. La bibliothèque PVM permet de réaliser une réception bloquante. En effet, il est possible d'appeler une fonction de réception qui reste bloquée jusqu'au moment où le message a été effectivement lu. Par contre, elle ne fournit pas de fonction pour réaliser un envoi point à point d'une message synchrone. En effet, quand un envoi est réalisé, la fonction se termine dès que le message a été émis sur le réseau. Pour permettre l'envoi de messages synchrones, nous avons ajouté à la fin de chaque message émis un accusé de réception. Si l'accusé de réception est à vrai (émission d'un message synchrone), le processeur qui reçoit le message émet en retour un accusé positif. La fonction d'envoi synchrone attend de façon bloquante la réception de l'accusé avant de rendre la main à l'utilisateur. Cette solution est efficace, cependant elle engendre des messages de contrôles supplémentaires.

Terminaison de l'application. Pour permettre une détection de la terminaison aisée pour l'utilisateur, nous avons développé les fonctions membres *ProcesseurVrai()*, *ProcesseurFaux()* et *OuGlobal()*. Ces fonctions n'existent pas dans la bibliothèque PVM, c'est pourquoi nous avons été amenés à développer une technique pour les simuler.

Comme le nombre de stations composant un réseau local est généralement limité, nous avons choisi une gestion centralisée de ces fonctions. L'implémentation est efficace, ne génère pas un trafic important sur le réseau, et ne risque pas de devenir un goulot d'étranglement. À l'initialisation de l'application, une tâche particulière (appelée *contrôleur*) est créée sur une station de travail du réseau qui peut ou non faire partie de la machine parallèle virtuelle (en fonction du choix de l'utilisateur).

Quand un processeur de la machine parallèle change d'état (passe de l'état vrai à faux par exemple) en appelant *ProcesseurVrai()* puis *ProcesseurFaux()*, il émet un message en direction du contrôleur pour lui signaler ce changement. Quand un processeur désire connaître l'état global du système (en appelant la fonction *OuGlobal()*), il interroge le contrôleur. En effet, le contrôleur connaît l'état de chaque processeur et il peut donc facilement évaluer la fonction *OuGlobal()*.

3.2 Implantation sur une machine parallèle

Pour pouvoir utiliser une application FTII sur une CM-5, nous avons développé la classe *MachinesCM5* à l'aide de la bibliothèque de communications *CMMD* fournit par *Thinking Machine Corporation* [Thi 93a].

3.2.1 Description de la CM-5

Les différentes *Connection Machine* furent construites par *Thinking Machines Corporation* dont le nom même constitue un rappel des ambitions d'intelligence artificielle du fondateur W. D. Hillis [Hil 85].

La Connection Machine CM-5 est une machine parallèle à gros grain de parallélisme constituée de 128 nœuds (Sparc processeurs) dont la vitesse de pointe est de 16 GFops (milliards d'opérations à virgule flottante par seconde) [Thi 93b]. Sa mémoire dynamique de 4 Goctets est répartie sur l'ensemble des 128 processeurs. Ces nœuds peuvent exécuter un code identique distribué dans leurs mémoires spécifiques (SIMD) ou des instructions indépendantes (MIMD). Dans les deux cas, une version du code sera distribuée dans la mémoire locale de chaque nœud. Ces processeurs sont supervisés par un processeur de contrôle fonctionnant sous Unix, qui leur diffuse à intervalles réguliers des blocs d'instructions. Pour améliorer le taux d'efficacité du système, les processeurs sont divisés en partition de 32, 64 ou 128 nœuds utilisés en temps partagé. Enfin, un ensemble de périphériques spécialisées permet la gestion d'entrées-sorties en parallèle.

Les processeurs sont interconnectés par trois réseaux arborescents à haut débit :

le réseau de contrôle utilisé pour les communications concernant tous les nœuds simultanément comme la diffusion des blocks de code et l'échange de synchronisations ;

le réseau de données utilisé pour les communications deux à deux ;

le réseau de diagnostic qui permet de collecter des informations de supervision du système.

Cette architecture offre une grande souplesse de communication et permet d'exécuter avec rapidité des algorithmes réputés coûteux en temps de calcul.

3.2.2 Développement de la classe MachinesCM5

La classe *MachinesCM5*, comme la classe *MachinesPVM* hérite de la classe virtuelle *MachinesParallèles*. Nous avons choisi d'utiliser la bibliothèque CMMD pour la réalisation pratique dans le but d'obtenir une efficacité maximale. De plus, dans la grande majorité des cas, l'écriture de fonctions membres de la classe *MachinesCM5* à l'aide des fonctions fournies par Thinking Machine Corporation est aisée. Par exemple, la fonction

membre *Synchronisation()* appelle simplement la fonction *CMMD_sync_with_nodes()* de la bibliothèque CMMD.

3.3 Extension de la machine parallèle virtuelle

La vue offerte par l'environnement FTII du réseau de communication est un graphe complet. Il semble judicieux de permettre à l'utilisateur de choisir une topologie classique du réseau d'interconnexion pour la machine parallèle (tore, hypercube,...). Si cette information est prise en compte au niveau de l'environnement, elle aura des répercussions sur les parties suivantes de l'environnement :

- la partie MM (modèle matriciel),
- la partie LB (algorithmes d'équilibre de charge dynamique),
- la partie AD (choix de la stratégie d'équilibre de charge la plus adaptée).

En effet, la topologie du réseau d'interconnexion doit être prise en compte au niveau de l'analyse matriciel, car suivant la topologie choisie, le coût d'une communication peut être variable.

De même, la politique d'appariement des stratégies d'équilibre de charge dynamique peut être modifiée pour utiliser au mieux les caractéristiques du réseau de communication (voir paragraphe 3.8, chapitre 2). Par exemple, il est préférable de privilégier les communications locales quand il existe une notion de voisinage.

Enfin, la partie AD, qui comme nous le verrons au paragraphe 6, devra non seulement prendre en compte le caractère irrégulier de l'application et le nombre de processeurs de la machine, mais aussi la topologie de cette machine afin de choisir la stratégies d'équilibre de charge dynamique la plus adaptée au problème.

4 Équilibre de charge dynamique

Nous avons développé les cinq stratégies d'équilibre de charge dynamique présentées au chapitre 3. La partie *LB* regroupe cet ensemble de stratégies sous la forme d'un composant logiciel.

La catégorie *ÉquilibreDeCharge* regroupe les classes qui implantent les différentes stratégies. Pour être indépendants de la machine parallèle cible, les accès directs sont interdits. C'est un objet de type *MachinesParallèles* qui est chargé d'accéder à la machine en fonction des demandes de la stratégie d'équilibre de charge dynamique. Les relations existantes entre ces différentes classes sont exposées à l'aide du diagramme des classes de la figure 4.3.

Chaque tâche est représentée au niveau de la stratégie d'équilibre de charge par un numéro unique. C'est l'application FTII qui réalise le lien entre ce numéro et le travail qui est associé à cette tâche. Grâce à cette représentation, les stratégies d'équilibre de charge sont indépendantes de l'application FTII considérée.

La classe *Charges* est la classe de plus haut niveau. Elle définit les différents services offerts par les stratégies d'équilibre de charge dynamique. De plus, elle réalise un placement statique élémentaire des tâches sur l'ensemble des processeurs. Ainsi, si la machine comporte p processeurs et l'application est divisée en n tâches (numérotées de 0 à $n-1$),

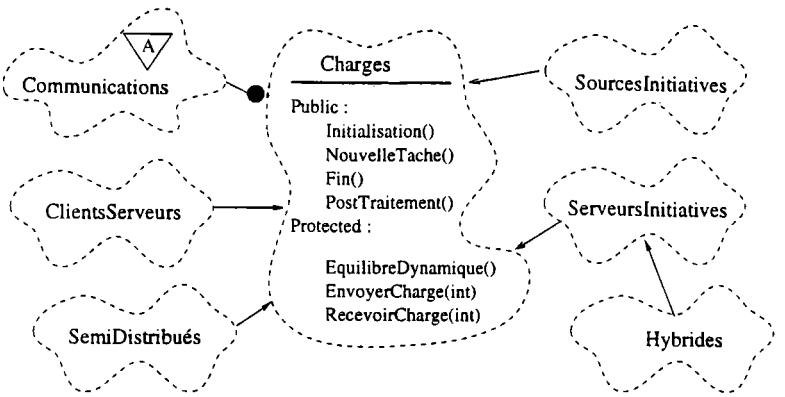


FIG. 4.3 –*Diagramme des classes pour la définition des différentes stratégies d'équilibre de charge*

chaque processeur recevra n/p tâches si n est divisible par p. Dans le cas contraire les $n\%p$ premiers processeurs recevront $n/p + 1$ tâches et les autres auront n/p tâches à calculer.

L'utilisateur n'est pas responsable de la gestion de l'équilibre de charge, il se contente de créer un objet correspondant à la stratégie qu'il veut appliquer. Le programme principal de l'application parallèle FTII peut être représenté par l'algorithme suivant :

Algorithme 4.8 *Programme principal d'une application parallèle FTII*

```
proc Main ()
  MachinesParalleles * machine();
  machine = new MachinesPVM();          /* pour un réseau de stations par exemple */
  machine.Initialisation();
  Charges equi(machine);
  equi.Initialisation();
  tant que ¬equi.Fin() faire                    /* il y a encore du travail... */
          tache = equi.NouvelleTache();
          Calcul(tache);
  fintant que
  equi.Postraitement().
```

Pour chaque stratégie d'équilibre de charge, les fonctions membres des classes associées *Fin()* et *ÉquilibreDynamique()* sont redéfinies. Nous pouvons remarquer que la fonction membre *ÉquilibreDynamique()* est protégée. Ainsi, l'utilisateur n'a pas accès à cette fonction. En fait, c'est la fonction *Fin()* qui se charge de rééquilibrer dynamiquement la charge lorsque cela s'avère nécessaire.

Ainsi, s'il n'y a pas de redistribution dynamique des tâches, (c'est le cas de la classe *Charges*) la fonction *Fin()* peut être implantée ainsi :

Algorithme 4.9 *Détection de la terminaison pour la classe Charges*

```
proc Fin () : Boolean
  si NbTaches == 0 alors
    retourner vrai;
  sinon retourner faux;
  finsi.
```

Dans le cas de la stratégie serveur initiative, la fonction *Fin()* peut être implémentée par l'algorithme 4.10. En effet, si un processeur n'a plus de tâches à calculer, il doit s'assurer qu'il ne peut récupérer du travail au niveau d'un processeur distant avant de quitter l'application. Pour réaliser cette opération, il déclenche une nouvelle phase d'équilibre de charge. S'il a pu trouver de nouvelles tâches, il doit les calculer, dans le cas contraire il est sûr que l'application se termine.

Algorithme 4.10 *Détection de la terminaison pour la classe ServeursInitatives*

```
proc Fin () : Boolean
  si NbTaches == 0 alors
    si EquilibreDynamique() alors          /* des tâches distantes ont été récupéré. */
      retourner faux;
    sinon
      retourner vrai;
    finsi
  sinon retourner faux;
  finsi.
```

Pour la stratégie client-serveur, la fonction *Fin()* doit différencier le processeur P_0 (le serveur) des autres. En effet, P_0 ne participe pas à la résolution du problème, mais il est chargé de distribuer les tâches (voir l'algorithme 4.11).

Algorithme 4.11 *Détection de la terminaison pour la classe ClientsServeurs*

```
proc Fin () : Boolean
  si (MonAdresse() == 0 alors                              /* Je suis le serveur */
    tant que NbTaches > 0 faire                  /* Il y a encore des tâches à calculer */
          RepondsClients()  /* J'envoie une tâche quand un client m'adresse une requête */
          retourner vrai;
    fintant que
    retourner vrai;
  sinon                                                    /* Je suis un client */
    si EquilibreDynamique() alors              /* Je demande une tâche au serveur */
      retourner faux;
    sinon retourner vrai;
    finsi
  finsi.
```

5 Le modèle matriciel

Le modèle matriciel MIMD que nous avons proposé pour valider les différentes stratégies d'équilibre de charge dynamique représente une partie de l'environnement parallèle pour les applications FTII. En effet, c'est un outil d'analyse qui permet de confronter les performances de chaque solution parallèle pour une application FTII particulière. Le modèle matriciel peut être utilisé de deux façons complémentaires :

1. Si l'utilisateur peut modéliser le caractère irrégulier de son application FTII, il peut prédire le comportement de chaque solution parallèle grâce à cette partie de l'environnement.

2. Si le comportement de l'application FTII ne peut être modélisé par une loi de probabilité, il est néanmoins possible d'étudier le comportement de l'application parallèle en utilisant des jeux de données réels obtenus à l'aide de plusieurs exécutions de l'algorithme FTII séquentiel correspondant.

Ces deux utilisations de la partie *MM* sont complémentaires. En effet, si le comportement dynamique de l'application FTII peut être approché par une loi de probabilité classique (normale, binomiale ou log normale par exemple), une étude à l'aide de jeux de données réels permet de renforcer les conclusions émises en utilisant la loi de probabilité. Par contre, si le comportement dynamique de l'application FTII dépend largement des données du problème à traiter, il est parfois difficile d'utiliser une seule loi de probabilité pour analyser la solution parallèle. C'est pourquoi, une étude générale utilisant plusieurs lois classiques a été menée. Il est alors nécessaire d'utiliser des jeux de données réels pour affiner les conclusions générales obtenues à l'aide de l'analyse statistique.

Les résultats de l'analyse menée à l'aide du modèle matriciel sont dépendants de l'application FTII, mais aussi des algorithmes d'équilibre de charge dynamique modélisés. Les conclusions obtenues à l'aide de cette partie ont une influence directe sur l'algorithme de décision qui constitue une des parties de cet environnement.

6 Un algorithme de décision

La partie *AD* (pour *Algorithme de Décision*) aide l'utilisateur à choisir l'algorithme d'équilibre de charge dynamique le plus adapté à la fois à l'algorithme FTII qu'il veut paralléliser et à la machine cible qu'il veut exploiter. La réponse apportée par l'environnement est basée sur les résultats obtenus au chapitre précédent. L'étude des différents algorithmes d'équilibre de charge a conduit aux observations suivantes :

- les stratégies distribuées sont adaptées aux algorithmes FTII, particulièrement si le nombre de processeurs est raisonnable et si le réseau d'interconnexion est performant ;
- quand le nombre de processeurs est relativement important et l'application FTII très irrégulière, l'algorithme client-serveur est très efficace ;
- quand le nombre de processeurs est important, la solution semi-distribuée est la plus adaptée. En effet, elle évite le goulot d'étranglement que représente le serveur et elle ne surcharge pas le réseau de communication.

En nous basant sur l'ensemble de ces conclusions, il est possible de proposer un algorithme de décision général pour choisir une stratégie d'équilibre de charge dynamique en fonction de l'application FTII et de la machine cible :

Algorithme 4.12 *Algorithme de décision*

```
proc Decision(FTII, Machine)/* FTII est l'application considérée et Machine, la machine cible */
  si Regulier(FTII) alors                          /* L'application est peu irrégulière */
    si NbProcMoyen(Machine) alors          /* Le nombre de processeurs est raisonnable */
      Decision ⟵ ServeurInitiative, Hybride      /* L'algorithme proposé est distribué */
    sinon                                        /* Le nombre de processeurs est élevé */
```

```
        Decision ⟵ Semi – distribue          /* L'algorithme proposé est semi-distribué */
    finsi
sinon                                        /* L'application est très irrégulière */
    si NbProcMoyen(Machine) alors            /* Le nombre de processeurs est raisonnable */
        Decision ⟵ Client – serveur          /* L'algorithme proposé est centralisé */
    sinon                                    /* Le nombre de processeurs est élevé */
        Decision ⟵ Semi – distribue          /* L'algorithme proposé est semi-distribué */
    finsi
finsi.
```

7 Conclusion

Après avoir présenté formellement au chapitre précédant la famille d'algorithmes FTII et étudié plusieurs algorithmes d'équilibre de charge dynamique, nous avons proposé dans ce chapitre un environnement logiciel qui permet de paralléliser une application FTII particulière. Nous avons présenté trois exemples d'applications FTII (dont deux issues de la CAO) et nous avons exprimé les limites des applications FTII. L'environnement que nous avons proposé est constitué de cinq parties fondamentales. La partie FTII caractérise l'application FTII à paralléliser, la partie MP représente la machine parallèle cible. Les algorithmes d'équilibre de charge dynamique sont regroupés au niveau d'une même partie qui est indépendante à la fois de la machine parallèle et de l'application FTII. Le modèle matriciel ainsi que l'algorithme de décision sont les deux dernières composantes de cet environnement qui permet à l'utilisateur de choisir une stratégie d'équilibre de charge dynamique en fonction du problème étudié.

Le dernier chapitre de ce mémoire, nous permet d'étudier de façon plus approfondie un exemple d'application FTII. Nous montrons que le lancer de rayons est une application FTII et, à l'aide de l'environnement développé dans ce chapitre, nous proposons une parallélisation efficace de cet algorithme.

Chapitre 5

Analyse expérimentale : le lancer de rayons parallèle

1 Introduction

Ce dernier chapitre est consacré à l'étude d'un exemple d'application FTII. Après une étude bibliographique des principaux algorithmes séquentiels et parallèles de lancer de rayons, nous montrons que l'algorithme de lancer de rayons est une application FTII . De plus, nous exprimons le fait que l'irrégularité du lancer de rayons est très dépendante de la scène à visualiser (c'est à dire des données en entrée).

Nous présentons ensuite l'algorithme de lancer de rayons orienté objets que nous avons parallélisé à l'aide de l'environnement FTII. Pour conclure ce chapitre, les résultats de l'étude expérimentale menée sur une CM-5 sont analysés.

2 Le lancer de rayons parallèle

Le lancer de rayons est une méthode puissante et répandue pour visualiser des images de synthèse très réalistes. Cette technique est appliquée dans un grand nombre de domaines : pour la conception et la fabrication de pièces mécaniques (dans les systèmes de CAO et CFAO), en architecture pour visualiser l'impact d'un nouveau bâtiment dans un paysage existant par exemple.

Bien que de nombreuses optimisations soient apparues, le principal inconvénient de cet algorithme réside dans ses performances. En effet, pour obtenir une image réaliste, une puissance de calcul très importante est requise. De plus, l'indépendance des calculs à réaliser pour obtenir la couleur de chaque point, fait du lancer de rayons un candidat évident pour le parallélisme.

Nous présentons dans cette partie, le principe de cet algorithme ainsi que ses principales optimisations. Les solutions parallèles existantes sont étudiées afin de comparer les mérites de chacune d'entre elles.

2.1 Principe

L'algorithme du lancer de rayons est fondé sur un principe très simple. Il est basé sur la remarque suivante : l'œil ne perçoit des objets que les rayons lumineux transmis par

réflexion ou réfraction. L'idée est donc de retrouver la trajectoire de ces rayons pour déterminer l'éclairement des objets vus. Calculer l'éclairement d'un point nécessite d'étudier plusieurs types de rayons (voir figure 5.1) :

les rayons primaires déterminent les parties visibles de la scène ;

les rayons d'ombrage détectent les ombres portées ;

les rayons secondaires permettent d'obtenir l'éclairement indirect, c'est à dire par réflexion et transparence.

Toutes les informations recueillies par le suivi des rayons sont utilisées dans un modèle d'éclairement pour déterminer réellement la couleur d'un pixel de l'écran. Un algorithme, qui peut s'appliquer aux rayons primaires, est le suivant :

Algorithme 5.1 *lancer de rayons*

pour chaque point de l'écran **faire**
Calculer le rayon qui passe par le centre du point et par l'observateur
Calculer l'ensemble des intersections avec les objets
Classer les intersections obtenues
La couleur du point est la couleur de l'objet dont
l'intersection est la plus proche
finpour

Si l'algorithme de lancer de rayons donne des images réalistes, il est malheureusement très coûteux en temps. C'est pourquoi, il n'a connu un réel développement que depuis les années 1980 (T. Whitted présente plusieurs images avec des effets très réalistes [Whi 80]) alors qu'il fut proposé par A. Appel en 1968 [App 68].

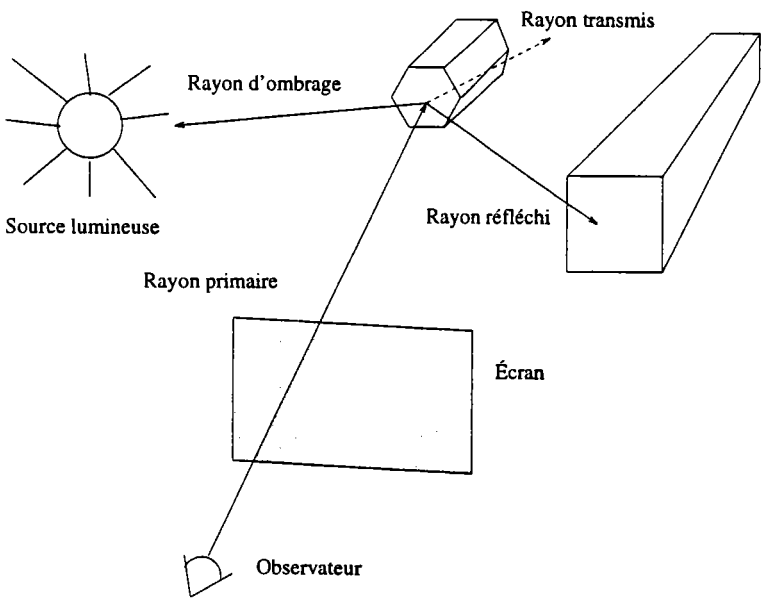


FIG. 5.1 –*Principe de l'algorithme de lancer de rayons.*

2.2 Les optimisations séquentielles

On constate que l'essentiel du temps d'exécution de l'algorithme est consacré à calculer des intersections rayon-objets. C'est pourquoi des techniques d'optimisation ont été développées, parmi lesquelles on distingue trois classes principales :

- la réduction du nombre d'intersections calculées ;
- la réduction du nombre de rayons lancés ;
- la réduction du coût de calcul d'une intersection rayon-objet.

Une classification plus précise des optimisations du lancer de rayons peut être trouvée dans [Viv 93, Gla 89].

2.2.1 Réduction du nombre d'intersections calculées

2.2.1.1 Les englobants. Pour éviter de calculer des intersections inutiles, on utilise la notion d'englobant. Un englobant est une forme géométrique simple qui contient complètement un objet. Avant de calculer l'intersection d'un rayon avec un objet donné, on teste si le rayon traverse l'englobant de l'objet, si oui, alors l'intersection avec l'objet est effectivement calculée, sinon il est inutile de réaliser ce calcul.

L'utilisation de sphères comme englobants peut être intéressante. En effet, l'intersection avec une sphère est très facile à calculer. De plus, elle reste simple même pour les rayons secondaires qui sont de directions quelconques. Mais les sphères ont pour inconvénient d'être un englobant assez grossier (l'erreur d'approximation du volume de l'objet est importante). S. D. Roth utilise des volumes parallélépipédiques rectangles pour englober les objets [Rot 82]. Les boîtes utilisées ont pour particularité d'avoir les faces parallèles aux différents axes et approchent mieux les objets que les sphères. Cependant le calcul d'intersection est plus long.

2.2.1.2 Les divisions spatiales. Pour éviter de calculer des intersections inutiles, une autre méthode consiste à subdiviser l'espace en régions. Dans ce cas, pour un rayon donné, les intersections avec des objets ne se trouvant pas dans les régions traversées par le rayon ne sont pas calculées.

Les divisions uniformes. L'espace est divisé en régions (*voxel*) de façon régulière. Cette méthode est utilisée par Amanatides *et al.* ainsi que par Fujimoto *et al.* [AW 87, FTI 86]. L'inconvénient principal est qu'ici la structure de la scène n'est pas prise en compte. Ainsi chaque rayon va traverser un certain nombre de régions vides. Par contre, pour déterminer la cellule voisine sur le parcours du rayon, un algorithme incrémental très simple est utilisé.

Les divisions non uniformes. L'espace est subdivisé en tenant compte de la cohérence de la scène. A. S. Glassner découpe l'espace à l'aide de la méthode des *octrees* [Gla 84]. Cette méthode est l'extension de *quadtree* à la troisième dimension. L'espace est divisé récursivement en huit parties jusqu'à l'obtention de parties vides, ou de tailles minimales. Quand un rayon traverse une boîte, les intersections avec les objets

sont calculées. Si une intersection est trouvée et qu'elle appartient à cette boîte alors cette intersection est le point vu par l'observateur, sinon le rayon passe dans la boîte suivante. Le problème le plus difficile à résoudre est de déterminer le voisin d'une boîte en un temps minimum. Bouatouch *et al.* utilisent la notion de grille pour trouver le voisin d'une zone [BMPA 87]. Une autre méthode utilisée par Bouatouch *et al.* consiste à projeter les englobants sur l'écran. La connexité de l'ensemble des boîtes est réalisée avec des pointeurs.

Un reproche souvent formulé à l'encontre des méthodes utilisant les subdivisions spatiales est que la même intersection entre un rayon et un objet donné est calculée plusieurs fois. En effet, si un objet est partagé par plusieurs boîtes et qu'un rayon traverse ces régions, l'intersection avec cet objet sera calculé à chaque fois que le rayon change de boîte. Pour éviter cet inconvénient, J. Amanatides et A. Woo utilisent la notion de *boîte aux lettres* [AW 87] : chaque rayon est numéroté de façon unique. A chaque calcul d'intersection entre un objet et un rayon, la boîte aux lettres est consultée afin de vérifier si ce calcul a déjà été effectué. Si ce n'est pas le cas, le calcul est réalisé et le résultat mémorisé dans la boîte aux lettres.

2.2.2 Réduction du nombre de rayons lancés

2.2.2.1 Sous échantillonnage. Pour diminuer le nombre de rayons à étudier, une solution consiste à ne lancer qu'un rayon pour un voisinage de points (*sous échantillonnage*). Ainsi, la couleur du point pour lequel aucun rayon n'a été lancé est la moyenne des couleurs des points voisins. Pour éviter un trop fort aliassage, des rayons supplémentaires sont lancés pour les zones à fort contraste (*sur échantillonnage*).

2.2.2.2 Projection des objets. Afin de réduire le nombre de rayons primaires lancés, une optimisation simple peut être appliquée. Chaque objet de la scène à visualiser est projeté à l'écran. Il est alors inutile de considérer un rayon primaire associé à un point de l'écran, si aucun des objets ne s'est projeté sur ce pixel. En effet, ce rayon ne rencontrera aucun objet sur son parcours, et n'engendrera donc aucun rayon d'ombrage, ni secondaire. La projection des objets à l'écran étant coûteuse, il est possible de projeter uniquement les englobants parallélépipédiques afin d'accélérer le prétraitement (voir paragraphe 2.2.1.1). Nous avons proposé un algorithme parallèle utilisant cette optimisation afin d'atteindre un double objectif : réduire le nombre de rayons primaires et estimer la charge liée aux rayons primaires du système [KHH 95, Kra 95].

2.2.2.3 Filtre des faces arrières. Si un objet est approché par un ensemble de faces polygonales, un filtre simple permet d'éliminer un nombre important des faces non visibles par l'observateur. Nous supposons que la normale de chaque face composant le polyèdre est orientée vers l'extérieur. Si la normale d'une face s'éloigne de l'observateur, nous pouvons déduire que cette face n'est pas vue par l'observateur, car elle est cachée par d'autres faces de l'objet plus en avant. Cette technique est appelée *élimination des faces arrières* [FvDF$^+$ 95]. Pratiquement, une face arrière est détectée par un produit scalaire positif ou nul (entre la normale de la face et le vecteur formé par l'observateur et un point de la face).

Cette technique peut être appliquée au lancer de rayons, pour déterminer s'il est utile d'étudier un rayon d'ombrage. En effet, pour savoir si une face (dont un point est P_f et

la normale est N_f) est éclairée par une source lumineuse placée en P_l, l'étude du produit scalaire $\overrightarrow{N_f}.\overrightarrow{P_lP_f}$ fournit un début de réponse. Si le produit est positif ou nul, la face n'est pas éclairée par la source lumineuse considérée. Dans le cas contraire, il est nécessaire d'étudier le parcours de ce rayon.

2.2.3 Réduction du temps de calcul d'une intersection rayon-objet

Une dernière approche pour accélérer le rendu d'une scène est d'optimiser le temps de calcul d'une intersection rayon-objet. Plusieurs méthodes sont applicables. En premier lieu, il est possible de proposer des algorithmes de calcul d'intersection très spécifiques et performants pour chaque type de primitive reconnue par le modèle [Gla 89]. Par exemple, nous pouvons remarquer qu'il est impossible de voir plus de trois faces d'un cube à la fois. Il est donc possible de proposer une procédure d'intersection qui détermine les trois faces visibles, puis teste uniquement les trois faces déterminées [PAGM 88].

Y. Lanuel propose le modèle *VSG* (*Visual Solid Geometry*) adapté à la visualisation qui est une extension du modèle *CSG* (*Constructive Solid Geometric*) [Lan 94, Req 80]. Le modèle CSG est un modèle géométrique qui peut être décrit comme un arbre binaire où les nœuds sont soit des opérations booléennes régularisées (union, intersection, différence), soit des transformations géométriques (rotation, translation, homothétie, etc..). Les feuilles de l'arbre contiennent soit des volumes primitifs, soit les paramètres des transformations géométriques. La définition du modèle CSG est récursive ; par conséquent un nœud de l'arbre CSG est lui-même un arbre CSG. Le calcul d'intersection entre un rayon et un objet CSG peut être parfois long. En effet, un même objet CSG peut être représenté par différents arbres CSG dont le parcours est plus ou moins coûteux. De plus, les englobants déduit d'un arbre CSG, sont parfois grossier. C'est pourquoi, Y Lanuel transforme les objets CSG en objets VSG. Le modèle VSG permet une représentation unique des objets et l'englobant associé à chaque objet est plus précis. Ainsi, on réalise un double objectif : réduire le temps de calcul d'une intersection rayon-objet et réduire le nombre de calculs d'intersection rayon-objet en utilisant des englobants plus précis.

Pour conclure, on constate que malgré les nombreuses techniques d'optimisations développées à ce jour, le temps de calcul séquentiel d'une image reste important. C'est pourquoi, de nombreuses optimisations parallèles de l'algorithme de lancer de rayons ont été développées [GP 89, KH 95, LRN 95, SGS 95, KKHN 93, KNK+ 88].

2.3 Le lancer de rayons en parallèle

Une classification simple qui peut être appliquée aux algorithmes de lancer de rayons en parallèle est de distinguer le type des communications réalisées par ces algorithmes. En général, la scène est distribuée sur l'ensemble des processeurs. Pour traiter un rayon, le processeur peut obtenir les objets qui lui manquent en envoyant des requêtes aux processeurs distants: on parle dans ce cas de *flot d'objets*. Il peut également déléguer le calcul du rayon à des processeurs distants. On parle dans ce cas de *flot de rayons*. On peut également distinguer les algorithmes par les techniques employées pour résoudre les problèmes d'équilibre de charge.

2.3.1 Flot de rayons

Généralement, les algorithmes orientés flot de rayons réalisent une subdivision spatiale qui permet de répartir l'ensemble de la scène dans les mémoires locales propres à chaque processeur. Chaque processeur possède en mémoire au moins une région et les objets qui y sont contenus. Il calcule les intersections de ces objets avec un rayon donné, si aucune intersection n'est trouvée, le rayon est transmis au processeur qui gère la région voisine sur le parcours du rayon. L'inconvénient majeur de cette méthode est de réitérer plusieurs fois le même calcul d'intersection entre un rayon et un objet considérés. En effet, lorsqu'un objet est partagé par plusieurs cellules sur le parcours d'un rayon, l'intersection entre le rayon et cet objet est recalculé dans chaque cellule. Pour résoudre ce problème, une solution efficace est l'utilisation d'une *boîte aux lettres* [AW 87]. Cependant, une implémentation de cette dernière nécessite le principe d'une mémoire centralisée. La mémoire étant distribuée, ce principe engendre un surcoût en communication pour mettre à jour la boîte aux lettres dans les mémoires des différents processeurs [Pri 89]. L'efficacité de cette catégorie d'algorithme est fonction du choix de la subdivision qui doit limiter au maximum les échanges de rayons entre les différents processeurs afin d'éviter de surcharger le réseau de communications.

Aykanat *et al.* divisent l'espace à l'aide de plans [AIÖ 94]. Ils définissent un coût a priori de la division à l'aide d'une heuristique. Le coût d'une division est fonction du nombre d'objets partagés par les deux zones créées et de la régularité de la division (afin d'obtenir deux zones de tailles identiques). Ils minimisent les partages d'objets pour éviter leur duplication dans différentes mémoires locales. En effet, le partage des objets implique une redondance de calculs car la notion de boîte aux lettres n'est pas appliquée. Ainsi, les auteurs réalisent un double objectif : ils répartissent les objets sur l'ensemble des mémoires distribuées à l'aide de la subdivision spatiale et essaient d'uniformiser la tâche de chacun des processeurs à l'aide de l'heuristique. Les communications entre les processeurs consistent en des échanges de rayons. L'équilibre de charge n'est pas dynamique, mais est réalisé de façon statique en même temps que la subdivision spatiale (c'est la fonction heuristique qui estime la charge d'une région).

T. Priol et K. Bouatouch procèdent différemment [PB 89]. Dans un premier temps, les auteurs construisent une subdivision spatiale uniforme. Les auteurs expriment la propriété suivante : si un point traverse un objet, il est très probable que le point voisin rencontre le même objet, c'est ce qu'ils appellent *la cohérence des rayons*. Ils choisissent donc un échantillonnage de rayons pour chacune des boîtes et calculent la couleur de chacun de ces rayons. Ils obtiennent ainsi une bonne estimation de la charge de travail attachée à une boîte. De cette façon, ils peuvent répartir de manière équitable la charge sur l'ensemble des processeurs en attribuant à chacun d'entre eux un nombre variable de boîtes. Les objets sont répartis dans les mémoires locales, un processeur conserve les objets qui sont dans au moins une des boîtes qu'il gère. Cette méthode réalise un équilibre de charge statique qui résulte du sous échantillonnage effectué pour répartir les boîtes entre chaque processeur.

Kobayashi *et al.* proposent un algorithme orienté flot de rayons qui résout le problème de l'équilibre de charge de façon statique [KKHN 93]. Les auteurs utilisent une grille régulière pour réduire le nombre d'intersections rayon-objet calculées. Le lancer de rayons est utilisé pour visualiser une animation. L'écran est divisé en un nombre fixe de régions qui sont distribuées sur l'ensemble des processeurs. La méthode d'équilibre de charge utilise

la propriété de cohérence qui existe entre deux images d'une animation pour répartir la charge. À la fin du calcul d'une image, le coût de chaque région est estimé. Ensuite, un processeur calcule la charge moyenne que doit avoir chaque processeur. Une nouvelle répartition de la charge est calculée pour la visualisation de la prochaine image.

2.3.2 Flot d'objets

Les communications réalisées ne concernent plus les rayons mais les objets. Les processeurs sont responsables du calcul d'un certain nombre de rayons dans leur intégralité et demandent aux processeurs distants les objets qu'ils ne possèdent pas en mémoire locale.

Dans le cas d'un modèle SIMD avec mémoire centralisée, le temps de calcul d'une image est le temps de calcul du rayon le plus coûteux (à condition de disposer d'un processeur par pixel). Chaque processeur peut accéder à tous les objets de la scène. Cet algorithme ne résout pas le problème de la répartition de la charge, qui est un problème difficile à résoudre en mode SIMD [Fon 94].

Une méthode utilisant le flot d'objets est proposée par Badouel *et al.* [BBLP 89, BBPA 94]. Les auteurs choisissent ici d'implanter une mémoire virtuelle partagée. Ainsi les algorithmes écrits sont plus élégants et plus faciles à mettre en œuvre. Pour simuler une mémoire virtuelle, il faut disposer de plusieurs outils (voir paragraphe 2.2.1.2 du chapitre 1). Dans un premier temps, il est nécessaire de mettre à la disposition de chacun des différents processeurs un système de mémoire cache. La mémoire de chaque processeur doit être divisée en trois parties. La première partie est la mémoire locale où les objets propres au processeur sont conservés. La seconde partie de la mémoire contient le programme à exécuter. La troisième partie de la mémoire est allouée au système de cache. Le cache doit pouvoir faire le lien entre mémoire virtuelle et mémoire physique. La mémoire est paginée et chaque processeur possède un certain nombre de pages. Le cache est chargé de fournir à chaque processeur les pages qu'il demande. Si cette page est déjà dans la mémoire du cache, alors il suffit de la fournir au processeur. Par contre, si cette page n'est pas présente, alors le cache se charge de demander au processeur qui en est le propriétaire, de lui fournir cette information. Une politique de remplacement doit être appliquée dans le cas où un cache est plein. La stratégie choisie par les auteurs est d'éliminer la page qui a été la moins récemment utilisée.

Cette méthode engendre un nombre important de communications au début de l'exécution. Par la suite, le nombre de communications diminue et se stabilise. En effet, chaque processeur est responsable d'un groupe de rayons très proches. Et donc la propriété de cohérence des rayons a pour effet de stabiliser le nombre d'objets nécessaires à chaque processeur. Une fois cet ensemble chargé, le nombre de messages échangés diminue.

Une méthode de répartition dynamique de la charge envisagée par les auteurs est de confier un pixel à chaque processeur. Quand un processeur a terminé de calculer ce point, il en demande un autre. Mais cette méthode perd les avantages de la cohérence des rayons et donc le nombre de communications augmente de façon non négligeable. Pour exploiter la cohérence des rayons, la solution proposée est de diviser l'écran en petites zones de taille fixe. Chaque processeur demande une zone et quand il a terminé de la traiter, il en demande une nouvelle jusqu'à ce que l'image entière soit calculée.

S. A. Green et D. J. Paddon proposent un algorithme orienté flot d'objets [GP 89]. Afin d'accélérer le calcul d'une image, une subdivision non régulière est appliquée (les auteurs utilisent l'algorithme de A. Glassner [Gla 84]). La charge est répartie par un algorithme de

type client-serveur. L'algorithme utilise une méthode similaire à l'algorithme de Badouel *et al.* En effet, ils utilisent un principe proche de la mémoire virtuelle partagée pour répartir la scène entre les processeurs. Les auteurs présentent une méthode permettant de construire un ensemble d'objets que tous les processeurs doivent connaître. Les autres objets sont connus uniquement du serveur qui les distribue quand il reçoit une demande d'un de ses clients. Les résultats présentés sont très proches des résultats obtenus par Badouel *et al.*

Un dernier exemple de l'utilisation du principe de mémoire virtuelle partagée pour le lancer de rayons est la méthode proposée par M. J. Keates et R. J. Hubbold [KH 95]. Dans cet algorithme, la MVP est gérée de façon matérielle par la machine (une *KSR-1* [FIR 93]) et non pas de façon logicielle comme pour les algorithmes décrits dans [BBPA 94, GP 89]. Les auteurs utilisent une grille régulière pour déterminer le parcours d'un rayon. L'écran est divisé en un nombre fixe de régions. Chaque région est composée de plusieurs tâches. La stratégie d'équilibre de charge mise en œuvre est de type source initiative (voir paragraphe 3.3.1 du chapitre 2). Chaque processeur doit calculer une région. Quand il termine le calcul de sa région initiale, il aide un processeur qui n'a pas encore terminé son travail.

Dans [KHHG 96b], nous avons montré formellement qu'un algorithme orienté flot d'objets et plus efficace qu'un algorithme flot de rayons. Intuitivement on peut justifier ce résultat, par le fait que dans un algorithme orienté flot d'objets, le calcul d'intersection d'un rayon avec un objet ne sera réalisé qu'une seule fois. Ce qui n'est pas le cas pour un algorithme orienté flot de rayons où cette intersection sera calculée plusieurs fois. Pour éviter ces calculs, il faudrait disposer d'une boîte aux lettres, ce qui est difficilement réalisable quand la mémoire est distribuée. L'algorithme proposé réalise un équilibre de charge statique pour le calcul des rayons primaires. Une subdivision adaptative de l'écran basée sur un *quadtree* est appliquée. Un coût est associé à chaque région de l'écran en fonction du nombre d'objets qui s'y projettent et de sa surface. Comme il est difficile de prévoir la charge engendrée par les rayons secondaires, nous avons choisi de développer deux algorithmes dynamiques d'équilibre de charge : le premier est centralisé et le second distribué. Nous avons montré qu'ils étaient de complexité comparable : le choix d'une des deux méthodes dépend alors de la machine cible sous-jacente. Enfin nous avons implémenté ces algorithmes sur la CM-5 (*Connection Machine*) du Centre National de Calcul Parallèle en Sciences de la Terre. Les résultats pratiques obtenus sont comparables aux travaux récents rencontrés dans la littérature.

2.3.3 Les algorithmes mixtes

Certains algorithmes réalisent parfois des échanges concernant à la fois des rayons et des objets. Généralement les communications portent sur des rayons quand il n'y a pas de problème de charge. Quand, il est nécessaire de redistribuer la charge, des communications d'objets apparaissent.

M. Dippé propose dès 1984 une répartition dynamique de la charge [Dip 84]. Il suggère de réaliser une subdivision spatiale de l'espace. A l'aide de cette subdivision spatiale, il répartit les objets dans les mémoires des processeurs. Il choisit de subdiviser l'espace en régions limitées par six faces en forme de losange (*rhomboèdre non régulier*). Un moniteur est chargé de superviser l'ensemble des processeurs. Pour répartir la charge, il peut déplacer un sommet d'une région de façon à agrandir ou, au contraire à réduire, la taille de cette région. Le moniteur choisit de diminuer la taille d'une région si sa charge est trop

importante. La région voisine est donc agrandie et se retrouve ainsi avec une charge plus importante. De plus, lors du déplacement d'un sommet, un certain nombre d'objets sont échangés entre les zones dont la taille a été modifiée. Mais cet algorithme pose des problèmes. Comment choisir quel sommet doit être déplacé? Que se passe-t-il si deux zones veulent déplacer le même sommet au même instant? Ces difficultés expliquent pourquoi cet algorithme n'a pas été implanté. K. Nemoto et T. Omachi simplifient l'algorithme de M. Dippé [NO 86]. Les cellules associées aux processeurs sont des parallélépipèdes orthogonaux. L'équilibre de charge est réalisé plus simplement en faisant glisser les surfaces des cellules chargées. Ce glissement est réalisé le long d'un axe principal prédéfini. Les indicateurs de contrôle de charge sont définis explicitement par les auteurs. Cet algorithme reste trop compliqué et n'a pas été mis en œuvre.

W. Lefer propose un algorithme avec équilibre de charge dynamique [Lef 93]. Il ne se préoccupe pas de la méthode d'accélération choisie. (méthode des englobants, subdivision spatiale). Une architecture client-serveur est mise en place. L'algorithme comporte deux tâches distinctes qui peuvent être exécutées par deux processeurs. La première tâche se charge de calculer l'ensemble des objets qui ont une intersection avec un rayon donné. La seconde tâche se charge de calculer les intersections avec les objets renvoyés par la première tâche. Si un processeur est surchargé, il envoie sa file d'attente au serveur. Le serveur la mémorise et attend qu'un processeur lui réclame du travail. A ce moment, il lui transmet la file du processeur chargé ainsi que les informations nécessaires pour qu'il puisse effectuer le travail. Cette méthode résout donc de façon dynamique le problème de l'équilibre de charge. Le système de résolution est centralisé : il en résulte un nombre important de communications en cas de surcharge, puisque le serveur est un intermédiaire entre processeurs surchargés et processeurs oisifs.

S. Whitman propose une méthode proche de la précédente sur un algorithme d'élimination de parties cachées [Whi 94]. L'auteur constate de façon empirique, qu'une subdivision régulière est la meilleure solution pour la méthode d'équilibre de charge statique. D'après l'auteur, en tenant compte des données, 25 % de temps en plus est nécessaire pour le prétraitement de la méthode statique lorsque la subdivision n'est pas régulière. La charge globale du système est partagé en un ensemble de tâches élémentaires. Un processus demande une tache et l'exécute. Si un processeur est surchargé, il peut être utile de couper la file d'attente des messages de ce processeur en deux parties. Un autre processeur est alors chargé de traiter une partie de cette file d'attente afin de permettre un retour à un équilibre de charge.

2.4 Conclusion

Dans cette section, nous avons exposé les différentes difficultés inhérentes au lancer de rayons. Ainsi, la puissance de calcul nécessaire pour obtenir des images réalistes est importante. Bien que la complexité théorique de cet algorithme a été réduite de façon sensible grâce à différents types d'optimisation (réduction du nombre de rayons, subdivision spatiale,...), les temps de calcul restent conséquents.

Les solutions parallèles proposées dans la littérature améliorent sensiblement les résultats obtenus. Il existe deux grandes techniques de parallélisation de cet algorithme : les algorithmes orientés flots de rayons et flots d'objets. Dans un cas (flots de rayons), les communications sont très nombreuses mais leur taille est réduite, et dans le second cas (flots d'objets), les communications sont moins nombreuses mais de taille plus importante.

Nous pouvons remarquer que la plupart des solutions existantes se préoccupent peu de l'équilibre de charge. La solution la plus souvent utilisée est de centraliser la gestion de la charge dans une architecture client-serveur.

Après avoir présenté un bref état de l'art de la parallélisation du lancer de rayons, nous montrons que le lancer de rayons est une application FTII. Ainsi, nous pouvons paralléliser cet algorithme en appliquant la méthode générale que nous avons proposée pour ce type d'algorithme. La solution parallèle que nous proposons est donc indépendante non seulement de la machine parallèle cible mais aussi de la stratégie d'équilibre de charge choisie par l'utilisateur.

3 Étude expérimentale

Le lancer de rayons est une application FTII. En effet, il est possible de décomposer le travail à accomplir en un ensemble fini de tâches élémentaires et indépendantes. Pour cela, nous divisons l'écran en un nombre fini de rectangles. Chaque rectangle (identifié par un entier) est composé de plusieurs points écran (*pixels*). L'ensemble des tâches est dans ce cas constitué par les rectangles et calculer une tâche revient à déterminer la couleur de chaque point qui constitue le rectangle considéré. Cependant, le temps de calcul de chaque tâche est imprévisible.

Notre étude est originale, dans le sens où nous proposons d'étudier le lancer de rayons parallèle en fonction de la stratégie d'équilibre de charge dynamique appliquée. En général, les auteurs qui étudient le lancer de rayons parallèle, ne concentrent pas leurs efforts sur ce point particulier, c'est pourquoi dans la majorité des cas, une répartition statique est réalisée ou bien la charge est répartie par une stratégie centralisée.

Dans les chapitres précédents, nous avons proposé différentes stratégies d'équilibre de charge dynamique pour les applications FTII. Pour chaque stratégie, nous avons estimé le surcoût induit par la gestion dynamique de l'équilibre de charge. Un modèle matriciel adapté à l'ensemble des hypothèses que nous avons exprimées a été développé afin d'évaluer théoriquement les performances respectives de ces différentes stratégies.

Afin de valider cette étude théorique, nous présentons l'algorithme de lancer de rayons que nous avons développé. Cet algorithme (utilisant la programmation orientée objets) met en œuvre chacune des stratégies présentées. Grâce à cette implantation d'une application FTII, nous pouvons d'une part, comparer expérimentalement les différentes stratégies d'équilibre de charge proposées et d'autre part, valider l'environnement proposé pour paralléliser les applications FTII.

L'algorithme parallèle est composé de deux parties principales. La première partie est chargée du contrôle du parallélisme pour les applications FTII. Elle est responsable de la coordination des processeurs, de l'initialisation des différentes structures sur l'ensemble des processeurs ainsi que de la répartition de la charge (initiale et dynamique). Nous avons présenté en détail cet environnement parallèle au chapitre 4. La seconde partie est consacrée à l'algorithme de lancer de rayons lui même et à ses optimisations.

3.1 Irrégularité du lancer de rayons

Le but de cette étude est d'expliquer pourquoi le lancer de rayons est une application irrégulière. Après avoir déterminé la complexité théorique de cet algorithme, nous met-

tons en évidence les différents critères qui sont des sources de travail et qui modifient la répartition de la charge.

Eric Haines a proposé un ensemble composé de six scènes afin de tester les qualités d'un algorithme de lancer de rayons [Hai 87]. À l'aide de ces scènes, il est possible de tester l'efficacité de l'algorithme en considérant le temps d'exécution mais aussi le nombre de rayons générés, le nombre de tests d'intersection rayon-englobant, rayon-objet... Cependant, cette étude n'est pas satisfaisante dans le cadre de l'algorithme parallèle de lancer de rayons. En effet, il n'est pas possible de prendre en compte certaines notions comme, par exemple, la répartition spatiale des objets. C'est pourquoi, nous nous proposons d'étudier les différents paramètres qui sont susceptibles de modifier le comportement de l'algorithme parallèle.

Dans le but de montrer les liens existants entre l'analyse matricielle que nous avons réalisée et cette étude, nous proposons une méthode pour créer des scènes à l'aide de lois de probabilité. Les scènes ainsi créées possèdent des caractéristiques très particulières qui permettent d'isoler chacun des paramètres influant sur la charge de l'application (répartition spatiale et caractéristiques des objets).

Enfin, nous concluons en proposant une méthode générale pour décider du choix de l'algorithme d'équilibre de charge dynamique dans le cadre du lancer de rayons parallèle.

3.1.1 Complexité théorique de l'algorithme de lancer de rayons

Il est possible d'estimer le nombre maximal de rayons générés par le lancer de rayons en fonction de la taille de l'image, du nombre de sources lumineuses et de la profondeur de l'arbre de rayons. Nous proposons d'utiliser les notations suivantes :

- T : la taille de l'image (en points écran) ;
- S : le nombre de sources lumineuses ;
- p : la profondeur de l'arbre de rayons ;
- R : le nombre de rayons primaires ;
- R_o : le nombre de rayons d'ombrages ;
- R_s : le nombre de rayons secondaires.

On suppose que le calcul des rayons primaires est le niveau 0 de l'arbre de rayons. Le nombre maximum de rayons générés N est alors :

$$
\begin{aligned}
N &= R + R_o + R_s \\
N &= R \times \left(1 + S \times \sum_{i=0}^{p} 2^i + \sum_{i=1}^{p} 2^i\right) \\
N &= R \times \left(1 + S(2^{p+1} - 1) + 2^{p+1} - 2\right) \qquad (5.1) \\
N &= R(1 + S)(2^{p+1} - 1) \text{ avec } R = T \qquad (5.2)
\end{aligned}
$$

Il est à noter que cette valeur représente la complexité théorique dans le pire des cas. Généralement le nombre réel de rayons calculés est très inférieur à cette borne. De plus, cette valeur ne tient pas compte de la nature de chaque rayon alors que le coût moyen d'un rayon secondaire (réfléchi ou transmis) est sensiblement supérieur au coût d'un rayon d'ombrage.

3.1.2 Les sources lumineuses

Les sources lumineuses ont un impact direct sur le comportement de l'algorithme de lancer de rayons. Le nombre de sources lumineuses ainsi que leur position influent sur la charge globale du système distribué et sur la répartition des calculs entre les différents processeurs.

3.1.2.1 Le nombre de sources lumineuses. Les rayons d'ombrage représentent une part importante de la charge globale du système. À l'aide de l'équation 5.1, il est aisé de déterminer la répartition des rayons par type (primaire, ombrage, secondaire) :

$$\text{Rayons primaires} : \frac{1}{(1+S)(2^{p+1}-1)}$$

$$\text{Rayons d'ombrage} : \frac{S}{S+1}$$

$$\text{Rayons secondaires} : \frac{2^{p+1}-2}{(1+S)(2^{p+1}-1)}$$

Par exemple, si la profondeur de l'arbre de rayons est 4 et le nombre de sources lumineuses égal à 3, les rayons primaires représentent 0,81 % des rayons engendrés, les rayons secondaires 24,19 % et les rayons d'ombrage 75 %.

Il est utile de rappeler que cette répartition est calculée à l'aide de la complexité théorique. En réalité, la répartition peut être différente. Il est facile d'éviter de lancer un nombre non négligeable de rayons d'ombrage en utilisant des filtres simples, comme par exemple le filtre des faces arrières (voir paragraphe 2.2.2.3). On constate en particulier que l'importance des rayons primaires est généralement diminuée au profit des rayons d'ombrage et secondaires. En effet, le calcul théorique suppose que chaque rayon primaire va engendrer un arbre de rayons complet ce qui est rarement le cas. Comme l'arbre de rayons associé à un point écran n'est pas toujours complet, la proportion des rayons primaires est supérieure à celle calculée précédemment. En particulier, si le nombre de rayons primaires effectifs (*i. e.* qui rencontrent réellement un objet) est faible, l'importance des rayons d'ombrage est largement surestimée et, dans une moindre mesure, c'est aussi le cas des rayons secondaires. De même, si les objets sont opaques, il n'y aura aucun rayon transmis ce qui réduit de moitié le nombre de rayons secondaires et d'ombrage. Enfin, pour le calcul des rayons d'ombrage, il est inutile de déterminer l'objet le plus proche sur le parcours du rayon (comme c'est le cas pour les rayons primaires et secondaires), mais il faut simplement déterminer si un objet est sur le parcours de ce rayon. Ainsi le coût d'un rayon d'ombrage est plus faible que le coût d'un rayon primaire ou secondaire.

Pour exemple, nous comparons la répartition théorique et réelle des rayons pour deux des scènes proposées par E. Haines. Les résultats obtenus pour les scènes *Sphereflake* et *Mount* sont résumés dans le tableau 5.1. La taille de la base de données choisie pour créer les scènes est 2. La résolution des images est 200 × 200 pixels et la profondeur maximale de l'arbre de rayons est 3. La scène *Sphereflake* est éclairée par quatre sources lumineuses, la scène *Mount* par une seule.

Pour la scène *Sphereflake*, il n'y a pas de rayons transmis calculés car tous les objets sont opaques, c'est pourquoi l'importance des rayons secondaires et d'ombrage est surestimée au détriment des rayons primaires. Par contre, pour la scène *Mount*, le nombre de rayons effectifs est largement inférieur (environ 40 %) au nombre de rayons primaires calculés, ce qui explique les résultats obtenus.

TAB. 5.1 – *Comparaison des répartitions réelles et théoriques.*

	Sphereflake			*Mount*		
Rayons	nombre	rép. réelle	rép. théorique	nombre	rép. réelle	rép. théorique
Primaires	40000	21,22 %	1,33 %	40000	24,13 %	3,33 %
D'ombrage	131934	69,98 %	80 %	50630	30,5 %	50 %
Secondaires	16585	8,80 %	18,67 %	75128	45,32 %	46,67 %

3.1.2.2 La répartition spatiale des sources lumineuses. Comme nous l'avons vu au paragraphe précédent, le nombre de sources lumineuses a une influence directe sur la charge globale du système. La répartition spatiale de ces sources lumineuses dans l'espace 3D de l'utilisateur peut avoir une influence sur la répartition des calculs au niveau du système distribué. Dans ce document, nous nous intéressons uniquement aux sources lumineuses ponctuelles, non orientées (pas de projecteur, ni de tube fluorescent).

Les rayons d'ombrage sont tous dirigés vers la source lumineuse considérée. En supposant que l'algorithme utilise une grille régulière pour réduire le nombre de tests d'intersection rayon-objet [AW 87], les *voxels* qui contiennent les sources lumineuses auront probablement un nombre plus important de rayons d'ombrage à traiter que les *voxels* dépourvus de source lumineuse. En particulier, si la méthode parallèle utilisée est orientée flot de rayons, les processeurs responsables des régions contenant des sources lumineuses risquent d'être surchargés (voir paragraphe 2.3.1). De même, le réseau d'interconnexion du système peut être localement surchargé autour des processeurs responsables des sources lumineuses vers qui convergent tous les rayons d'ombrage. Par contre, si l'algorithme est orienté *flot d'objets*, la position des sources lumineuses n'a pas d'influence directe sur la répartition de charge. Par contre, si la scène n'est pas recopiée sur l'ensemble des processeurs mais distribuée, alors cette localisation déterminera une partie des communications concernant la description des objets. En effet, comme les processeurs calculent entièrement l'arbre de rayons associé à chaque point dont ils ont la responsabilité, ils devront connaître les objets situés dans les différentes régions qui contiennent les sources lumineuses afin d'être en mesure de calculer complètement les rayons d'ombrage.

Nous pouvons retenir de cette analyse, que l'augmentation du nombre de sources lumineuses entraîne une augmentation de la charge globale du système. Cependant ce paramètre n'a pas d'influence sur la répartition de la charge. De même, leur localisation ne modifie pas cette répartition dans le cas où l'algorithme mis en œuvre est orienté flot d'objets.

3.1.3 Les objets

Un des facteurs qui influent sur la charge globale du système et sa répartition concerne les objets. Il semble évident que le nombre d'objets, mais aussi leur type et leur répartition spatiale influent largement sur le comportement de l'algorithme.

3.1.3.1 Les caractéristiques des objets. Le calcul d'intersection avec un objet comme un cône ou un tore est bien plus coûteux que le même calcul avec une sphère. Suivant la nature des objets, la charge du système peut être très variable. De même, des objets à la fois réfléchissants et transparents sont à l'origine d'un grand nombre de rayons secondaires qui modifient la charge.

Les différentes primitives et leur représentation. Si, pour représenter une scène, le modèle n'utilise qu'une seule primitive (par exemple un triangle), le coût d'une intersection entre un rayon et un objet sera constant. Grâce à cette représentation homogène, nous pouvons espérer que la charge sera plus facile à répartir. En effet, il suffit dans ce cas que tous les processeurs calculent le même nombre d'intersection rayon-objet pour avoir une répartition équitable. Cependant, malgré cette hypothèse d'homogénéité, il n'est pas possible de déterminer *a priori* le nombre de calculs d'intersection à réaliser.

Quand le modèle est hétérogène, le comportement de l'algorithme est encore plus difficile à prévoir. En effet, le coût de calcul d'une intersection rayon-objet peut varier énormément suivant l'objet considéré. Bien que l'utilisation d'englobants permet de réduire cet écart, le type des objets modifie le comportement dynamique de l'algorithme. Si un processeur doit tester un grand nombre de rayons avec des objets dont la représentation est complexe, il sera surchargé par rapport à un processeur qui n'a qu'à tester des objets plus simples.

Les caractéristiques optiques des objets. Deux caractéristiques optiques des objets ont une influence importante sur la génération des rayons secondaires et donc sur l'évolution de la charge du système :

1. L'objet est il réfléchissant ? Un objet réfléchissant est à l'origine d'un rayon réfléchi à chaque fois qu'il intervient dans le calcul de la couleur d'un point.

2. L'objet considéré est-il transparent ? Un objet transparent engendre un rayon transmis quand il faut évaluer sa couleur.

L'arbre de rayons associé à un point écran est déterminé par les propriétés optiques de chaque objet qui intervient dans le calcul de la couleur du point considéré. Quand un rayon primaire rencontre en premier sur son parcours un objet mat et opaque, il n'engendre aucun rayon secondaire, mais uniquement des rayons d'ombrage en direction de chacune des sources lumineuses. À l'opposé, un rayon primaire qui rencontre un objet à la fois réfléchissant et transparent va créer un arbre de rayons beaucoup plus complexe, car un rayon réfléchissant et un rayon transmis s'ajoutent aux rayons d'ombrage à calculer. De plus, chacun de ces rayons secondaires peut être à l'origine d'un sous arbre dont l'évaluation sera peut-être coûteuse.

C'est pourquoi les caractéristiques optiques des objets sont à l'origine de l'évolution dynamique et difficilement prévisible de la charge globale du système. De plus, ces caractéristiques modifient la répartition de la charge sur l'ensemble des processeurs, car un processeur qui visualise un objet transparent et réfléchissant doit réaliser plus d'opérations qu'un processeur qui visualise un objet opaque.

3.1.3.2 Le nombre d'objets et leur répartition spatiale.

On peut estimer le nombre de calculs d'intersection rayon-objet dans le pire des cas pour l'algorithme de base de lancer de rayons sans aucune optimisation. Dans ce cas, chaque rayon est testé avec l'ensemble des objets. Donc, si le nombre d'objets est égal à O, d'après l'équation 5.2 le nombre de calculs d'intersection rayon-objet I est égal à :

$$I = O.R(1 + S)(2^{p+1} - 1)$$

Par conséquent, si le nombre d'objets qui composent une scène est réduit, la charge globale du système sera raisonnable. Cependant il n'est pas possible de savoir si elle sera bien répartie. En effet, comme nous l'avons vu au paragraphe précédent, la nature des objets intervient aussi dans cette répartition. De plus, la répartition spatiale de ces objets influe sur la répartition de cet charge.

Quand un rayon primaire traverse un nombre important de régions denses en objets, la charge associée à ce rayon est importante car il est nécessaire de tester dans l'ordre des régions chacun de ces objets. Les rayons secondaires et d'ombrage issus de ce rayon seront probablement aussi coûteux car ils traverseront probablement un certain nombre ces régions denses.

3.1.4 Génération de scènes aléatoires

Cette étude nous a permis de mesurer l'influence des deux principales composantes d'une image que sont les sources lumineuses et les objets. Nous pouvons résumer ainsi cette étude :

- Les sources lumineuses sont génératrices de charge, mais elles n'influent pas sur la répartition de cette charge entre les différents processeurs quand l'algorithme utilisé est orienté flot d'objets ;

- Les objets ont une influence directe à la fois sur la charge globale du système, mais aussi sur sa répartition.

Comme la grande majorité des algorithmes actuels de lancer de rayons parallèle met en œuvre une technique de subdivision spatiale, le comportement de l'application est fortement influencée par la répartition spatiale des objets dans la scène. Pour étudier ce paramètre, nous avons créé un ensemble de scènes à l'aide de lois de probabilité. Intuitivement, une scène peut être considérée comme bien répartie si des objets sont présents dans toutes les zones de l'espace et dans les mêmes proportions. Ainsi, si une zone de l'espace est vide et une autre contient un grand nombre d'objets, nous considérons que la répartition spatiale des objets est déséquilibrée.

La répartition des charges est aussi déséquilibrée quand le temps de calcul d'une intersection rayon-objet est très variable suivant l'objet testé. Nous avons donc aussi proposé une méthode qui met en évidence l'influence du temps de calcul d'intersection rayon-objet sur la répartition de la charge.

3.1.4.1 Principales propriétés des scènes. Toutes les scènes que nous avons créées pour étudier le facteur de la répartition spatiale ont un ensemble de propriétés communes. L'englobant de la scène est un cube d'arête fixe ($a = 10\,000$). L'ensemble des objets est contenu dans ce cube.

L'observateur est placé en (12 000, -3 000, 13 000) et regarde au centre du cube (5 000, 5 000, 5 000). Deux sources lumineuses éclairent la scène, la première source est positionnée en (5 000, -13 000, 5 000), la seconde en (5 000, 5 000, 13 000), voir figure 5.2.

Afin d'étudier le seul critère de la répartition spatiale, nous avons choisi d'autoriser une unique primitive pour les objets qui est la sphère. De même, les caractéristiques optiques des sphères sont toutes identiques, à l'exception de la couleur qui n'influe pas sur la charge de travail, ni sa répartition. Le nombre d'objets composants chaque scène est variable.

L'ensemble des centres des sphères est contenu dans un cube dont les coordonnées sont $(0,0,0) - (9\,000, 9\,000, 9\,000)$. Le rayon de chaque sphère est inférieur ou égal à 1000 (10% de l'arête du cube).

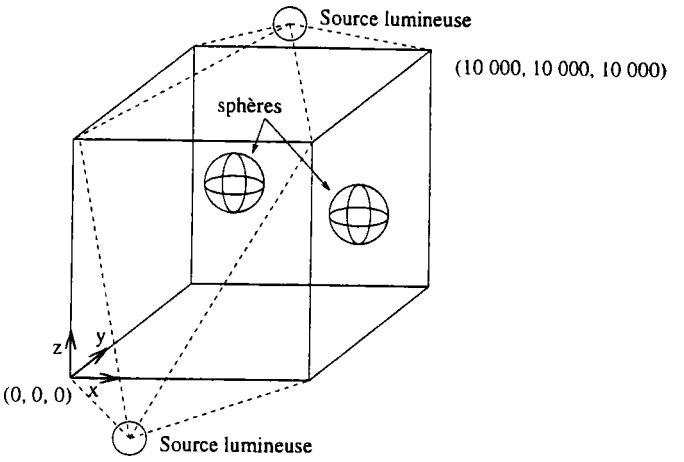


FIG. 5.2 –*représentation générale d'une scène engendrée aléatoirement.*

3.1.4.2 Répartition spatiale des objets. Nous avons développé une technique permettant d'obtenir des scènes dont la répartition spatiale est variable. Pour obtenir cet ensemble de scènes, nous avons subdivisé l'espace cubique à l'aide d'un *octree* de profondeur variable afin d'obtenir un découpage en régions de l'espace. Nous considérons qu'une scène est uniformément répartie si toutes les régions ont le même nombre d'objets. De même, une scène est très concentrée (donc déséquilibrée) si une région du cube englobant la scène contient tous les objets. En utilisant des lois statistiques, il est possible de construire des scènes qui reflètent une grande partie des cas possibles.

Pour construire l'ensemble des scènes, nous avons utilisé une loi *binomiale*. Une loi binomiale de paramètres n et p est notée $\mathcal{B}(n;\ p)$. Nous pouvons exprimer l'espérance et la variance de la variable aléatoire X suivant une loi binomiale $\mathcal{B}(n;\ p)$:

$$E(X) = np \text{ et } V(X) = np(1-p).$$

Il est possible de montrer que $P(X = k) = C_n^k p^k (1-p)^{n-k}$.

Le paramètre n est la taille de l'échantillon et p est la probabilité. Supposons qu'une expérience peut avoir deux résultats : 0 ou 1 (faux ou vrai). Le paramètre n exprime le nombre de fois où l'expérience est répétée et p est la probabilité pour que le résultat d'une expérience soit 1. Donc, si nous fixons $n = 10$ et $p = 0,1$, il est probable qu'une seule expérience aura pour résultat la valeur 1.

Pour chaque région du cube, le nombre d'objets est déterminé par une variable aléatoire de loi binomiale. Pour obtenir une scène bien répartie, il suffit de choisir une valeur moyenne pour p. À l'inverse pour obtenir une scène déséquilibrée, p doit prendre une valeur faible (proche de 0) ou bien élevée (proche de 1).

3.1.4.3 Représentation des objets. Nous avons vu que la représentation d'un objet influe directement sur le coût de calcul d'une d'intersection rayon-objet. Quand les objets

sont modélisés par des arbres CSG, la complexité de l'arbre mais aussi les différents types de primitives autorisés (sphère, bloc, cylindre, cône, pyramide, tore,...) rendent difficile l'estimation du coût d'un calcul d'intersection rayon-objet. De même, quand la scène est modélisée à l'aide d'un modèle B-REP, un objet peut être modélisé à l'aide d'un grand nombre de faces alors qu'un autre ne nécessite qu'un nombre réduit de faces pour l'approcher, c'est pourquoi le temps de calcul d'une intersection rayon-objet peut être très variable.

Comme nous avons proposé une méthode permettant d'obtenir des scènes qui étudient l'influence de la répartition spatiale des objets sur le caractère irrégulier du comportement du lancer de rayons, nous souhaitons utiliser une variable aléatoire de loi multinomiale pour créer des objets dont le temps de calcul d'intersection rayon-objet sera très variable.

En supposant, que l'on dispose d'un ensemble composé de k primitives pour modéliser des objets, nous proposons de construire des scènes comprenant un nombre n d'objets utilisant l'une de ces k primitives. Afin de n'étudier que l'influence de la représentation des objets, nous répartissons régulièrement les objets dans l'espace 3D. Si cette hypothèse est vérifiée, le déséquilibre mesuré au niveau de la répartition des calculs entre les processeurs est induit par le coût variable du calcul d'intersection rayon-objet. Pour obtenir des scènes où le temps de calcul d'une intersection est relativement constant, il suffit qu'un nombre réduit de primitives soit utilisé pour représenter une majorité d'objets. Si nous considérons la variable aléatoire X suivant une loi multinomiale $\mathcal{B}(n, p_1, p_2, ..., p_k)$, le paramètre n est la taille de l'échantillon (le nombre de fois où l'expérience est répétée) et k le nombre de résultats possibles pour une expérience. La loi multinomiale est une extension de la loi binomiale : supposons qu'une expérience peut avoir 3 résultats : 1 (associé à une sphère), 2 (associé à un bloc), 3 (associé à un cylindre). Le paramètre p_i est la probabilité pour que le résultat d'une expérience soit i ($\sum_i p_i = 1$). Donc, si nous fixons $n = 100$ et $p_1 = 0,1$; $p_2 = 0,7$; $p_3 = 0,2$, il est probable que seulement 10 expériences auront pour résultat la valeur 1 (une sphère), 70 expériences auront la valeur 2 (un cube) et 20 expériences auront pour résultat 3 (un cylindre). Donc, pour obtenir une scène dont le temps de calcul d'une intersection rayon-objet est relativement constant, il suffit de privilégier un nombre réduit de primitives en leur associant une probabilité élevée, alors que les autres primitives auront une probabilité très faible d'être utilisées pour représenter un objet. Par contre, pour obtenir des scènes dont le temps de calcul d'une intersection rayon-objet sera très variable, la valeur de p_i doit être proche de $\frac{1}{k}$.

En combinant les deux approches présentées (scènes où la répartition spatiale des objets est variable, scènes où le temps de calcul d'une intersection rayon-objet est variable), il est possible d'obtenir des scènes dont le déséquilibre de la répartition des tâches sera induit par deux facteurs simultanément : la répartition spatiale des objets et leur représentation.

3.1.5 Algorithme de décision pour le lancer de rayons

L'étude menée au paragraphe 3.1 nous a permis de mettre en évidence le caractère irrégulier du lancer de rayons. De plus, le temps de chaque tâche est fortement dépendant des données du problème à résoudre. Quand la scène à visualiser possède certaines caractéristiques (répartition spatiale satisfaisante, représentation des objets homogènes), la répartition de la charge est plus facile, car les tâches ont un temps d'exécution moins irrégulier. Par contre, si la scène à visualiser utilise une représentation des objets dont le

temps d'évaluation est très irrégulier ou bien si les objets sont concentrés dans une partie restreinte de l'espace, le comportement de l'application FTII sera très fortement irrégulier. C'est pourquoi, il est difficile de proposer un algorithme d'équilibre de charge dynamique qui sera le plus performant dans tous les cas de figure. Cependant, l'algorithme de décision général pour les applications FTII apporte une première solution (cf paragraphe 6, chapitre 4). Si l'utilisateur peut définir plus précisement le type de scène qu'il souhaite calculer, cette solution peut être affinée. Par exemple, nous pouvons distinguer trois cas :

1. Visualisation de molécules. Si le but de l'utilisateur est de visualiser des molécules, nous pouvons estimer que les différents types de primitive à utiliser seront peu nombreux et que la plupart des molécules sera représenté par des sphères. Dans ce cas, le temps de calcul d'une intersection rayon-objet pourra être considéré comme constant. Le caractère irrégulier de l'application FTII proviendra alors essentiellement de la répartition spatiale des objets.

2. Visualisation en architecture. Quand un architecte veut estimer l'impact d'un futur bâtiment dans un environnement, il souhaite obtenir une vue d'ensemble. Donc, tous les objets ont la même importance (ou niveau de la qualité de la représentation) et de plus, l'espace sera très largement occupé car l'avant-plan de la scène a autant d'importance que l'arrière-plan. Donc, le caractère irrégulier de l'application sera le fait des qualités optiques variables des objets à représenter et aussi le fait du coût de calcul d'une intersection rayon /objet.

3. Visualisation en CAO. Quand un ingénieur demande un rendu réaliste d'une pièce mécanique, seule cette pièce a une réelle importance. C'est pourquoi le fond de la scène, mais aussi les bords sont vides de tout objet. L'essentiel du travail à réaliser est regroupé au centre de l'image. La répartition spatiale est dans ce cas fortement déséquilibrée. De plus la représentation d'un objet peut être plus ou moins complexe et le temps de calcul d'une intersection rayon-objet est très variable.

3.2 L'algorithme de lancer de rayons

Après avoir montré l'intérêt de présenter le lancer de rayons comme une application FTII particulière, nous présentons à l'aide de la méthode de Grady Booch l'architecture générale du lancer de rayons utilisé[1] [Boo 94].

3.2.1 Le lancer de rayons : une application FTII particulière

Comme le lancer de rayons est une application FTII, nous sommes en mesure d'utiliser l'environnement pour les applications FTII afin d'obtenir une implantation parallèle. Bien que les solutions parallèles utilisant une répartition statique des calculs proposées dans la littérature obtiennent des résultats corrects, notre méthode offre plusieurs avantages :

- une méthode générale : indépendante de l'application et de la machine parallèle cible ;

- une facilité de programmation : le parallélisme est géré par l'environnement, il est donc complètement transparent pour l'utilisateur ;

1. Cet algorithme a été développé dans le cadre du projet fédérateur REGAIN.

- les performances obtenues sont très satisfaisantes.

De plus le caractère FTII du lancer de rayons n'est pas remis en cause par la représentation des objets (CSG, B-REP), ni par la majorité des optimisations existantes. En effet, les techniques d'accélérations utilisant des subdivisions spatiales (régulière, octree, BSP) sont très efficaces et vérifient les hypothèses des applications FTII. De même, les notions d'englobants sont applicables dans le cadre de l'environnement FTII. Enfin les techniques directionnelles, qui permettent par exemple de réduire le nombre d'objets à tester sur le parcours d'un rayon d'ombrage [HG 86], sont des optimisations valides pour l'environnement FTII.

Par contre, il existe un nombre de techniques d'accélération qui ne sont pas utilisables directement dans l'environnement FTII. Nous pouvons donner deux exemples. Si la notion de cohérence est utilisée pour déterminer les objets à tester pour deux pixels très proches [Gla 89, J. Arvo et D. Kirk, chap. 6], il est possible que des communications soient nécessaires entre deux tâches qui ont des pixels voisins. Cependant, il semble que cette technique soit peu utilisée car les bénéfices sont faibles par rapport au surcoût de cette méthode. De même, si l'utilisateur applique l'algorithme d'antialliassage qui met en place un post-traitement calculant la couleur d'un pixel en fonction de la couleur des pixels voisins, ce post-traitement ne vérifie pas la propriété d'indépendance des tâches (des communications sont nécessaires), nous pouvons néanmoins remarquer que le temps de post-traitement est faible par rapport au temps de calcul de l'image et qu'il n'est donc pas indispensable de le paralléliser.

Dans notre étude expérimentale, nous avons choisi de dupliquer la base de données de la scène dans les mémoires de chaque processeur. Nous avons fait ce choix pour deux raisons principales :

- la répartition des objets entre les processeurs est à l'origine de messages (échange des descriptions des objets) dans le cas d'une méthode orientée flot d'objets. Ces messages se confondent avec les messages induits par la stratégie d'équilibre de charge dynamique et rend donc difficile l'analyse des résultats. De plus les problèmes de répartition de la base de données ont largement été étudiés dans la littérature [GPL 89, BBPA 94] et les résultats obtenus sont satisfaisants.

- La mémoire disponible au niveau de chaque processeur est de plus en plus importante sur les nouvelles machines parallèles (par exemple 32 Mo par processeur sur la CM-5) et il est donc possible de dupliquer des scènes de taille conséquente sur tous les processeurs.

3.2.2 Architecture globale de l'application

Le but de l'algorithme de visualisation développé est d'être indépendant des différentes techniques mises en œuvre, en particulier, l'algorithme est indépendant :

- de la machine cible : l'algorithme ne fait aucune hypothèse sur l'architecture de la machine, ni sur le système d'exploitation utilisé. La seule contrainte est que la machine doit disposer d'un compilateur C++.

- De la représentation de la scène à visualiser : l'algorithme proposé peut être appliqué à une scène représentée à l'aide d'un modèle *CSG (Constructive Solid Geometry)* ou *B-REP (Boundary Representation)*.

- Des optimisations développées : le lancer de rayons doit pouvoir être facilement étendu afin de prendre en compte une nouvelle optimisation développée pour l'application.

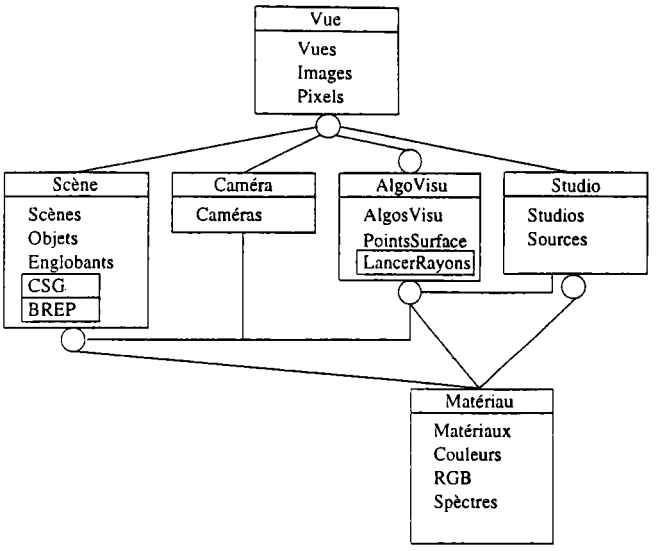


FIG. 5.3 –*Diagramme des catégories principales.*

La programmation orientée objet nous permet de développer notre application en respectant chacune de ces contraintes. De plus, nous bénéficions des qualités des logiciels orienté objets : portabilité, extensibilité, modularité, mise à jour aisée. L'architecture globale de la partie lancer de rayons est présentée à l'aide du diagramme des principales catégories (voir figure 5.3). L'algorithme est articulé autour de six catégories principales. Chaque catégorie est chargée de la gestion d'un sous-ensemble de l'application :

- la catégorie *Caméra* gère les paramètres relatifs à l'observateur de la scène parmi lesquels la position de l'observateur, le vecteur de visée, la distance focale.

- La catégorie *Studio* gère les différentes sources lumineuses et tous les paramètres concernant l'éclairage de la scène.

- La catégorie *Matériau* est chargée de la représentation des couleurs des objets à l'aide de différents modèles (RGB, spectres).

- La catégorie *Scène* représente la scène à visualiser, elle est composée de plusieurs sous-catégories présentées au paragraphe 3.2.3. Elle fournit des méthodes homogènes pour connaître la position d'un objet, mais aussi sa couleur, son englobant, etc...

- L'algorithme de lancer de rayons est représenté par la catégorie *AlgoVisu* dont il fait partie. Une présentation plus détaillée de sa structure est présentée au paragraphe 3.2.4.

- La catégorie *Vue* coordonne l'ensemble des objets qui composent l'application. Elle contient l'image obtenue en résultat de l'application.

3.2.3 Représentation de la scène

La représentation de la scène est indépendante du modèle utilisé. Une scène est composée de plusieurs objets. Chaque objet est caractérisé par un certain nombre de propriétés. Il possède en particulier des propriétés optiques (couleur, coefficient de réflexion, de transparence,...), un englobant ainsi qu'une représentation géométrique. L'ensemble des classes qui caractérisent une scène ainsi que les différentes relations qui existent entre elles sont détaillés à l'aide du diagramme des classes de la figure 5.4.

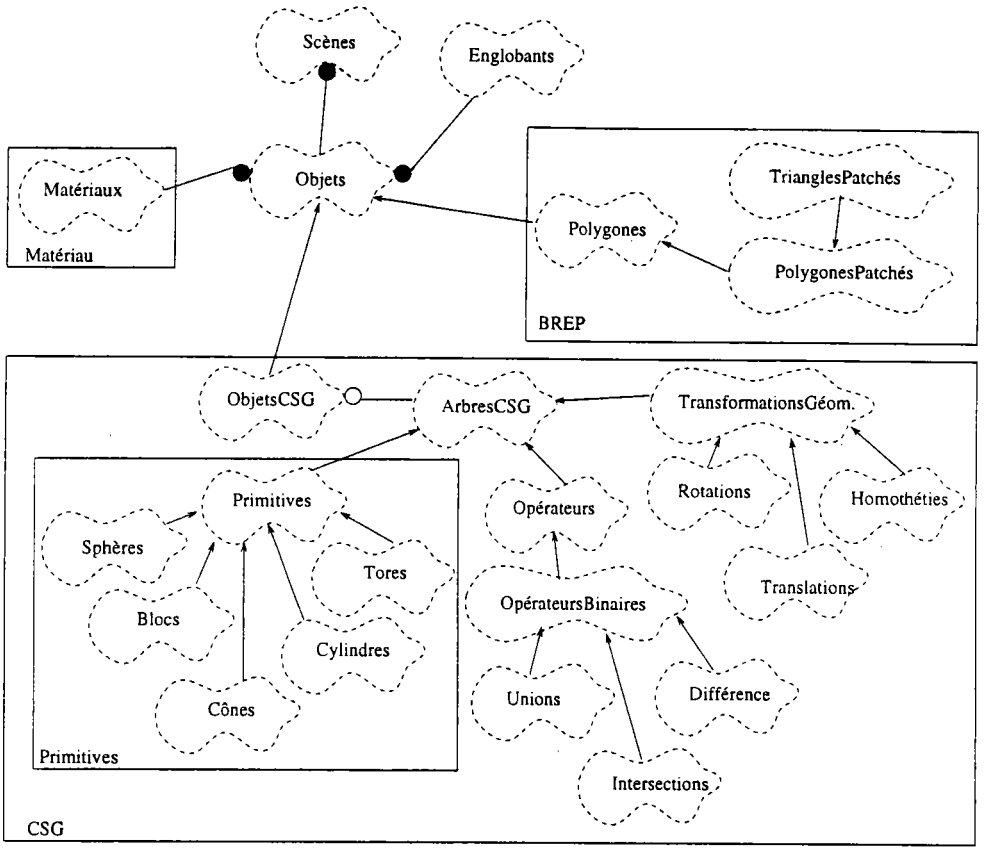


FIG. 5.4 -*Diagramme des classes de la catégorie 'Scène'.*

Le modèle CSG est représenté par la catégorie *CSG* qui est composée d'une sous catégorie pour les primitives. Une primitive est toujours unitaire. Pour obtenir un objet de taille quelconque et de position variable, il est nécessaire d'appliquer des transformations géométriques (rotation, translation, homothéties). Chaque objet peut être composé de plusieurs primitives combinées à l'aide d'opérateurs booléens (union, intersection, différence). L'arbre CSG résultant est utilisé pour calculer l'intersection de l'objet avec un rayon donné.

Le modèle BREP (représenté par la catégorie *BREP*) permet une représentation des objets à l'aide de polygones. Les polygones peuvent avoir un nombre de sommets quelconques. Les triangles *patchés* sont des polygones à trois sommets dont on connaît les normes aux sommets grâce auxquels l'algorithme interpole la normale en tout point de la surface.

Pour éviter des calculs inutiles, chaque objet possède une boîte englobante. Cette boîte est un parallélipipède rectangle dont les faces sont parallèles aux axes. Ainsi, avant

de calculer l'intersection entre un objet et un rayon, l'algorithme commence par tester si une intersection existe entre l'englobant et le rayon considéré (voir paragraphe 2.2.1.1).

3.2.4 Visualisation de la scène

L'algorithme de lancer de rayon est développé par la catégorie *AlgoVisu*. L'algorithme est indépendant des optimisations qui peuvent être appliquées et du modèle d'éclairement choisi pour calculer l'illumination d'un point. Les différentes relations qui existent entre les classes qui définissent l'algorithme complet sont présentées par le diagramme de la figure 5.5.

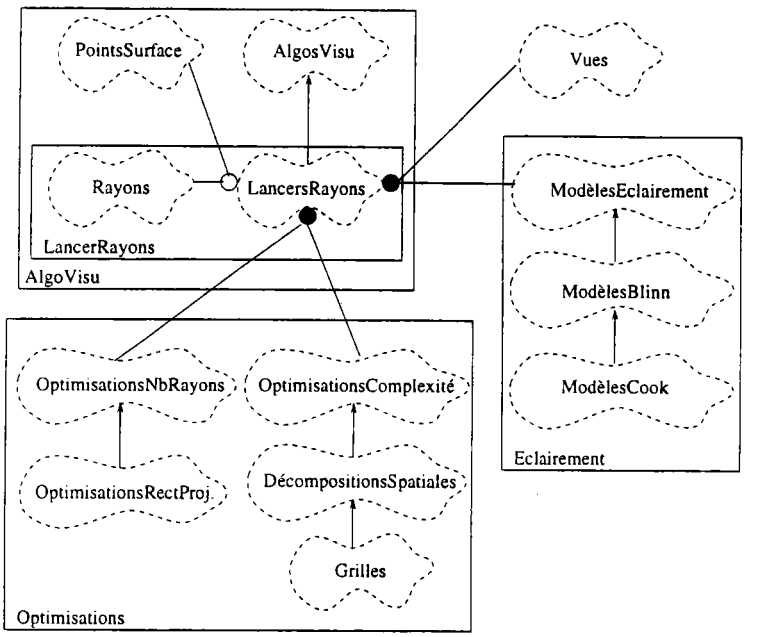


FIG. 5.5 –*Diagramme des classes pour le lancer de rayons.*

Les notions de *PointSurface* et *Rayons* sont définies. Un objet de type *PointSurface* est le résultat du calcul d'intersection entre un un objet de type *Rayons* et l'ensemble des objets de la scène. En particulier, un objet de type *PointSurface* possède comme attributs, la position de l'intersection dans l'espace 3D, le nom de l'objet le plus proche rencontré, la normale en ce point, etc... L'ensemble de ces informations est utilisé pour calculer la couleur du point en fonction du modèle d'éclairement.

Pour éviter de calculer l'intersection d'un rayon avec tous les objets de la scène, des optimisations sont développées. Ces optimisations sont indépendantes de l'algorithme de lancer de rayons de base. Deux types d'optimisations sont distinguées : les optimisations qui réduisent le nombre de rayons (*OptimisationNbRayons*) et celles qui réduisent le nombre d'objets testés pour un rayon donné (*OptimisationsComplexité*). La classe *Optimisations-RectanglesProjetés*) permet d'éviter de lancer des rayons primaires inutiles en utilisant la projection des englobants à l'écran (voir paragraphe 2.2.2.2). Les objets de type *Grilles* permettent de réduire le nombre d'objets à tester pour un rayon. Une grille régulière est construite et seuls les objets contenus par des régions traversées par le rayon considéré sont testés (voir paragraphe 2.2.1.2).

L'architecture de l'application permet d'avoir un algorithme robuste et modulable. Grâce aux avantages de la méthode orientée objet, elle est facilement extensible pour

prendre en compte de nouvelles possibilités : nouveaux types de primitives, nouvelles optimisations de la complexité, nouvelle technique d'accélération du calcul comme le parallélisme.

3.2.5 Parallélisation à l'aide de l'environnement FTII

Comme le lancer de rayons est une application FTII, nous avons parallélisé cette application à l'aide de l'environnement FTII. Pour fournir à l'environnement une fonction qui calcule le travail associé à une tâche identifiée par un entier, nous avons créé une nouvelle classe *LancersRayonsPar* qui hérite de la classe *LancersRayons*. Une nouvelle fonction membre est fournie par *LancersRayonsPar* , c'est la fonction *CalculTache* qui visualise la couleur de chaque pixel associé à la tâche en cours d'exécution à l'aide de l'algorithme séquentiel. C'est pourquoi, si une nouvelle optimisation ou bien une nouvelle représentation des objets est proposée par l'algorithme séquentiel, elle est directement prise en compte par l'algorithme parallèle.

Cette solution est suffisamment souple pour permettre une maintenance et une extension aisée de l'application parallèle.

3.3 Validation expérimentale du modèle matriciel

Afin de valider expérimentalement le modèle matriciel proposé au chapitre 3, nous avons comparé les indices de performance théorique fournis par le modèle et les indices expérimentaux obtenus sur une CM-5 pour deux scènes particulières [KHHG 98].

TAB. 5.2 – *Comparaison des résultats théoriques et pratiques pour la scène Sphereflake.*

Stratégie	Sphereflake			
	théorique		pratique	
Nombre de processeurs	32 p.	64 p.	32 p.	64 p.
Sans équilibre de charge (temps en s.)	183,17	93,96	183,17	93,96
accélération	23,51	45,82	23,51	45,82
efficacité	73 %	72 %	73 %	72 %
Alg. serveur initiative (temps en s.)	138,36	71,41	139,07	71,61
accélération	31,12	60,29	30,96	60,12
efficacité	97 %	94 %	97 %	94 %
temps min.	132,38	64,77	130,96	65,38
Nbre d'équi.	125	153	121	156
Nbre de proc. interrogés	1573	5653	1391	5257
Nbre de tâches déplacées	148	112	127	107
Alg. client-serveur (temps en s.)	139,82	69,75	143,47	71,11
accélération	30,79	61,73	30,01	60,55
efficacité	96 %	96 %	94 %	95 %

Dans un premier temps, nous avons construit les jeux de données pour les scènes de E. Haines *Mount* et *Sphereflake* [Hai 87]. La taille de la base de données est 2, la profondeur de l'arbre de rayons est fixée à 5 (les rayons primaires étant le premier niveau de l'arbre). La scène *sphereflake* est construite à l'aide de 1 polygone et 91 sphères. La scène *mount* est composée de 32 polygones et 4 sphères. Les temps d'exécution séquentiel des deux scènes considérées sont respectivment de 3750,8 et de 4305,5 secondes.

Le temps de chaque tâche (groupe de pixels) a été calculé à l'aide de l'algorithme séquentiel. Puis, en fonction de la machine cible et de la stratégie d'équilibre de charge, le modèle matriciel a été appliqué. Les résultats obtenus sont présentés dans les tableaux 5.2 et 5.3.

Nous remarquons que les résultats expérimentaux et théoriques sont très proches. L'erreur entre les indices estimés par le modèle matriciel et les indices mesurés est inférieure à 7% pour le temps d'exécution, l'accélération et l'efficacité. Cependant, d'autres indicateurs comme le nombre de phases d'équilibre de charge ou bien le nombre de requêtes émises sont approchés avec un erreur inférieure à 20%. Le modèle matriciel est donc un outil fiable qui permet d'estimer les performances d'une application FTII avant même d'avoir implanter réellement son code.

TAB. 5.3 – *Comparaison des résultats théoriques et pratiques pour la scène Mount.*

Stratégie	Mount			
	théorique		pratique	
Nombre de processeurs	32 p.	64 p.	32 p.	64 p.
Sans équilibre de charge (temps en s.)	246,45	124,88	246,45	124,88
accélération	15,22	30,04	15,22	30,04
efficacité	48 %	47 %	48 %	47 %
Alg. serveur initiative (temps en s.)	124,01	63,97	126,35	68,51
accélération	30,25	58,63	29,69	54,75
efficacité	95 %	92 %	93 %	86 %
temps min.	113,57	54,70	113,59	55,66
Nbre d'équi.	286	373	240	338
Nbre de proc. interrogés	2223	7437	1551	5982
Nbre de tâches déplacées	566	537	454	440
Alg. client-serveur (temps en s.)	125,92	65,47	130,34	66,87
accélération	29,79	57,29	28,78	56,09
efficacité	93 %	90 %	90 %	88 %

3.4 Analyse de performances des algorithmes distribués

Afin de comparer les stratégies distribuées, tous les tests ont été réalisés avec différentes scènes composées de 50 à 450 primitives. Nous présentons dans cette étude les résultats obtenus pour une scène particulière représentant un court de tennis [KHHG 96a]. Elle est composée de plus de 50 cubes et sphères. La scène est modélisée à l'aide d'un modèle CSG (*Constructive Solid Geometry*). La résolution utilisée pour effectuer les tests est de 640 × 480 points. Seuls les rayons primaires et d'ombrage ont été calculés. Le seuil T, qui est utilisé pour déterminer si un processeur est surchargé ou non, est fixé à 1. Dans le but d'analyser les performances de nos algorithmes, nous avons étudié expérimentalement la valeur du seuil S. Comme Eager *et al*, nous avons utilisé la valeur $S = 1$ (une unique région est échangée à chaque étape d'équilibre de charge). Ensuite, nous avons considéré la valeur dynamique $S = \frac{w_i}{2}$ où w_i est la charge du processeur P_i. Dans ce cas, un équilibre de charge local est réalisé à chaque phase d'équilibre de charge.

3.4.1 Temps d'exécution

Le temps d'exécution est le premier critère pour évaluer les performances d'un algorithme. Le tableau 5.4 récapitule les temps d'exécution des algorithmes source initiative, serveur initiative, hybride et client-serveur. Les résultats montrent que pour les trois stratégies distribuées, la gestion dynamique de S améliore sensiblement les performances obtenues. Ce résultat est opposé à celui donné par Eager et al qui affirme que les performances globales ne dépendent pas de la valeur choisie pour S.

TAB. 5.4 -*Comparaison des algorithmes distribués.*

Nombre de processeurs	1	32	64
source initiative S=1 (temps en s.)	2257,5	87,3	48,2
serveur initiative S=1 (temps en s.)		74,2	40,3
algorithme hybride S=1 (temps en s.)		86,5	40,1
source initiative S=L/2 (temps en s.)		82,9	47,3
serveur initiative S=L/2 (temps en s.)		74,6	37,8
algorithme hybride S=L/2 (temps en s.)		74,1	40,7
client-serveur (temps en s.)		75,3	39,6

Comme nous l'avons montré théoriquement, le temps d'exécution de l'algorithme source initiative est supérieur à ceux des algorithmes serveur initiative et client-serveur. De plus, dans notre étude, l'algorithme serveur initiative semble légèrement plus performant que l'algorithme client-serveur.

3.4.2 Nombre de phases d'équilibre de charge

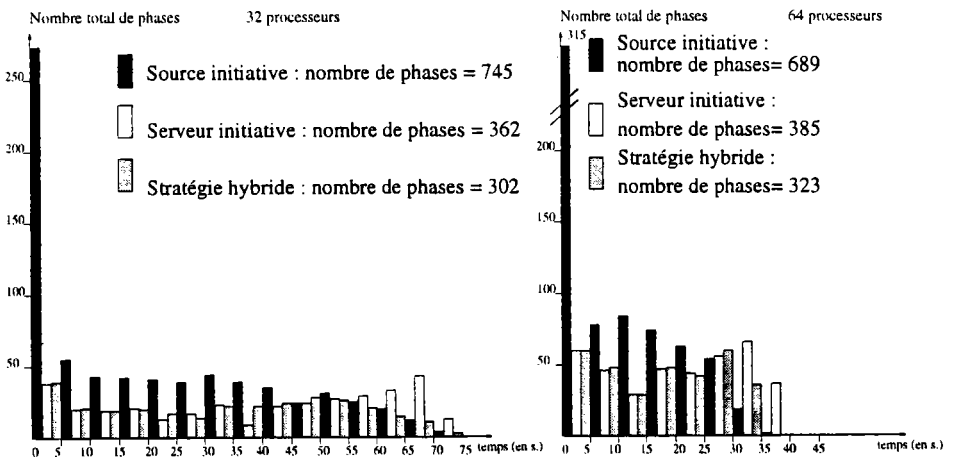


FIG. 5.6 -*Nombre de phases d'équilibre de charge pour* $S = 1$.

Les figures 5.6 et 5.7 présentent le nombre de phases d'équilibre de charge réalisées par les trois stratégies distribuées pour $S = 1$ et $S = \frac{w_i}{2}$ respectivement. Nous pouvons remarqué que les comportements des algorithmes sont très proches pour $S = 1$ et $S = \frac{w_i}{2}$. Le nombre de phases d'équilibre de charge est plus important quand $S = 1$. La charge est plus difficilement distribuée, car à chaque phase d'équilibre de charge au plus une tâche est déplacée.

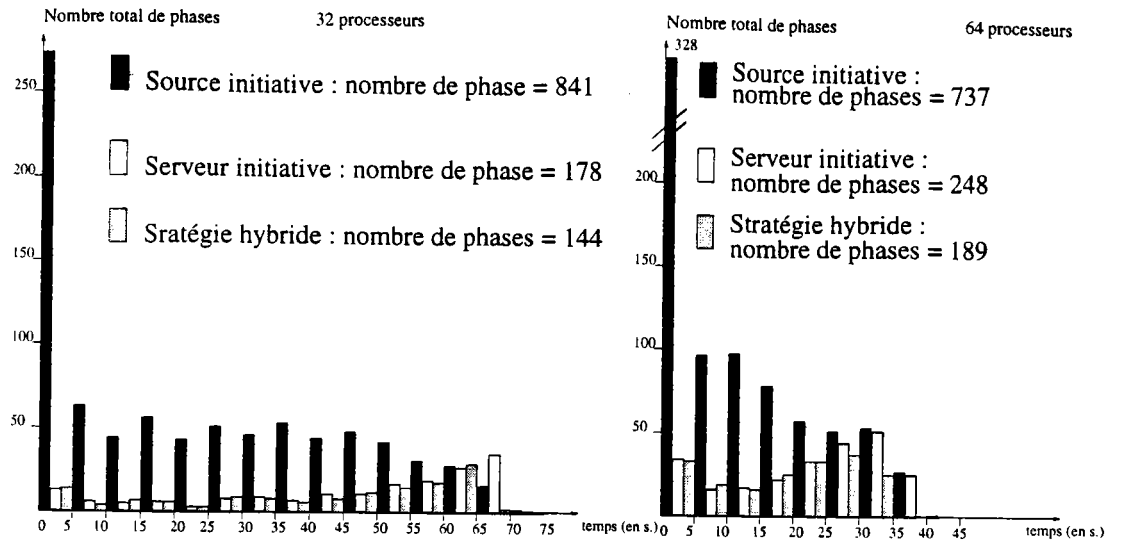


FIG. 5.7 –*Nombre de phases d'équilibre de charge pour* $S = \frac{w_i}{2}$.

Le comportement de la stratégie source initiative n'est pas surprenant : au début de l'exécution, le nombre de phases d'équilibre de charge est très important et se réduit au fur et à mesure de l'exécution. À l'opposé, l'algorithme serveur initiative est très efficace en début d'exécution puis ses performances se dégradent en fin d'exécution. Enfin, l'algorithme hybride observe le comportement le plus stable, le nombre de phases d'équilibre de charge reste pratiquement constant tout au long de l'exécution. Grâce à cette constance, elle réalise moins de phases d'équilibre de charge que les algorithmes source et serveur initiative, mais cette stratégie n'apporte pas d'amélioration significative en terme de temps d'exécution.

3.4.3 Étude de l'algorithme hybride en fonction du seuil P

Il est utile de rappeler que l'algorithme hybride modifie sa stratégie pendant l'exécution quand il détecte que $P\%$ des processeurs sont déchargés. La valeur de P est à déterminer. La figure 5.8 montre que pour $S = 1$ et $S = \frac{w_i}{2}$, il est souhaitable de modifier la stratégie quand un peu moins de 50 % des processeurs sont déchargés. Nous pouvons aussi remarqué que les performances de la stratégie ne sont pas très sensibles à la valeur de P.

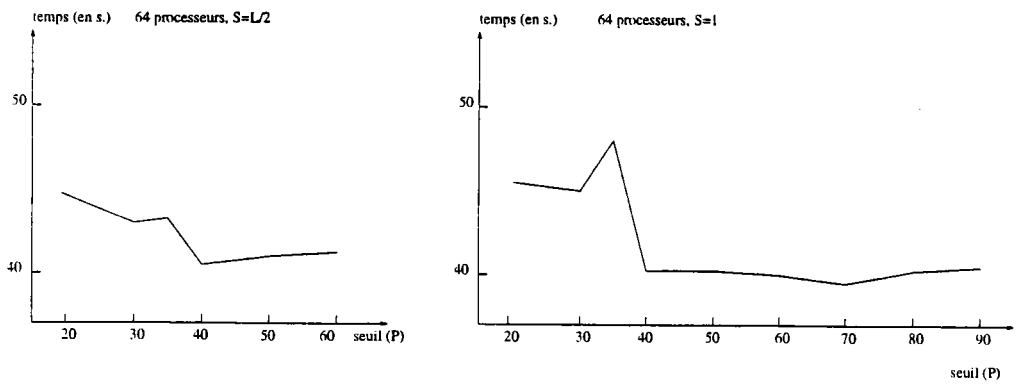


FIG. 5.8 –*Étude pratique du seuil P pour l'algorithme hybride.*

Les principaux résultats obtenus pour les stratégies distribuées peuvent être résumés

ainsi :

- pour les trois stratégies distribuées, la gestion dynamique du seuil S améliore les performances par rapport à une gestion statique de ce seuil ;
- l'algorithme hybride réduit le nombre de phases d'équilibre de charge sans améliorer de façon significative les performances des stratégies source et serveur initiative ;
- l'algorithme serveur initiative semble plus efficace que l'algorithme client serveur.

L'ensemble de ces remarques est conforme à l'analyse que nous avons faite à l'aide du modèle matriciel (cf paragraphe 4.5.2, chapitre 3).

3.5 Analyse de performances de la stratégie semi-distribuée

Comme pour les stratégies distribuées, nous avons développé la stratégie semi-distribuée sur une CM-5. Nous présentons dans cette partie les résultats obtenus pour trois scènes. Comme pour la validation du modèle matriciel, nous avons utilisé les deux scènes de E. Haines *mount* et *sphereflake* [Hai 87]. La dernière scène utilisée, la scène *Cubesphere* est composée de 343 sphères et elle représente le cube des couleurs. Le seuil T qui détermine si un processeur est surchargé ou non, est fixé à 1.

3.5.1 Temps d'exécution

Comme pour les stratégies distribuées, nous présentons en premier lieu le temps d'exécution de l'application pour trois scènes particulières. Le tableau 5.5 récapitule les temps d'exécution pour les algorithmes source et serveur initiative, client-serveur et semi-distribué. Nous constatons que l'algorithme semi-distribué est très efficace, et que ses performances sont légèrement supérieures à celles des algorithmes client-serveur et serveur initiative. De plus, nous remarquons que la stratégie semi-distribuée réduit le temps d'exécution de l'application sans équilibre de charge dynamique de 30 à 50 % en moyenne.

TAB. 5.5 – *Comparaison des temps d'exécution (en secondes).*

Stratégie	Sphereflake		Mount		CubeSphere	
	32 p.	64 p.	32 p.	64 p.	32 p.	64 p.
Statique	187,45	94,92	264,07	139,46	92,23	56,08
Source initiative	148,28	91,15	155,27	87,96	60,88	39,34
Serveur initiative	135,17	67,96	133,74	70,86	54,97	31,19
Semi-distribuée	134,07	67,03	128,64	70,04	58,15	30,01
Client-serveur	136,70	68,26	134,33	70,08	58,99	31,77

3.5.2 Analyse de la structuration optimale

Comme le réseau de communication de la CM-5 est un *fat tree* [Lei 85], nous pouvons supposer que le coût d'un message au niveau global est sensiblement égal à celui d'un message local. Ainsi, dans notre étude nous fixons k à 1. De plus, comme chaque région a la même taille, nous pouvons déduire le découpage optimal de la CM-5 (voir

TAB. 5.6 –*Structuration optimale du système distribué pour une CM-5.*

Nombre de processeurs (n)	32	64	128
Nombre de régions (r)	8	8 ou 16	16
Nombre de processeurs par région (m)	4	8 ou 4	8

paragraphe 4.4.3.2, chapitre 3). Les résultats de cette analyse sont présentés dans le tableau 5.6.

L'évolution observée expérimentalement du nombre de requêtes émises par la stratégie semi-distribuée est présentée à la figure 5.9. Le surcoût de la stratégie semi-distribuée est minimum pour 64 processeurs quand le système est structuré en 8 régions. Pour chaque scène, le nombre de messages émis est minimum quand les processeurs sont répartis dans 8 régions distinctes. D'après notre analyse théorique, le nombre optimal de régions pour 64 processeurs devrait être compris entre 8 et 16 régions. Comme l'analyse ne considère pas les deux optimisations de la stratégie qui peuvent être à l'origine de requêtes supplémentaires, les résultats pratiques sont conformes à ceux fournis par l'analyse.

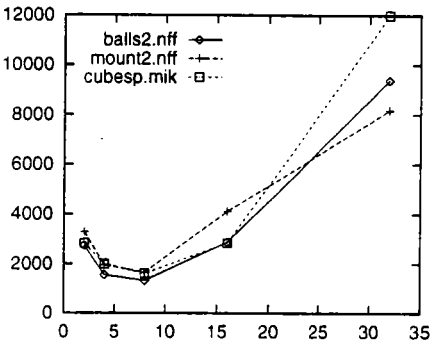


FIG. 5.9 –*Nombre de requêtes engendrées par la stratégie semi-distribuée pour 64 processeurs.*

3.5.3 Nombre de phases d'équilibre de charge

Les figures 5.10 et 5.11 présentent le nombre de phases d'équilibre de charge déclenchées par la stratégie semi-distribuée pour les 3 scènes nommées *sphereflake, mount* et *cubeSphere.* Nous observons que la scène *sphereflake* est très régulière et c'est pourquoi il y a peu de phases d'équilibre de charge au début de l'exécution. À l'opposé, pour les deux autres scènes, des phases d'équilibre de charge sont nécessaires dès le début de l'exécution.

De plus, nous remarquons qu'il y a beaucoup plus de phases d'équilibre de charge locales que globales. En effet, pour déclencher une phase globale, il ne doit plus y avoir aucune tâche disponible au niveau de la région. Dans ce cas, s'il y a m processeurs dans chaque région, il y aura exactement m phases d'équilibre de charge locales infructueuses et seulement une phase globale. Enfin, s'il existe un déséquilibre local au niveau de la région r, la stratégie semi-distribuée ne pénalise pas tout le système car seules des phases d'équilibre de charge locales sont déclenchées (voir le début de l'exécution pour la scène *CubeSphere* par exemple.

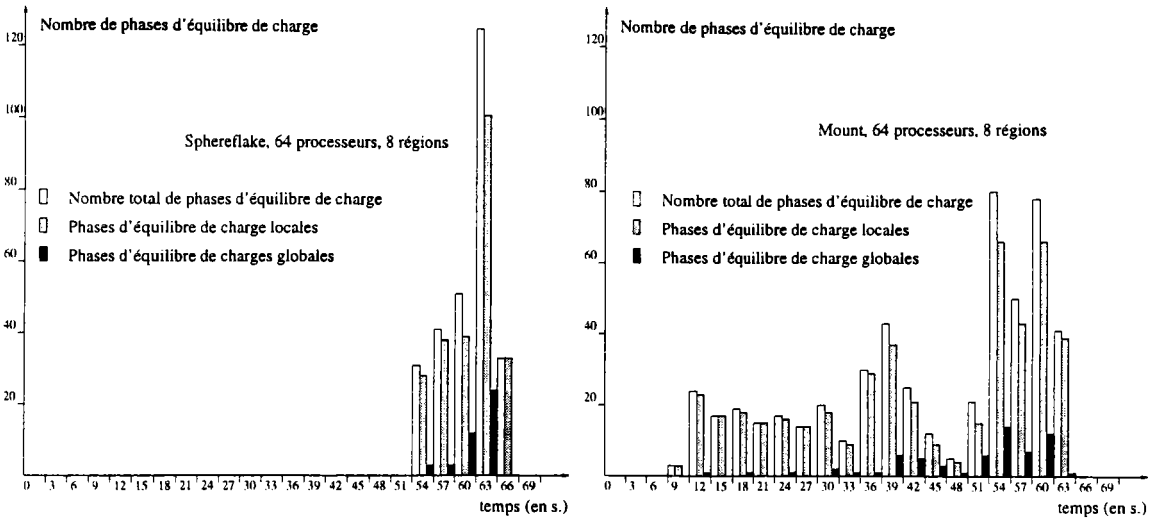


FIG. 5.10 –*Nombre de phases d'équilibre de charge déclenchées pour les scènes de E. Haines.*

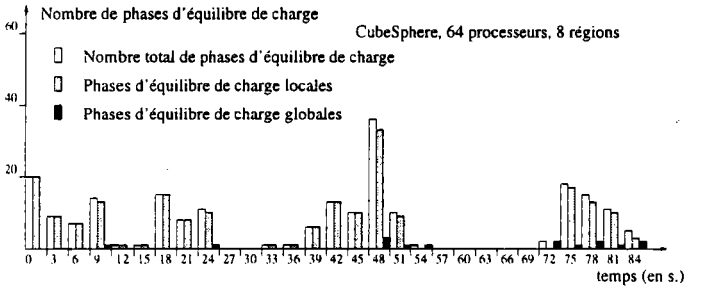


FIG. 5.11 –*Nombre de phases d'équilibre de charge déclenchées pour la scène CubeSphere.*

Il est évident, que la stratégie semi-distribuée réalise moins de phases d'équilibre de charge que les stratégies distribuées. (cf tableau 5.7). Ce résultat est conforme à notre objectif initial. Cependant, le réseau d'interconnexion de la CM-5 est très efficace et donc le surcoût en communication d'une phase d'équilibre de charge est faible. C'est pourquoi, même si nous avons réduit le nombre de messages induits par l'équilibrage dynamique de la charge, le temps d'exécution n'a pas été réduit de façon très sensible. Néanmoins, si le coût d'une communication est plus élevé, minimiser le nombre de messages engendrés devrait permettre de réduire le temps d'exécution.

TAB. 5.7 –*Nombre de phases d'équilibre de charge.*

Stratégie	Sphereflake		Mount		CubeSpheres	
	32 p.	64 p.	32 p.	64 p.	32 p.	64 p.
Source initiative	3963	3914	3869	3732	3905	3737
Serveur initiative	304	416	477	836	531	882
Semi-distribuée	162	239	322	467	300	538

4 Conclusion

Nous avons présenté dans ce chapitre un exemple d'application FTII qu'est le lancer de rayons. Après, une étude bibliographique de cet algorithme et des solutions parallèles existantes, nous avons montré le caractère irrégulier de cet algorithme. En particulier, nous avons mis en évidence que le comportement de l'application est fortement dépendant de la scène à visualiser. À l'aide de l'environnement FTII, nous avons parallélisé cet algorithme et une analyse expérimentale sur CM-5 a été menée. Nous résumons ainsi les conclusions auxquelles nous sommes parvenus :

- les résultats obtenus à l'aide du modèle matriciel sont très proches des résultats expérimentaux observés sur la CM-5.

- Les stratégies distribuées sont efficaces pour le lancer de rayons. En effet, les trois stratégies distribuées proposées améliorent de façon significative les performances de l'algorithme sans équilibre de charge dynamique. Nous avons constaté que la stratégie source initiative est moins performante que les stratégies serveur initiative et hybride. Bien que la stratégie hybride réduise le nombre de messages de contrôle, ses performances sont très proches de celles de la solution serveur initiative. Ces deux dernières stratégies sont aussi performantes que la stratégie client-serveur et dans certains cas, elles prennent l'avantage.

- La stratégie semi-distribuée permet de conserver les avantages des stratégies distribuées (extensibilité) et des stratégies centralisées (surcoût raisonnable) en évitant les principaux inconvénients de ces deux types de méthode (nombre de messages induits et goulot d'étranglement). Cette stratégie est particulièrement bien adaptée au lancer de rayons.

- L'environnement FTII nous a permis de paralléliser efficacement l'algorithme de lancer de rayons sans se soucier de la machine cible ni de la stratégie d'équilibre de charge dynamique appliquée.

Conclusion et perspectives

Ce travail s'inscrit dans le cadre de l'étude de la parallélisation d'une famille particulière d'applications : les applications FTII. Dans un premier temps, nous avons classifié les stratégies d'équilibre de charge dynamique dans le cadre du modèle MIMD. Nous avons exprimé les avantages et les inconvénients de chaque type d'algorithme (client-serveur, distribué ou semi-distribué). Nous avons aussi présenté les modèles généralement utilisés pour évaluer les performances théoriques des stratégies d'équilibre de charge. En particulier, nous avons distingué les méthodes basées sur les files d'attente des méthodes matricielles.

Après cette étude bibliographique, nous avons défini formellement les applications FTII que nous souhaitons paralléliser. Concrètement, une application FTII est caractérisée par un ensemble fini de tâches qui sont indépendantes et irrégulières. Nous avons exprimé pourquoi une implémentation parallèle de ces algorithmes est souhaitable. Pour que la parallélisation soit efficace, il est nécessaire de proposer des stratégies d'équilibre de charge dynamique adaptés aux algorithmes FTII car il est difficile de proposer une répartition statique satisfaisante du fait du caractère irrégulier d'une tâche.

Nous avons ainsi développé cinq stratégies d'équilibre de charge pour les applications FTII. Afin de comparer sur un plan théorique ces solutions, nous avons modélisé le comportement de chaque stratégie dans un modèle matriciel. Cependant, le caractère adaptatif des stratégies d'équilibre de charge dynamique ne nous permet pas d'utiliser les modèles matriciels existants pour analyser nos algorithmes. C'est pourquoi, nous avons proposé un modèle matriciel adapté au mode MIMD. Pour chaque stratégie, nous avons exprimé sa complexité en communication et une étude statistique conclut cette étude générale.

Nous avons proposé un environnement d'aide à la parallélisation des applications FTII. Cet environnement orienté objets est non seulement indépendant de l'algorithme particulier considéré, mais aussi de la machine parallèle cible. L'utilisateur peut s'aider de cet environnement pour estimer les performances de son algorithme avant de l'implémenter. De plus, un algorithme de décision fourni par l'environnement FTII permet à l'utilisateur de choisir la stratégie d'équilibre de charge dynamique la mieux adaptée en fonction de l'application considérée et de la machine cible.

Pour conclure ce travail, nous avons étudié un exemple concret d'application FTII. Nous avons montré que le lancer de rayons est une application FTII et nous avons expliqué pourquoi les tâches définies sont indépendantes et irrégulières. À l'aide de l'environnement FTII, nous avons développé un algorithme parallèle pour une CM-5 et un réseau de stations de travail utilisant PVM. L'analyse expérimentale menée nous a permis de valider l'ensemble de notre travail. En particulier, les résultats expérimentaux sont conformes aux résultats obtenus à l'aide du modèle matriciel. De plus, la solution parallèle proposée à l'aide de l'environnement est efficace. Toutes les stratégies distribuées proposées dans le

cadre FTII sont plus efficaces qu'une répartition statique.

Les perspectives de ce travail sont nombreuses. À court terme, nous envisageons l'extension de la notion FTII afin de permettre de considérer les problèmes de satisfaction de contraintes[2] qui ont été étudiés par P.-P. Mérel. En effet, les problèmes CSP sont résolus par des algorithmes pour lesquels il est possible de définir un ensemble de tâches indépendantes et imprévisibles. Cependant, dans le cadre de la thèse de P.-P. Mérel, une tâche est divisible. C'est pourquoi, nous devons étendre la définition de FTII pour prendre en compte la possibilité qu'une tâche puisse être divisée.

Il est souhaitable que le modèle matriciel soit étendu afin de prendre en compte les messages échangés entre deux processeurs pour permettre le calcul d'une tâche (échange de données). En effet, quand un processeur ne dispose pas d'une donnée dans sa mémoire locale qui lui est nécessaire pour terminer le calcul d'un tâche, il est dans l'obligation d'interroger le processeur distant qui possède cette donnée. À l'heure actuelle, ces communications sont ignorées par le modèle matriciel : nous supposons qu'un processeur peut accéder à toutes les données sans attendre.

Une autre perspective intéressante de ce travail est de permettre à l'utilisateur de choisir le réseau d'interconnexion dont dispose la machine parallèle cible au niveau de l'environnement FTII. Si l'environnement prend en compte cette information, les résultats de l'analyse matricielle qui étudient les mérites des différentes stratégies d'équilibre de charge seront probablement différents suivant le réseau d'interconnexion de la machine parallèle étudiée. Le choix da la stratégie proposée par l'algorithme de décision sera donc encore plus affiné qu'auparavant.

Une perspective à plus long terme de ce travail est d'élaborer un modèle stratifié permettant d'inclure différents modèles de programmation (et non plus le seul modèle MIMD), différents types d'application et les outils de mise en œuvre associés (stratégies d'équilibre de charge, *monitoring*,...). Une couche sémantique sera associée au modèle pour permettre de spécifier, dans des modèles abstraits, le parallélisme inhérent à l'application. Une aide à la parallélisation permettra d'obtenir un premier algorithme parallèle conforme aux attentes exprimées dans le modèle général. L'utilisateur pourra modifier cette première solution parallèle afin de prendre en compte des propriétés (liées à l'application ou à la machine parallèle par exemple) qui sont difficiles à exprimer dans ce modèle général.

2. *CSP : Constraint Satisfaction Problem.*

Bibliographie

[AFR+ 94] G. AUTHIÉ, A. FERREIRA, J-L. ROCH, G. VILLARD, J. ROMAN, C. ROUCAIROL, et B. VIROT. *Algorithmes parallèles, analyse et conception.* Hermès, 1994.

[AG 91] I. AHMAD et A. GHAFOOR. « Semi-Distributed Load Balancing For Massively Parallel Multicomputer Systems ». *IEEE Transactions on Software Engineering*, 17(10):987–1004, octobre 1991.

[AGF+ 95] G. AUTHIÉ, J.-M. GARCIA, A. FERREIRA, J-L. ROCH, G. VILLARD, J. ROMAN, C. ROUCAIROL, et B. VIROT. *Parallélisme et applications irrégulières.* Hermès, 1995.

[AIÖ 94] C. AYKANAT, V. ISLER, et B. ÖZGÜÇ. « Efficient parallel spatial subdivision algorithm for object-based parallel ray tracing ». *CAD*, 26(12):883–890, décembre 1994.

[AL 92] S. AKL et L. LINDON. *Modèles de calcul parallèle à mémoire partagée.* Algorithmique Parrallèle. Cosnard et al. eds. Masson, 1992.

[All 78] A. O. ALLEN. *Probability, statistics, and queueing theory with computer science applications.* Academic Press, 1978.

[App 68] A. APPEL. « Some techniques for shading machine renderings of solids ». Dans *SJCC*, 1968, pages 37–45.

[AW 87] J. AMANATIDES et A. WOO. « A Fast Voxel Traversal Algorithm for Ray Tracing ». Dans *Proceedings of EUROGRAPHICS 87*, New York, 1987. Elsevier Science Publ., pages 65–77.

[Bar 89] A. BARAK. « Mosix: An integrated multiprocessor unix ». Dans *Winter USENIX Conference*, San Diego, 1989. pages 101–112.

[BBLP 89] D. BADOUEL, K. BOUATOUCH, Z. LAHJOMRI, et T. PRIOL. « KOAN: un outil général pour paralléliser des algorithmes de synthèse d'images réalistes ». Rapport technique 598, IRISA Rennes, juillet 1989.

[BBPA 94] D. BADOUEL, K. BOUATOUCH, T. PRIOL, et B. ARNALDI. « Distributing Data and Control for Ray Tracing in Parallel ». *IEEE Computer Graphics and Applications*, pages 69–77, juillet 1994.

[Ber 89] F. BERMAN. « Experience with an automatic solution to the mapping problem ». Dans *Proc. of the 22th annual Hawaii International Conference on System Sciences*, 1989.

[BF 81] R. M. BRYANT et R. A. FINKEL. « A Stable Distributed Scheduling Algorithm ». Dans *Proc of 2nd International Conference on Distributed Computing Systems*, Huntsville, Alabama, USA, 1981. IEEE Comput. Soc. Press, pages 314–22.

[BMPA 87] K. BOUATOUCH, M. O. MADANI, T. PRIOL, et B. ARNALDI. « A New Algorithm of Space Tracing Using a CSG Model ». Dans *proceedings of EUROGRAPHICS 87*, New York, 1987. Elsevier Science Publ., pages 65–77.

[Boo 94] G. BOOCH. *Analyse & conception orientées objets, 2e edition.* Addison Wesley, France, 1994.

[Bou 93] L. BOUGÉ. « Le modèle de programmation à parallélisme de données : une perpective sémantique ». *Techniques et science informatiques*, 12(5):541–562, 1993.

[Bre 74] R. P. BRENT. « The parallel evolution of general arithmetic expressions ». *J. ACM*, 21(2):201–206, 1974.

[BS 85] A. BARAK et A. SHILOH. « A Distributed Load-Balancing Policy for a Multicomputer ». Dans *Software Pratice and Experience*, 1985, pages 901–913.

[CG 94] S. R. CHOWDHURY et B. GUPTA. « A Probalistic Dynamic Load Balancing Algorithm For Homogeneous Distributed Systems (With Extension to Hypercubes) ». Dans *22 th Annual ACM Computer Science Conference Proc.*, Phoenix, 1994. pages 165–172.

[CK 84] T. L. CASAVANT et J. G. KUHL. « Design of a Loosely-Coupled Distributed Multiprocessing Network ». Dans *Proceedings of the 1984 International Conference On Parallel Processing*, 1984, pages 42–45.

[CK 88] T. L. CASAVANT et J. G. KUHL. « A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems ». *IEEE Transactions on Software Engineering*, 14(2):141–154, février 1988.

[CLZ 92] A. CORRADI, L. LETIZIA, et F. ZAMBONELLI. « Load balancing strategies for massively parallel architectures ». *Parallel Processing Letters*, 2(2/3):139–148, septembre 1992.

[CNR 92] M. COSNARD, M. NIVAT, et Y. ROBERT. *Algorithmique parallèle.* Masson, 1992.

[CT 93] M. COSNARD et D. TRYSTRAM. *Algorithmes et architectures parallèles.* InterEditions, 1993.

[Cyb 89] G. CYBENKO. « Dynamic Load Balancing for Distributed Memory Multiprocessors ». *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[Dew 85] A. DEWDNEY. « Récréations informatiques : un microscope informatique pour observer l'objet le plus complexe jamais défini par les mathématiciens ». *Pour la science*, (26):87–93, octobre 1985.

[DFM 94] J.-L. DEKEYSER, C. FONLUPT, et P. MARQUET. « Équilibre de charge sur machine SIMD ». *Calculateurs parallèles*, 6(4):45–50, 1994.

[DFvG 83] E. W. DIJKSTRA, W. H. J. FEIGEN, et A. J. M. van GESTEREN. « Derivation of a termination detection algorithm for distributed computation ». *I. P. L.*, 16(3):217–219, juillet 1983.

[Dip 84] M. DIPPÉ. « An adaptative Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis ». *ACM Computer Graphics*, 18(3):149–157, 1984.

[EB 93] D. J. EVANS et W. U. N. BUTT. « Dynamic load balancing using task-transfer probabilities ». *Parallel Computing*, 19:897–916, 1993.

[Eck 94a] J. ECKSTEIN. « Control Strategies for Parallel Mixed Integer Branch and Bound ». Dans *Proc. Supercomputing'94*, Washington, DC, USA, 1994. IEEE Comput. Soc. Press, pages 41–48.

[Eck 94b] J. ECKSTEIN. « Parallel Branch-and-Bound for Mixed Integer Programming ». *SIAM News*, 27:12–15, 1994.

[ELZ 86a] D. L. EAGER, E. D. LAZOWSKA, et J. ZAHORJAN. « Adaptative Load Sharing in Homogeneous Distributed Systems ». *IEEE transactions on software engineering*, SE-12(5):662–675, mai 1986.

[ELZ 86b] D. L. EAGER, E. D. LAZOWSKA, et J. ZAHORJAN. « A Comparaison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing (Extended Abstract) ». Dans *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Austin, TX, août 1986. pages 1–3.

[ELZ 86c] D. L. EAGER, E. D. LAZOWSKA, et J. ZAHORJAN. « A Comparaison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing ». *ACM Performance Evaluation*, 6:53–68, 1986.

[FG 96] M. A. FRANKLIN et V. GOVINDAN. « A general matrix iterative model for dynamic load balancing ». *Parallel computing*, 22(7):969–989, octobre 1996.

[FIR 93] S. FRANK, H. Burkhardt III, et J. ROTHNIE. « The KSR1 : Bridging the Gap between Shared Memory and MPPs ». Dans *Proc. Compcon'93*. IEEE Comput. Soc. Press, février 1993, pages 285–294.

[Flo 67] R. W. FLOYD. « Assigning meanings to programs ». Dans *Proc. of the Symposium on Applied Mathematics.* American Mathematical Society, 1967, volume 19, pages 19–32.

[Fly 66] M. J. FLYNN. « Very high-speed computing systems ». *Proceedings IEEE*, 54(12):1901–1909, 1966.

[Fon 94] C. FONLUPT. « *Distribution dynamique de données sur machines SIMD* ». Thèse de doctorat, Université des sciences et technologies de Lille, décembre 1994.

[FTI 86] A. FUJIMOTO, T. TANAKA, et K. IWATA. « ARTS : Accelerated Ray-Tracing System ». *IEEE Computer graphics and applications*, 6(4):16–26, avril 1986.

[FvDF+ 95] J. D. FOLEY, A. van DAM, S. K. FEINER, J. F. HUGHES, et R. L. PHILLIPS. *Introduction à l'inforgraphie.* Addison Wesley, France, 1995.

[FYN 88] D. FERGUSON, Y. YEMINI, et C. NIKOLAOU. « Microeconomic Algorithms for Load Balancing in Distributed Computer Systems ». Dans *IEEE 8th Int. Conf. on Distributed Computing Systems*, San José, California, 1988. Computer Society Press, pages 491–499.

[GBD+ 94] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, et V. SUNDERAM. « PVM 3 User's guide and reference manual ». Rapport technique, Oak Ridge National Laboratory, septembre 1994.

[GJ 79] M. GAREY et D. JOHNSON. *Computers and intractability : A guide to the theory of NP-completness.* W. H. Freeman, New York, 1979.

[GK 94] E. GELENBE et R. KUSHWAKA. « Dynamic Load Balancing in Distributed Systems ». Dans *MASCOTS'94: proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Durham, North Carolina, 1994. IEEE Comput. Soc. Press, pages 245–249.

[GL 92] F. GLOVER et M. LAGUNA. *Modern Heuristic Techniques for Combinatorial Problems.* W. H. Freeman, New York, 1992.

[Gla 84] A. S. GLASSNER. « Space subdivision for fast ray tracing ». *IEEE Computer graphics and applications*, pages 15–22, octobre 1984.

[Gla 89] A. S. GLASSNER. *An Introduction to Ray Tracing.* Academic Press, 1989.

[GP 89] S. A. GREEN et D. J. PADDON. « Exploiting Coherence for Multiprocessor Ray Tracing ». *IEEE Computer Graphics and Applications*, 9(6):12–26, novembre 1989.

[GPL 89] S. A. GREEN, D. J. PADDON, et E. LEWIS. « A Parallel Algorithm and Tree-Based Computer Architecture for Ray-Traced Computer Graphics ». *Parallel Processing for Computer Vision and Display*, pages 431–442, 1989.

[GS 93] H. GUYENNET et F. SPIES. « Étude comparative de différents algorithmes de répartition de charge dans les systèmes distribués ». *La lettre du transputer et des calculateurs parallèles*, pages 31–55, juin 1993.

[GSS 92] H. GUYENNET, E. SANCHIS, et F. SPIES. « Load balancing in Interconnected Transputer Networks ». Dans *PACTA'92 International Conference on Parallel Computing and Transputers Application*, Barcelona, Spain, septembre 1992.

[Hai 87] E. HAINES. « A Proposal for Standard Graphics Environments ». *IEEE Computer Graphics and Applications*, 7(11):3–5, novembre 1987.

[HG 86] E. HAINES et D. P. GREENBERG. « The light buffer: a shadow testing accelerator ». *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[Hil 85] W. D. HILLIS. *The Connection Machine*. MIT Press, 1985.

[HLM+ 90] S. H. HOSSEINI, B. LITOW, M. MALKAWI, J. MCPHERSON, et K. VAIRAVAN. « Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm ». *Journal of Parallel and Distributed Computing*, 10:160–166, 1990.

[Hoa 69] C. A. R. HOARE. « An axiomatic basis for computer programming ». *Comm. of ACM*, 12(10), octobre 1969.

[Hoa 78] C. A. R. HOARE. « Communicationg Sequential Processes ». *Comm. of ACM*, 21(8):666–676, août 1978.

[KGV 94] V. KUMAR, A. Y. GRAMA, et N. R. VEMPATY. « Scalable Load Balancing Techniques for Parallel Computers ». *Journal of Parallel and Distributed Computing*, 22(1):60–79, juillet 1994.

[KH 95] M. J. KEATES et R. J. HUBBOLD. « Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer ». *Computer graphics forum*, 14(4):189–202, octobre 1995.

[KHH 95] M. KRAJECKI, F. HERRMANN, et Z. HABBAS. « Politiques d'équilibre de charge appliquées à l'algorithme parallèle de lancer de rayons ». Dans *Journées de Recherche sur le Placement Dynamique et la Répartition de Charge*, Paris, 1995. pages 95–98.

[KHHG 96a] M. KRAJECKI, Z. HABBAS, F. HERRMANN, et Y. GARDAN. « Distributed Load Balancing Strategies for Parallel Ray-tracing ». Dans K. YETONGNON et S. HARIRI, éd., *9th International Conference on Parallel and Distributed Computing Systems*, Dijon (France), septembre 1996. ISCA, pages 50–55.

[KHHG 96b] M. KRAJECKI, Z. HABBAS, F. HERRMANN, et Y. GARDAN. « Équilibre de charge pour le lancer de rayons: étude théorique et expérimentale ». *Calculateurs parallèles*, 8(1):69–87, 1996.

[KHHG 98] M. KRAJECKI, Z. HABBAS, F. HERRMANN, et Y. GARDAN. « A performance prediction tool for parallel ray tracing ». Dans *6th International Conference in Central Europe on Computer Graphics and Visualisation (WSCG'98)*, Plzen-Bory (République Tchèque), février 1998.

[Kho 92] J. W. KHO. « Performance issues for Message-Passing MIMD machines ». Dans *Proc. of the 5 th SIAM conference on Parallel Processing for Scientific Computing*, SIAM Philadelphia, 1992. pages 501–506.

[KKHN 93] H. KOBAYASHI, H. KUBOTA, S. HORIGUCHI, et T. NAKAMURA. « Load Balancing Based on Load Coherence between Continuous Images for an Object-Space Parallel Ray-Tracing System ». *IEICE Transactions on Information and Systems*, V:1490–1499, 1993.

[KNK+ 88] H. KOBAYASHI, S. NISHIMURA, H. KUBOTA, T. NAKAMURA, et Y. SHIGEI. « Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision ». *The Visual Computer*, 4:197–209, 1988.

[Kra 95] M. KRAJECKI. « *Étude de l'équilibre de charge: application au lancer de rayons* ». Mémoire de DEA, Université Henri Poincaré Nancy I, septembre 1995.

[Kra 97] M. KRAJECKI. « Une classification des stratégies d'équilibre de charge dynamique : application au lancer de rayons parallèle ». Rapport technique 97-02, Laboratoire de Recherche en Informatique de Metz (LRIM), Université de Metz, mai 1997.

[Kra 98a] M. KRAJECKI. « Un algorithme d'équilibre de charge dynamique semi-distribué pour les applications à nombre Fini de Tâches Indépendantes et Irrégulières ». Dans *Deuxièmes Journées de Recherche sur le Placement Dynamique et la Répartition de Charge (JRPRC2)*, Lille, mai 1998.

[Kra 98b] M. KRAJECKI. « Un environnement d'aide à la parallélisation pour les applications à nombre Fini de Tâches Indépendantes et Irrégulières ». Dans *Renpar'10*, Strasbourg, juin 1998.

[Lam 94] L. LAMPORT. « The Temporal Logic of Actions ». *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, mai 1994.

[Lam 95] L. LAMPORT. « TLA+ ». Digital Equipment Corporation, juillet 1995.

[Lan 94] Y. LANUEL. « *Les arbres VSG: une nouvelle approche multimodèles de la visualisation* ». Thèse de doctorat, Université de Metz, février 1994.

[Lav 95] C. LAVAULT. *Complexité des algorithmes distribués.* Hermès, 1995.

[Lee 91] C.-L. LEE. « Parallel machines scheduling with nonsimultaneous machine available time ». *Discrete Applied Mathematics*, 1991.

[Lef 93] W. LEFER. « Parallel Strategies for the Ray Tracing ». Dans *IEEE Visualization '93 Parallel Symposium*, San Jose (USA), octobre 1993.

[Lei 85] C. LEISERSON. « Fat-Trees : Universal Networks for Hardware-Efficient Supercomputing ». *IEEE Transaction on Computers*, 34(10):892–901, octobre 1985.

[Li 86] K. LI. « *Shared Virtual Memory on Loosely Coupled Multiprocessors* ». Thèse de doctorat, Yale University, septembre 1986.

[LK 86] F. C. H LIN et R. M. KELLER. « Gradient Model: A Demand-Driven Load Balancing Scheme ». Dans *proc. 6th international Conference on Distributed Computing systems*, Cambridge, septembre 1986. pages 329–336.

[LM 82] M. LIVNY et M. MELMAN. « Load Balancing in Homogeneous Broadcast Distributed Systems ». Dans *proc. ACM Computer Network Performance Symposium*, avril 1982, pages 47–55.

[LM 93] R. LÜLING et B. MONIEN. « Load balancing for distributed branch & bound ». Dans *Third Mons Workshop on Parallel Computing*, septembre 1993.

[LMR 91] R. LÜLING, B. MONIEN, et F. RAMME. « Load Balancing in Large Networks: A Comparative Study (Extended Abstract) ». Dans *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, décembre 1991. pages 686–689.

[LRN 95] T.-Y. LEE, C. S. RAGHAVENDRA, et J. B. NICHOLAS. « Parallel Implementation of Ray-tracing Algorithm on the Intel Delta Parallel Computer ». Dans *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, California, avril 1995. IEEE Computer Society Press, pages 688–692.

[Luc 90] M. LUCAS. « Techniques de parallèlisation en synthèse d'images ». Dans *MICAD*, 1990.

[Luc 97] A. LUCCHESE. « Implémentation d'une bibliothèque de communication pour la portabilité des algorithmes de charge dynamique ». Rapport technique, LRIM, Université de Metz, juillet 1997.

[LYJ 96] A. L. LAW, R. YAGEL, et D. N. JAYASIMHA. « Parallel Volume Rendering for Scientific Visualization ». *International Journal of Computers and their Applications*, 3(3):138–148, décembre 1996.

[McC 93] W. F. MCCOLL. « General Purpose Parallel Computing ». Dans A. M. GIBBONS et P. SPIRAKIS, éd., *Proc. 1991 ALCOM Spring School on Parallel Computation.* Cambridge Internatinonal Series on Parallel Computation, 1993, volume 4, pages 261–336.

[McC 95] W. F. MCCOLL. « Scalable Computing ». Dans J. van LEEUWEN, éd., *Computer Science Today: Recent Trends and Developments.* Springer-Verlag, 1995, numéro 1000 dans Lecture Notes in Computer Science, pages 46–61.

[MHHS 96] P.-P. MÉREL, Z. HABBAS, F. HERRMANN, et D. SINGER. « N-ary Consistencies and Constraint-based Backtracking ». Dans *Proceedings of the second International Conference on Principles and Practice of Constraint Programming*, Cambridge, MA, 1996. extended abstract.

[Mil 83] R. MILNER. « A calculus of communicating system ». *Lecture Notes in Computer Science*, (92), 1983.

[Mon 74] U. MONTANARI. « Networks of Constraints: Fundamental Properties and Applications to Pictures Processing ». *Information Sciences*, 7:95–132, 1974.

[MZ 95] F. J. MUNIZ et E. J. ZALUSKA. « Parallel load-balancing: an extension to the gradient model ». *Parallel computing*, 21(2):287–301, février 1995.

[Nic 95] C. NICOLAS. « *Modèles de facettisation de carreaux restreints* ». Mémoire de DEA, Université Henri Poincaré Nancy I, septembre 1995.

[NO 86] K. NEMOTO et T. OMASHI. « An adaptive subdivision by sliding boundary surfaces for fast ray tracing ». Dans *Proc. Graphics Interface'86*, 1986, pages 43–48.

[NS 81] D. NASSIMI et S. SAHNI. « Data Broadcasting in SIMD Computers ». *IEEE Transactions on computers*, c-30(2):101–106, février 1981.

[NXG 85] L. M. NI, C. XU, et T. GENDREAU. « A Distributed Drafting Algorithm for Load balancing ». *IEEE transactions on Software Engineering*, SE(11):1153–1161, 1985.

[PAGM 88] B. PEROCHE, J. ARGENCE, D. GHAZANFARPOUR, et D. MICHELUCCI. *La syntèse d'iamges*. Editions Hermes, 1988.

[PB 89] T. PRIOL et K. BOUATOUCH. « Ray Tracing on parallel computers: a performance study ». Rapport technique 451, IRISA Rennes, janvier 1989.

[PB 90] S. PATIL et P. BANERJEE. « A parallel branch and bound algorithm for test generation ». *Computer Aided Design*, 1990.

[Per 95] E. PERRIN. « *Une nouvelle approche pour les opérations booléennes : formalisation et mise en œuvre* ». Thèse de doctorat, Université de Metz, octobre 1995.

[Pet 77] J. L. PETERSON. « Petri Nets ». *Computing Surveys*, 9(3):223–253, septembre 1977.

[Pri 89] T. PRIOL. « *Lancer de rayon sur des architectures parallèles: Etude et mise en œuvre* ». Thèse de doctorat, Université de Rennes I, juin 1989.

[Pui 97] R. PUIGJANER. « Modèles d'évaluation de performances : des réseaux de file d'attente à la simulation ». Dans *2e Ecole d'Informatique des Systèmes Parallèles et Répartis*. IRIT (Institut de Recherche En Informatique de Toulouse), mars 1997, pages 1–14.

[Req 80] A. A. G. REQUICHA. « Representation for rigid solids : theory, methods and systems ». *ACM Computing Surveys*, 12(4):437–464, décembre 1980.

[Rot 82] S. D. ROTH. « Ray casting for modeling solids ». *Computer Graphics and Image Processing*, 18(2):109–144, février 1982.

[Rou 95] C. ROUCAIROL. « On Irregular Data Structures and Asynchronous Parallel Branch and Bound Algorithms ». *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:323–336, 1995.

[Sap 90] G. SAPORTA. *Probabilités, analyse des données et statistique.* Technip, Paris, 1990.

[SC 89] K. G. SHIN et Y. C. CHANG. « Load Sharing in Distributed Real-Time Systems With State-Change Broadcasts ». *IEEE Transactions on Computers*, 38(8):1124–1142, août 1989.

[SGS 95] C. N. SEKHARAN, V. GOEL, et R. SRIDHAR. « Load balancing methods for ray tracing and binary tree computing using PVM ». *Parallel Computing*, 21:1963–1978, 1995.

[Sha 93] A. U. SHANKAR. « An Introduction to Assertional Reasoning for Concurrent Systems ». *ACM Computing Surveys*, 25(9):225–262, septembre 1993.

[Smi 80] R. G. SMITH. « The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver ». *IEEE Transactions on Computers*, C-29(12):1104–1113, décembre 1980.

[Spi 94] F. SPIES. « *Conception et évaluation de stratégies de répartition de charge dynamique dans les systèmes distribués* ». Thèse de doctorat, Université de Franche-Comté, décembre 1994.

[SS 84] J. A. STANKOVIC et I. S. SIDHU. « An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups ». Dans *IEEE 4^{th} Int. Conf. on Distributed Computing Systems*, 1984, pages 49–59.

[Thi 93a] THINKING MACHINES CORPORATION. *CMMD Reference Manual V 3.0.* mai 1993.

[Thi 93b] THINKING MACHINES CORPORATION. *Connection Machine CM-5, Technical Summary.* novembre 1993.

[Val 90a] L. G VALIANT. « A Bridging Model for Parallel Computation ». *Communications of the ACM*, 33(8):103–112, août 1990.

[Val 90b] L. G. VALIANT. General Purpose Parallel Architectures. Dans J. van LEEUWEN, éd., *Handbook of theorical computer science*, volume A, algorithms and complexity, Chapitre 18, pages 942–971. Elsevier Science Publishers B.V., 1990.

[Var 62] R. S. VARGA. *Matrix Iterative Analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1962.

[Viv 93] R. VIVIAN. « *Définition d'une approche adaptative : application à la visualisation en CAO* ». Thèse de doctorat, Université de Metz, janvier 1993.

[Whi 80] T. WHITTED. « An improved illumination model for shaded display ». Dans *Commun. ACM*, juin 1980, volume 23, pages 343–349.

[Whi 94] S. WHITMAN. « Dynamic load balancing for parallel polygon rendering ». *IEEE Computer graphics and applications*, pages 41–48, juillet 1994.

[WM 85] Y. T. WANG et R. J. MORRIS. « Load Sharing in Distributed Systems ». *IEEE transactions on Computers*, C(34):202–217, mars 1985.

[WR 89] M. WILLEBEEK-LEMAIR et A. P. REEVES. « A Distributed Dynamic Load Balancing Strategie for Highly Parallel Multicomputer Systems ». Dans *proc. of the 4th SIAM Conference on Parallel Processing for Scientific Computing*, 1989, pages 351–356.

[WR 93] M. WILLEBEEK-LEMAIR et A. P. REEVES. « Strategies for Dynamic Load Balancing on Highly Parallel Computers ». *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, septembre 1993.

[XL 92] C.-Z. XU et F. C. M. LAU. « Analysis of the Generalized Dimension Exchange method for Dynamic Load Balancing ». *Journal of Parallel and Distributed Computing*, 16(4):385–393, décembre 1992.

[XML 95] C. XU, B. MONIEN, et R. LÜLING. « An analytic comparison of nearest neighbor algorithms for load balancing in parallel computers ». Dans *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, California, avril 1995. IEEE Computer Society Press, pages 472–479.

[Zho 88] S. ZHOU. « A Trace-Driven Simulation Study of Dynamic Load Balancing ». *IEEE Transactions on Software Engineering*, 14(9):1327–1341, 1988.

RÉSUMÉ

Le parallélisme permet l'utilisation simultanée de plusieurs processeurs pour résoudre plus rapidement un problème. Cependant, multiplier la puissance de la machine par le nombre de processeurs est un idéal qui est en général très difficile à atteindre, car si la répartition des tâches n'est pas optimisée, les performances du programme seront réduites. Le placement de ces tâches est un des problèmes principaux du parallélisme.
Dans ce travail, nous étudions un environnement d'aide à la parallélisation pour les applications FTII (applications à nombre Fini de Tâches Indépendantes et Irrégulières). Cet environnement propose une première solution parallèle qui résout dynamiquement le problème d'équilibre de charge. L'environnement FTII met à la disposition de l'utilisateur différents outils dont : cinq algorithmes MIMD d'équilibre de charge dynamique, un modèle mathématique basé sur les matrices pour valider les algorithmes théoriquement et un environnement de programmation sur une machine parallèle virtuelle.
Pour illustrer l'environnement FTII, nous considérons le lancer de rayons comme étude de cas. Nous avons montré le caractère irrégulier du lancer de rayons. En particulier, nous avons mis en évidence la dépendance existante entre le comportement de l'application et la scène à visualiser. À l'aide de l'environnement FTII, nous avons parallélisé le lancer de rayons et une analyse expérimentale sur CM-5 a été menée. L'environnement FTII nous a permis de paralléliser efficacement le lancer de rayons sans se soucier de la machine cible ni de la stratégie d'équilibre de charge dynamique appliquée.

Mots clés : parallélisme, algorithmique parallèle et distribuée, équilibre de charge, lancer de rayons.

ABSTRACT

Parallelism allows the use of several processors simultaneously to solve a given problem more quickly. However, to increase the power of the machine by the number of processors is an ideal which is in general very difficult to reach, because if the task allocation is not optimized, the performances of the program will be reduced. The placement of these tasks is one of the principal problems of parallelism.
In this work, we study a toolkit helping the parallelization of the FIIT applications (applications with a Finite number of Independent and Irregular Tasks). The result of this toolkit is a first parallel solution which dynamically solves the problem of load balancing. The FIIT toolkit proposes various tools to the user such as: five MIMD load balancing strategies, a mathematical model based on matrix to validate the algorithms theoretically and an environment of programming on a virtual parallel machine.
To illustrate the FIIT toolkit, we consider the ray tracing as case study. We showed the irregular character of the ray tracing. In particular, we highlighted that the behaviour of the application is strongly dependent on the scene to visualize. Using the FIIT environment, we parallelized it and an experimental analysis on a CM-5 was carried out. The FIIT toolkit enabled us to parallelize the ray tracing application efficiently without worrying about the target machine nor the dynamic load balancing strategy applied.

Key words: parallelism, distributed and parallel algorithms, load balancing, ray tracing.