Titre: Title:	Évolutivité et performance de systèmes de fichiers sur grappes d'ordinateurs
Auteur: Author:	Vu Anh Nguyen
Date:	2001
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Nguyen, V. A. (2001). Évolutivité et performance de systèmes de fichiers sur grappes d'ordinateurs [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/6977/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/6977/
Directeurs de recherche: Advisors:	Samuel Pierre
Programme: Program:	Génie électrique

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

ÉVOLUTIVITÉ ET PERFORMANCE DE SYSTÈMES DE FICHIERS SUR GRAPPES D'ORDINATEURS

VU ANH NGUYEN
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
GÉNIE ÉLECTRIQUE
FÉVRIER 2001

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ÉVOLUTIVITÉ ET PERFORMANCE DE SYSTÈMES DE FICHIERS SUR GRAPPES D'ORDINATEURS

Présenté par NGUYEN Vu Anh

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

A été dûment accepté par le jury d'examen composé de :

M. François GUILBAULT, Ph. D., président

M. Samuel PIERRE, Ph. D., directeur de recherche

M. Robert ROY, Ph. D., membre

REMERCIEMENTS

Je tiens à remercier Samuel PIERRE, mon directeur de maîtrise, pour ses conseils et son appui constant, ainsi que les employés d'Ericsson Canada qui ont collaboré à ma recherche pour l'aide qu'ils m'ont apportée et leurs commentaires.

Enfin, merci aux membres du LARIM qui ont partagé ma vie d'étudiant et ainsi participé à l'aboutissement de ma recherche.

Je remercie également ma famille et mes amis pour leur soutien inconditionnel et leurs encouragements.

RÉSUMÉ

Les télécommunications à grande échelle dont l'Internet nécessitent des capacités de traitement importantes. Les machines parallèles sont depuis longtemps considérées comme une solution à ce problème. Mais, avec l'accroissement des performances du matériel grand public, de nouvelles architectures parallèles peuvent être utilisées ; les grappes d'ordinateurs constituent une solution peu coûteuse qui possède aussi l'avantage d'être évolutive. Ainsi, il est théoriquement facile d'augmenter la puissance d'une grappe d'ordinateurs en lui rajoutant des nœuds. En pratique, de nombreux problèmes peuvent dégrader la performance du système résultant et compromettre cette évolutivité. Le but de ce mémoire est donc d'étudier l'évolutivité des grappes d'ordinateurs afin de proposer un système dont l'évolutivité est proche du cas idéal.

La plupart des recherches qui ont été menées sur les grappes d'ordinateurs ont pour cible des applications scientifiques. Il existe ainsi une théorie de l'évolutivité des systèmes parallèles, et des métriques comme l'iso-vitesse ou l'iso-efficacité permettent de mesurer l'évolutivité. Ces premières recherches s'intéressaient davantage à l'aspect algorithmique de l'évolutivité, alors que les recherches plus récentes se sont penchées sur l'architecture des systèmes parallèles afin de les rendre évolutifs. Dans ce cadre, le stockage et l'accès aux données jouent un rôle central et constituent souvent une limite à l'évolutivité. Dans ce mémoire, nous avons donc choisi d'étudier plus particulièrement les systèmes de fichiers répartis pour grappes d'ordinateurs. Nous avons commencé par une étude générique des grappes d'ordinateurs à l'aide d'un modèle analytique. Pour cela, nous avons utilisé une modélisation par réseau de Pétri et nous avons calculé les indices de performances de notre modèle. Nous avons ainsi montré que la façon dont les performances du système évoluent dépend de la capacité des ressources partagées et des délais de transaction. Mais, ce sont les contentions lors des accès simultanés à une ressource partagée qui bornent les performances ou les dégradent quand le nombre de nœuds de la grappe augmente.

Pour la suite, nous avons choisi le système de fichiers « Parallel Virtual File System » (PVFS) développé par l'Université de Clemson comme point de départ. Ce système de fichiers pour grappes d'ordinateurs Linux implémente le « wide stripping ». Nous avons implémenté et calibré un simulateur de PVFS grâce auquel nous avons montré que cette technique de distribution des données rend les performances du système évolutives tant que la taille de la grappe reste inférieure à un certain seuil qui correspond à la saturation de la bande passante du réseau. Pour palier cette limite, nous proposons d'exploiter les avantages du « wide striping » et de la réplication dans la même architecture. Notre nouvelle technique de distribution des données basée sur le « chained declustering » a pour but d'assurer à la fois la tolérance aux pannes et l'évolutivité du système. Elle permet également de faire évoluer le système à moindre coût et sans interruption ni dégradation trop importante du service déjà offert. De plus, la granularité de l'évolutivité est réduite à un nœud, ce qui permet d'ajuster la taille de la grappe aux besoins en capacité de traitement. Enfin, nous proposons une architecture basée sur le concept de grappes de grappes qui permet de s'affranchir de la limite en taille que le réseau impose.

Pour valider notre système de fichiers, nous avons utilisé le simulateur de PVFS dans lequel nous avons implémenté les améliorations proposées. Les résultats des simulations montrent que l'évolutivité de notre algorithme statique de lecture des données est proche du cas idéal. Une fois que l'on a bien choisi la taille de la grappe d'origine, les performances ne sont plus limitées par le réseau et sont proches des performances idéales. Nous avons également simulé un scénario de mise à jour du système sans interruption de service pour calculer la perturbation engendrée sur le service : celle-ci est minime et peut être contrôlée par des mécanismes de gestion des priorités.

ABSTRACT

Great scale telecommunications and the Internet require more and more processing capacity. Parallel computers have been used for a long time to solve this problem. Today, "off the shelf" components achieve good performances and they can be used to build new parallel systems such as computer clusters which are less expensive than the other architectures and can achieve better scalability. Thus, it is theoretically easy to increase the overall performances of a computer cluster by adding a node. Actually, there are numerous factors than can limit the performances or even lower the scalability. The purpose of this thesis is to study the scalability of computer clusters in order to propose improvements to existing systems and achieve real scalability.

Most of the researches on parallel systems are related to scientific applications. Thus a complete scalability theory and metrics like iso-speed or iso-efficiency have already been defined. But these first researches are more focused on the algorithmic aspect of scalability. Recent researches are more interested in the architecture of scalable parallel systems. In this context, data storage and data access are crucial and often limit the scalability. In this thesis, we choose to study more particularly distributed files system for computer clusters. We start with a general study of parallel systems with an analytical model. This model is based on Petri Nets and we have showed that the performance indices depend on the capacity of shared resources and transaction delay. But this is only the contention due to multiple accesses to shared resources that limits the overall performances and could even decrease the performances while the number of nodes is increasing.

For this study, we have chosen to work with Parallel Virtual File System (PVFS) from Clemson University; it is a distributed file system for Linux that implements wide stripping. First, we have programmed and calibrated a PVFS simulator; using this simulator, we demonstrated that PVFS has a very good scalability where the number of nodes in the cluster is smaller than a given threshold. This threshold actually

corresponds to the saturation of the network bandwidth. To go beyond this limit, we propose to use wide striping and replication in the same file system. We also propose a new data distribution technique based upon "chained declustering", that warranties high availability and scalability. This also allows the system to be improved with minimal cost, without service interruption and with a minimal degradation of service. In addition, the granularity of the file system is ideal: the size of the cluster can be adjusted to the needed performances with a precision of one node. Finally, we propose a complete architecture, using cluster of clusters, where the performances are not limited by the network performances.

In order to validate our file system, we use the PVFS simulator where the improvements have been implemented. The results show that the performances of the system are close to the ideal case. Once the size of the origin cluster is well defined, the total number of nodes in the system is not limited anymore, and the performances increase linearly. We have also simulated an upgrade of the system, in order to measure the perturbation caused by the update of the new nodes: it is minimal and can be controlled by priority mechanisms.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES SIGLES ET ABRÉVRIATIONS

CPU Processeur central d'un ordinateur (Central Processing Unit).

FFT Transformée de Fourier Rapide (Fast Fourrier Transform).

FIFO Politique premier arrivé, premier servi d'une file d'attente.

GSPN Classe des réseaux de Pétri stochastiques généralisés (Generalised Stochastic Petri Net).

PDD Algorithme parallèle de la diagonale dominante (résolution d'un système tri-diagonal).

PPT Algorithme parallèle de partitionnement (résolution d'un système tri-diagonal).

PVFS Parallel Virtual File System.

RAID Redundant Array of Inexpensive Disks.

Reduced PDD Algorithme parallèle réduit de la diagonale dominante (résolution d'un système tri-diagonal).

VoD Vidéo sur Demande (Video on Demand).

CHAPITRE 1
INTRODUCTION

Les applications commerciales ou scientifiques nécessitent de plus en plus de capacité de traitement. Pour répondre à ce besoin, les systèmes parallèles sont aujourd'hui largement utilisés. Il existe plusieurs types d'architectures parallèles parmi lesquelles les grappes d'ordinateurs connaissent un succès croissant. En effet, ils ont un rapport coût/performance très faible car ils sont souvent composés de matériel « grand public ». Mais un système parallèle doit aussi être évolutif : il faut pouvoir améliorer ses performances en augmentant les ressources matérielles. De par leur architecture, les grappes d'ordinateurs supportent bien l'ajout de nouvelles ressources matérielles. Toutefois, cela ne suffit pas pour garantir l'évolutivité des performances : c'est ce que nous allons étudier dans ce mémoire. Après avoir présenté les définitions et concepts de base, ce chapitre d'introduction définit notre problématique de recherche, puis énonce nos objectifs de recherche avant de finir en présentant le plan du mémoire.

1.1 Définitions et concepts de base

Le nom « grappe d'ordinateurs » (*computer cluster* en anglais) est une métaphore qui représente bien l'architecture de ce système formé d'ordinateurs personnels (PC) ou de stations de travail ordinaires qui constituent les nœuds de la grappe. Les nœuds sont interconnectés par un réseau local et coopèrent entre eux : les grappes d'ordinateurs sont donc un type d'architecture parallèle. Le matériel utilisé est en général du type grand

public (PC reliés par un réseau Ethernet) : comparés aux autres architectures parallèles, le rapport prix/performance est donc très faible.

L'évolutivité d'une grappe d'ordinateurs réfère à sa capacité à tirer partie d'une augmentation des ressources disponibles pour accroître ses performances globales. Par exemple, une grappe d'ordinateurs dont les performances doublent quand on multiplie par deux le nombre de nœuds (le matériel utilisé étant homogène) possède un évolutivité parfaite. D'autre part, il faut considérer la granularité de l'évolutivité. En effet, l'évolutivité d'un système peut être bonne, mais pour certaines valeurs du nombre de nœuds seulement, c'est-à-dire que les performances augmentent par paliers successifs.

Les coûts de parallélisme correspondent aux coûts des traitements supplémentaires lorsqu'on utilise un système parallèle plutôt que séquentiel. Ce sont par exemple les temps de communications entre les différents nœuds de la grappe d'ordinateurs ou les temps de calcul nécessaires à la répartition des tâches entre les nœuds.

1.2 Éléments de la problématique

L'un des points forts des grappes d'ordinateurs est de permettre très facilement l'ajout de nœuds à un système déjà existant. Mais l'évolutivité d'un système ne se limite pas à ce simple fait : il ne suffit pas d'augmenter la puissance théorique d'un système, il faut aussi pouvoir l'exploiter. Dans un cas extrême, l'augmentation du nombre de nœuds dans la grappe d'ordinateurs peut même dégrader les performances globales du système. Dans le cas idéal, les performances de la grappe d'ordinateurs augmentent linéairement avec le nombre de nœuds. Étudier l'évolutivité d'une grappe d'ordinateurs, c'est donc déterminer jusqu'à quel point les performances de celle-ci augmentent en fonction du nombre de nœuds. Pour se rapprocher du cas idéal, il faut bien répartir les tâches sur l'ensemble de la grappe tout en minimisant les coûts de parallélisme. Il faut aussi éviter tout goulot d'étranglement dans l'architecture de la grappe qui limitera les performances à cause des congestions.

Pour des applications scientifiques, les performances d'un système sont mesurées par le temps mis pour résoudre un problème de taille donnée. Dans le cadre d'une utilisation commerciale, c'est la capacité de service du système qui nous intéresse. On mesure ainsi le nombre de clients pouvant être servis simultanément, tout en garantissant à chaque client un certain niveau de qualité de service. Dans ce contexte, la grappe d'ordinateurs doit également garantir la continuité du service et la tolérance aux pannes. Une mise à jour du système devra perturber le moins possible le service déjà fourni : l'évolutivité du système devra se faire à un coût raisonnable. C'est pourquoi les reconfigurations nécessaires lors de l'ajout d'un nœud à la grappe doivent être minimales. Nous avons déjà mentionné que les grappes d'ordinateurs sont intéressantes car leur rapport coût/performance est faible : c'est une caractéristique qui reste valable lorsqu'on fait évoluer le système. Si le choix du matériel utilisé est restreint, ce sont donc l'architecture, les mécanismes et les algorithmes utilisés qui doivent répondre aux contraintes. Également pour des raisons économiques, un système dont la granularité de l'évolutivité est faible sera meilleur : on pourra ainsi ajuster plus précisément la taille de la grappe d'ordinateurs en fonction des besoins en performance. Étant donné l'évolution rapide du matériel et de son prix, cela peut représenter des économies substantielles.

Les systèmes de fichiers pour grappes d'ordinateurs ont fait l'objet de relativement peu de recherches. Pourtant, c'est une composante cruciale qui limite souvent les performances des grappes. Les contraintes d'évolutivité, de continuité de service et de tolérance aux pannes sont particulièrement importantes. En effet, les applications commerciales servent un client en accédant à des données : quand le nombre de clients augmente, la capacité du système de fichiers doit s'accroître. C'est donc la bande passante maximale du système qui détermine les performances du système de fichiers.

1.3 Objectifs de recherche

Cette recherche a pour but principal d'étudier l'évolutivité des systèmes parallèles afin de proposer des améliorations aux grappes d'ordinateurs existantes. Elle

vise plus spécifiquement à proposer un système de fichiers réparti pour grappe d'ordinateurs en :

- étudiant analytiquement et par des simulations l'évolutivité d'un système de fichiers de notre choix et qui se base sur les choix techniques retenus lors de notre étude préalable ;
- proposant des algorithmes et des mécanismes pour l'accès aux données qui améliorent l'évolutivité du système tout en répondant aux contraintes que nous nous sommes fixés ;
- énonçant aussi les procédures à respecter pour faire évoluer le système progressivement ;
- implémentant et calibrant un simulateur de système de fichiers qui nous permet d'évaluer les améliorations apportées par le système de fichier que nous proposons.

1.4 Plan du mémoire

Le deuxième chapitre fait une recension sélective des recherches sur l'évolutivité ainsi que les solutions techniques utilisées dans les systèmes de fichiers répartis. Le chapitre trois propose un modèle générique de système parallèle et son étude analytique. Le chapitre quatre présente les résultats obtenus en simulant notre modèle analytique. Ces simulations ont permis d'étendre le modèle pour le rendre fidèle au système de fichiers réparti que nous avons choisi pour notre recherche. Dans la suite de ce chapitre, nous proposons une amélioration basée sur une nouvelle distribution des données, une description des algorithmes utilisés, ainsi que les résultats obtenus avec notre simulateur. Enfin, le chapitre cinq conclut ce mémoire en synthétisant les travaux réalisés et les principaux résultats de notre recherche; il présente les limitations de ces travaux et formule des recommandations pour des recherches futures.

CHAPITRE 2
ASPECTS FONDAMENTAUX DU CONCEPT D'ÉVOLUTIVITÉ

Les grappes d'ordinateurs connaissent un succès important car il est théoriquement facile d'en améliorer les performances en augmentant le nombre de nœuds qu'elles intègrent. Un des principaux avantages des systèmes répartis est que leur taille peut augmenter de manière flexible [Buyya99]. Ainsi, l'étude de l'évolutivité (*scalability* en anglais) permettra de choisir la meilleure association architecture/algorithme pour une situation donnée, mais aussi d'extrapoler et de prévoir le comportement d'un système parallèle quand le nombre de processeurs augmente, à partir de données recueillies pour un nombre réduit de processeurs. Ce chapitre a pour but de faire le point sur les différents aspects du concept d'évolutivité. Dans un premier temps, nous allons donner les définitions et concepts de base nécessaires à l'étude de l'évolutivité. Puis, nous aborderons les métriques qui ont été développées pour mesurer celle-ci, ainsi que les problèmes qui y sont liés.

2.1 Définitions et concepts de base

On définit un *système parallèle* comme l'association d'une architecture parallèle avec un algorithme parallèle. Dans tout ce chapitre et sauf mention contraire, on considèrera que l'architecture parallèle utilisée est homogène, c'est-à-dire que les

processeurs et les canaux de communication sont tous identiques. La *taille d'un problème (W)* est une mesure du nombre d'opérations de base nécessaire pour résoudre ce problème. Étant donné qu'il peut exister plusieurs algorithmes pour résoudre le même problème, on choisira l'algorithme séquentiel le plus rapide pour assurer l'unicité de la définition. La taille du problème ne prend en compte que les calculs utiles à la résolution de ce problème.

L'évolutivité d'un système parallèle peut se définir comme une mesure de l'aptitude à utiliser efficacement un nombre croissant de processeurs [Kumar94]. Ainsi, lorsqu'un problème est résolu avec un système parallèle, il est naturel de s'attendre à ce que le temps d'exécution du même problème diminue quand on augmente les ressources de calcul disponibles. On peut aussi reformuler la définition de l'évolutivité en se basant sur le cas idéal illustré à la Figure 2.1 : l'évolutivité d'une architecture parallèle se définit alors comme la mesure de sa capacité à accélérer les calculs de façon proportionnelle aux nombres de processeurs utilisés [Grama93].

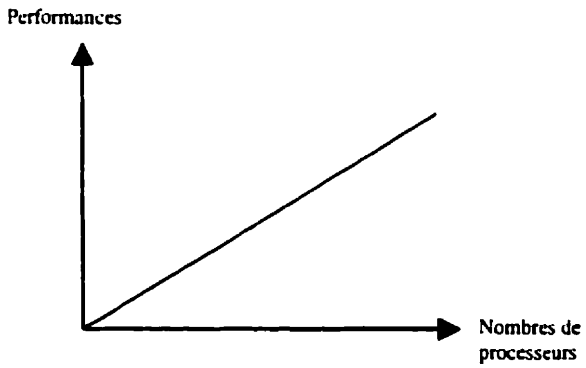

Figure 2.1 Cas idéal d'évolutivité

L'évolutivité peut aussi être appréhendée à travers les objectifs que l'on cherche à atteindre. Son analyse permet de répondre aux questions suivantes : de quelle manière l'augmentation du nombre de processeurs influence les performances d'un algorithme ? Dans quelle mesure la taille du problème affecte-t-elle les performances ? De quelle façon une modification de la vitesse de calcul des processeurs, de la vitesse du réseau

d'interconnexion et d'autres composants matériels affectent les performances d'un système parallèle [Grama93] ?

Keqin Li et Xiang He Sun [Sun98] définissent l'évolutivité d'un système parallèle comme une mesure du coût des traitements supplémentaires liés au parallélisme, alors que la taille du système et la taille du problème augmentent.

Pour un algorithme donné, sa *fraction séquentielle (s)* est le rapport du temps d'exécution de sa composante séquentielle au temps total d'exécution sur un processeur. La composante séquentielle d'un algorithme est la partie de l'algorithme qui ne peut être parallélisée et qui doit être exécutée sur un même processeur.

Le *temps d'exécution séquentiel* (T_S) fait référence au temps d'exécution de l'algorithme séquentiel le plus rapide pour un problème donné. On le compare au *temps d'exécution parallèle* (T_P) qui correspond au temps écoulé entre le début de l'exécution parallèle et le moment ou le dernier processeur utilisé complète ses calculs. Pour un système parallèle donné, T_P est une fonction de la taille du problème (W) et du nombre de processeurs (p), et on écrira parfois $T_P(W,p)$. Le ratio de ces deux valeurs T_P/T_S est appelé *accélération parallèle (S)*: c'est le gain en temps d'exécution lorsqu'on utilise une architecture parallèle plutôt qu'un seul processeur.

Le *coût* d'un système parallèle est le produit du temps d'exécution parallèle par le nombre de processeurs. Un système parallèle est dit de coût optimal si et seulement si le coût est asymptotiquement du même ordre de grandeur que le temps d'exécution séquentiel, c'est à dire $p.T_P = \Theta(W)$.

Le *coût supplémentaire du parallélisme* (T_O) est défini comme la somme de tous les surcoûts liés au parallélisme pour l'ensemble des processeurs. Il comprend les surcoûts de communication, de calculs supplémentaires, d'attente, de synchronisation de l'algorithme. L'expression mathématique est la suivante : $T_O = p.T_P - T_S$. Pour simplifier notre étude théorique, nous considérerons que T_O n'est jamais négatif, ce qui signifie que l'accélération parallèle est toujours bornée par p. Dans certains cas, l'accélération peut être « superlinéaire » et T_O prendra alors des valeurs négatives. C'est par exemple possible si la mémoire des processeurs est hiérarchique, ce qui a pour conséquence que

les temps d'accès à la mémoire augmente avec la quantité de mémoire utilisée. Ainsi, si l'algorithme utilise une quantité M de mémoire sur un processeur, il utilisera une quantité M/p de mémoire sur une architecture parallèle à p processeurs. La prise en compte de ce cas particulier n'a pas d'importance fondamentale dans l'étude théorique de l'évolutivité et la définition des métriques qui y sont liées, car on ne veut pas s'intéresser à une architecture particulière dans ce chapitre. Pour un système parallèle, T_O est une fonction de W et de p et on écrira parfois : $T_O(W,p)$.

L'efficacité (E) d'un système parallèle est définie comme le rapport de l'accélération parallèle (S) au nombre de processeurs (p) :

$$E = \frac{S}{p} = \frac{Ts}{p.Tp} = \frac{1}{1 + \frac{To}{Ts}} \qquad (2.1)$$

Le *degré de simultanéité* $\Gamma(W)$ désigne le nombre maximum de tâches qui peuvent être exécutées simultanément à tout moment pour un algorithme parallèle. Ainsi, pour un W donné, l'algorithme ne pourra pas utiliser plus de $\Gamma(W)$ processeurs. Ce paramètre est indépendant de l'architecture du système et il a été démontré que, si l'algorithme est à coût optimal (i.e. $p.T_P = \Theta(W)$), alors $\Gamma(W) < \Theta(W)$ [Kumar94].

La loi d'Amdhal

Dès 1967, Amdahl [Amdahl67] s'est intéressé aux systèmes parallèles : il a observé que l'accélération parallèle (S) est bornée supérieurement par l'inverse de la fraction séquentielle de l'algorithme (s) : $S \leq 1/s$. Cette limite est indépendante de l'architecture et a été un argument principal des détracteurs des systèmes parallèles de grande taille. Mais, l'accélération parallèle dépend de nombreux autres facteurs. En pratique, la loi d'Amdahl prédit que, dans le cas général, pour la résolution de l'instance d'un problème donné (c'est à dire de taille fixée), l'accélération parallèle ne croît pas linéairement avec le nombre de processeurs. Elle tend plutôt à atteindre une valeur limite. En d'autres termes, cela signifie que l'efficacité chute lorsque le nombre de processeurs augmente.

2.2 La fonction iso-efficacité

La fonction iso-efficacité est l'une des principales métriques d'évolutivité. Dans cette section, nous allons en donner la définition, étudier ses propriétés et l'utiliser sur un exemple d'application.

2.2.1 Définitions de base

La fonction appelée *iso-efficacité* mise au point par Grama, Gupta et Kumar [Grama93] est en fait une métrique de l'évolutivité pour les algorithmes et les architectures parallèles. Sur un ordinateur séquentiel, le meilleur algorithme pour résoudre un problème donné est le plus rapide. Mais, pour un système parallèle, cela devient beaucoup plus complexe : le temps mis par un algorithme parallèle pour résoudre un problème est fonction de la taille du problème, du nombre de processeurs utilisé mais aussi des caractéristiques de l'architecture parallèle (puissance des processeurs, type d'interconnexions, politique de routage...). C'est pourquoi cette métrique a pour objectif de permettre une meilleure analyse des performances d'un algorithme et d'une architecture parallèle.

Cette métrique se base sur la loi d'Amdahl et le constat suivant : pour un même nombre de processeurs et un même problème, une instance de plus grande taille augmente l'efficacité du système parallèle et accroît l'accélération parallèle. Ainsi, il est théoriquement possible de maintenir l'efficacité constante en augmentant à la fois le nombre de processeurs et la taille du problème : ce comportement se retrouve sur la plupart des systèmes parallèles. Grama, Gupta et Kumar définissent théoriquement l'évolutivité comme la possibilité de maintenir l'efficacité constante quand le nombre de processeurs croît en augmentant la taille du problème.

Mais, si un système parallèle est évolutif selon cette définition théorique, un autre problème se pose : à quel taux faut-il augmenter la taille du problème pour remplir la condition d'évolutivité ? C'est ce taux qui mesure le degré d'évolutivité du système. Si le coût pour exécuter une opération de base est t_c, l'équation (2.2) devient :

$$E = \frac{1}{1 + \frac{To}{t_c.W}} \qquad (2.2)$$

Dans cette équation, on se rend bien compte que si W est maintenu constant alors que p augmente, l'efficacité diminue parce que le surcoût total de parallélisme T_O croît avec p. Au contraire, si W augmente en maintenant p constant, l'efficacité augmente car T_O croît moins vite que $\Theta(W)$. Par définition, un système est évolutif si on peut maintenir l'efficacité à une certaine valeur, c'est à dire si le rapport T_O/W dans l'équation (2.2) reste constant. En considérant que E est constant dans (2.2), on peut écrire :

$$\frac{T_0}{W} = t_c\left(\frac{1-E}{E}\right)$$

$$W = \frac{1}{t_c}\left(\frac{E}{1-E}\right) \quad T_0$$

Posons $K = E/(t_c(1\text{-}E))$. L'équation devient alors :

$$W = K.T_O \qquad (2.3)$$

À partir de l'équation (2.3), on peut obtenir W comme une fonction du nombre de processeurs p. C'est cette fonction que l'on appelle *iso-efficacité d'un système parallèle*. Si la fonction iso-efficacité a des valeurs petites, cela signifie que le système parallèle possède une très bonne évolutivité, car lorsqu'on augmente le nombre de processeurs, l'augmentation de la taille du problème nécessaire pour maintenir l'efficacité constante est faible. Inversement, si la fonction iso-efficacité est grande, le système possède une mauvaise évolutivité. La fonction n'est pas définie pour les systèmes non évolutifs (selon la définition théorique).

2.2.2 Exemple d'iso-efficacité [Grama93]

Considérons un système parallèle qui possède la propriété suivante :

$$T_O = p^{3/2} + p^{3/4}W^{3/4}.$$

En remplaçant dans l'équation (2.3), on obtient :

$$W = Kp^{3/2} + Kp^{3/4}W^{3/4} \qquad (2.4)$$

Il est difficile d'exprimer W en fonction de p à partir de cette équation. Ignorons d'abord le deuxième terme dans T_0. L'équation (2.3) devient alors :

$$W = Kp^{3/2} \tag{2.5}$$

Si au contraire, on ignore le premier terme, on obtient :

$$W = Kp^{3/4}W^{3/4} \tag{2.6}$$

$$W = K^4p^3 \tag{2.7}$$

Si on veut pouvoir maintenir l'efficacité constante quand le nombre de processeurs croît, le premier et le second terme de T_0 imposent que la taille du problème augmente en $\Theta(p^{3/2})$ et $\Theta(p^3)$ respectivement. Le comportement asymptotique de ce système parallèle est donc décrit par une fonction iso-efficacité en $\Theta(p^3)$.

En effectuant cette analyse, nous pouvons tester les performances d'un système avec un petit nombre de processeurs puis en prévoir le comportement à grande échelle. Mais l'étude de l'iso-efficacité ne se limite pas à cet objectif : dans les prochains paragraphes, nous verrons que cette métrique permet aussi d'étudier le comportement de systèmes parallèles en fonction de paramètres tels que la vitesse des processeurs, les capacités des canaux de communication, etc.

2.2.3 Coût optimal et borne inférieure de la fonction iso-efficacité

La définition d'un système parallèle de coût optimal ($pT_P \propto W$) et celle du coût supplémentaire de parallélisme ($T_0 = p.T_P - T_S$) donnent :

$$T_P + T_0 \propto W$$

or :

$$T_P = W.t_c$$

d'où :

$$W t_c + T_0 \propto W$$

enfin :

$$W \propto T_0 \tag{2.8}$$

Un système est donc à coût optimal si le coût supplémentaire de parallélisme et la taille du problème sont proportionnels. C'est exactement la condition exprimée par l'équation (2.3) : la conclusion est que satisfaire l'équation (2.3) d'iso-efficacité quand le nombre de processeurs augmente permet de conserver la propriété du coût optimal d'un système parallèle.

Nous avons conclu précédemment qu'une fonction iso-efficacité plus petite signifie une meilleure évolutivité. Mais quelle est alors la borne inférieure de la fonction ? Si un problème est de taille W, il ne peut pas être exécuté en parallèle sur plus de W processeurs. Supposons que le système parallèle soit idéal et qu'il n'y ait aucun surcoût de parallélisme. Dans ce cas, si la taille du problème croît plus lentement que $\Theta(p)$, il existe un seuil à partir duquel des processeurs sont forcément inutilisés, et à partir de là, l'efficacité chute. Donc, pour maintenir une efficacité constante, il faut que la taille du problème ait une croissance asymptotique supérieure ou égale à $\Theta(p)$. La fonction iso-efficacité d'un système parallèle idéal est donc $\Theta(p)$.

Tel que mentionné précédemment, nous avons vu que le nombre maximum de processeurs sur lequel un problème est exécuté est borné par W. Il est en fait borné par le degré de parallélisme $P(W)$ de l'algorithme utilisé. La conséquence est que la fonction d'iso-efficacité ne peut être de coût optimal que si le degré de parallélisme est de l'ordre de W. Si cela n'est pas le cas, la fonction iso-efficacité est plus grande et l'évolutivité du système est moins bonne.

2.2.4 Exemple d'étude de l'évolutivité avec la fonction iso-efficacité [Grama93]

La métrique et la théorie développées autour de la fonction iso-efficacité ont été mises en place vers la fin des années 1980. Elles ont d'abord été utilisées dans l'étude des algorithmes de calcul parallèle. Par la suite, des recherches ont aussi été menées sur des systèmes parallèles moins spécialisés dans le calcul scientifique. Nous allons ici aborder l'exemple d'une étude sur la répartition dynamique des tâches.

Considérons l'application possédant les caractéristiques suivantes :

- Les tâches présentes sur chacun des processeurs peuvent être partitionnées en sous-tâches indépendantes de base (non décomposables) ;
- Il est difficile d'estimer la quantité de calcul nécessaire pour une sous-tâche ;
- Il existe un mécanisme « raisonnable » de répartition des tâches, c'est à dire qui vérifie la propriété suivante : si une tâche ω localisée sur un processeur est divisée en deux taches $\psi\omega$ et $(1-\psi)\omega$, alors il existe une constante $\alpha > 0$ petite tel que $\psi\omega > \alpha\omega$ et $(1-\psi)\omega > \alpha\omega$. Le rôle de la constante α est de fixer une borne inférieure du déséquilibre conséquent au partitionnement des tâches.

Pour cet exemple, nous allons utiliser l'algorithme de répartition des tâches suivant : toutes les tâches se trouvent initialement sur un seul processeur. Il existe une variable globale G qui pointe initialement sur le premier processeur : G est un pointeur modulo le nombre de processeurs (cette technique porte le nom de « Global Round Robin »). Un processeur libre P_i choisit le processeur P_G pointé par G et lui demande de lui donner du travail. Si le processeur pointé par G n'a pas de travail à partager, il rejette la requête. Après chaque requête, réussie ou non, G est incrémenté de un.

Il est clair que, pour un tel algorithme non déterministe, il est impossible de déterminer exactement les temps d'exécution. En revanche, on peut déterminer une borne supérieure du surcoût dû aux communications entre les processeurs [Grama93] : le nombre de communications pour cet algorithme est majoré par $O(p \log W)$, chaque communication dure $O(\log p)$. Donc, le surcoût total dû aux communications entre processeurs est majoré par $O(p \log p \log W)$. Pour garantir l'iso-efficacité, il faut donc : $W = O(p \log p . \log W)$. Après manipulation de cette expression, on obtient la condition suivante :

$$W = O(p \log^2 p)$$

Mais cette équation prend uniquement en compte le surcoût des communications. Un autre surcoût est engendré par le partage de la variable globale G dont les accès sont répétés. Il se pose donc le problème des accès concurrents car un seul processeur peut accéder à G à un instant donné. Analysons l'iso-efficacité due aux accès concurrents.

La variable globale G est accédée $O(p \log W)$ fois pendant toute l'exécution. Si les processeurs sont efficacement utilisés, le temps total d'exécution est de $\Theta(W/p)$. Supposons qu'il n'y ait pas de problème d'accès concurrent pendant toute l'exécution. Dans ce cas, W/p est très grand par rapport au temps durant lequel G est accédée. Si le nombre de processeurs augmente, le temps d'exécution (i.e. W/p) diminue, mais le nombre d'accès à G augmente. Ainsi, il existe un seuil à partir duquel les accès concurrents à G deviennent un goulot d'étranglement et le temps d'exécution ne pourra plus être réduit en augmentant le nombre de processeurs p. On peut éviter cette limite en faisant augmenter W à un taux tel que le rapport entre W/p et $O(p \log W)$ reste le même. En résolvant ce problème avec les équations de l'iso-efficacité, il a été démontré que l'iso-efficacité est de la forme $O(p^2 \log p)$. La conclusion est que la partie de l'iso-efficacité due à l'accès concurrent domine celle relative aux communications. Finalement, l'iso-efficacité du système est donnée par $O(p^2 \log p)$.

Dans cet exemple, nous avons vu comment il est possible d'étudier un modèle de surcoût d'un système grâce à la fonction iso-efficacité et ainsi de déterminer les facteurs qui interviennent dans l'évolutivité. Dans [Grama93], il a été confirmé expérimentalement que les schémas de distribution dynamique des tâches possédant une meilleure fonction d'iso-efficacité sont plus efficaces en terme d'évolutivité que ceux qui ont une mauvaise fonction iso-efficacité.

La métrique de l'iso-efficacité est utile dans des situations où l'on veut obtenir des performances qui augmentent linéairement avec le nombre de processeurs [Kumar94] : il faut que la taille du problème augmente conformément à la fonction iso-efficacité pour que l'accélération parallèle soit linéaire. Mais si un système peut être évolutif en théorie, dans la pratique, la taille du problème est limitée par la mémoire disponible à chaque processeur par exemple. Un système évolutif en théorie

peut ne plus l'être dans la pratique. C'est pourquoi, la fonction iso-efficacité est aussi utile pour mesurer le degré d'évolutivité d'un système parallèle.

2.3 Autre métrique de l'évolutivité : l'iso-vitesse

Dans cette section, nous abordons un autre moyen de mesurer l'évolutivité, proposé par Sun et Rover [Sun91] au début des années 1990 : la métrique de l'*iso-vitesse* a été développée pour mesurer l'évolutivité d'un système en étudiant les vitesses d'exécution, car c'est finalement le facteur intéressant dans la pratique. En effet, l'étude de l'évolutivité a d'abord été plutôt théorique car le but était, dans un premier temps, d'étudier les systèmes parallèles en tant que tel en les comparant à des systèmes séquentiels. Comme l'iso-efficacité, cette métrique propose une mesure de l'évolutivité d'un système parallèle et a fait l'objet de nombreuses études d'algorithmes et de systèmes parallèles. Dans la section 2.1, nous avons abordé la théorie liée à cette métrique. Dans ce qui suit, nous définirons le concept d'iso-vitesse et nous énoncerons les théorèmes et les corollaires qui seront ensuite vérifiés par l'étude d'algorithmes et l'expérimentation.

2.3.1 Définition de l'iso-vitesse

La vitesse d'un calcul parallèle est une fonction de la taille du problème W et du nombre de processeurs p. Elle est définie comme le rapport de la taille du problème au temps d'exécution parallèle :

$$V(W, p) = \frac{W}{T_P(W, p)}$$

La *vitesse moyenne* se calcule en divisant la vitesse par le nombre de processeurs :

$$\overline{V}(W, p) = \frac{V(W, p)}{p} = \frac{W}{p.T_P(W, p)}$$

Comme pour l'iso-efficacité, un système est dit évolutif au sens de l'iso-vitesse s'il est possible de maintenir la vitesse moyenne constante lorsque le nombre de

processeurs et la taille du problème augmentent. Ainsi, quand le nombre de processeurs augmente de p à p' et la taille du problème de W à W', l'évolutivité iso-vitesse d'un système parallèle, est évaluée par la fonction :

$$\Phi(p,p') = \left(\frac{W}{p}\right) \Big/ \left(\frac{W'}{p'}\right) = \frac{p'.W}{p.W'}$$

De la même manière que l'iso-efficacité, cette fonction indique dans quelle mesure la taille du problème doit augmenter quand le nombre de processeurs croît pour que la vitesse moyenne $\overline{V}(W,p)$ reste constante. Cela revient à trouver la fonction :

$$W = f(p)$$

tel que :

$$\overline{V}(W, p) = \frac{f(p)}{p.T_P(f(p), p)} = C$$

À défaut de déterminer exactement cette fonction, on peut rechercher le taux de croissance de $W = f(p)$ nécessaire pour maintenir la vitesse moyenne $\overline{V}(W,p)$ constante. Une faible croissance de $f(p)$ signifie une bonne évolutivité, alors qu'une forte croissance indique une mauvaise évolutivité : dans le cas d'une forte croissance, il faut augmenter la taille du problème de beaucoup – relativement à l'augmentation du nombre de processeurs – pour maintenir la vitesse moyenne constante.

En choisissant $W = f(p)$, la vitesse moyenne est constante donc :

$$\frac{W}{p.T_P(W, p)} = \frac{W'}{p'.T_P(W', p')}$$

$$\Phi(p,p') = \frac{T_P(W, p)}{T_P(W', p')}$$

En conclusion, la métrique de l'iso-vitesse permet d'évaluer la dégradation de performance d'un calcul à plus grande échelle (plus de processeurs et un problème de taille plus grande). Dans le cas idéal, la condition d'iso-vitesse est respectée sans qu'il soit nécessaire de changer la taille du problème. On a alors une évolutivité égale à un :

$$\Phi(p,p') = 1 \quad \Leftrightarrow \quad \Phi(p,p') = \left(\frac{W}{p}\right) \Big/ \left(\frac{W'}{p'}\right) = \frac{p'.W}{p.W'}$$

2.3.2 Théorèmes et corollaires

Le temps d'exécution est une mesure fondamentale pour les calculs parallèles [Sun95]. Les théorèmes 1 et 2 qui suivent montrent que la métrique de l'iso-vitesse favorise les systèmes qui ont un meilleur temps d'exécution.

Théorème 1 : Si les temps d'exécution des systèmes parallèles 1 et 2 sont respectivement aT et T pour un même état initial (même taille de problème W), alors le système 1 a une plus grande évolutivité que le système 2 si et seulement si les temps d'exécution pour un problème W' sont tels que $aT' < T'$.

Ce théorème soulève donc un problème simple : les performances d'un système peuvent être très bonnes pour une taille de problème donné, mais se détériorer et rendre le système pire qu'un autre pour d'autres tailles de problème. Dans le cas où les temps d'exécution initiaux des deux systèmes sont identiques, ou lorsque les deux systèmes ont la même évolutivité, on a les corollaires suivants :

Corollaire 1 : Si les temps d'exécution des systèmes parallèles 1 et 2 sont identiques pour une taille de problème W, le système 1 a une meilleure évolutivité que le système 2 si et seulement si le temps d'exécution d'un problème de taille supérieure sur le système 1 est plus petit que sur le système 2.

Corollaire 2 : Si les temps d'exécution des systèmes parallèles 1 et 2 sont respectivement aT et T pour une taille de problème W, les deux systèmes sont de même évolutivité si et seulement si, pour un problème de taille supérieure, on obtient le même rapport des temps d'exécution (aT' et T').

Les performances d'un système parallèle peuvent être évaluées par le temps d'exécution - comme c'est le cas dans le théorème n°1 - ou bien par la taille du problème nécessaire pour atteindre une vitesse d'exécution donnée. C'est le point de vue du théorème n°2 qui suit.

Théorème 2 : Si pour un même temps d'exécution, les tailles des problèmes sur les systèmes parallèles 1 et 2 sont respectivement αW et W, alors l'évolutivité du système 1 multiplié par α est plus grande que l'évolutivité du système 2 si et seulement si le temps d'exécution d'un problème de plus grande taille est inférieure sur le système 1.

Corollaire 3 : Si pour un même temps d'exécution, les tailles des problèmes sur les systèmes parallèles 1 et 2 sont respectivement αW et W, alors l'évolutivité du système 1 multipliée par α est égale à l'évolutivité du système 2 si et seulement si les temps d'exécution d'un problème de plus grande taille sont identiques sur les deux systèmes.

(Pour les démonstrations des théorèmes et des corollaires, voir [Sun95]).

2.3.3 Étude de la résolution de systèmes tri-diagonaux

La résolution des systèmes tri-diagonaux est un problème clef du calcul scientifique. Par exemple, beaucoup de méthodes de résolution d'équations différentielles partielles s'appuient sur la résolution de systèmes tri-diagonaux. Pour illustrer les résultats théoriques de l'iso-vitesse, Xian-He Sun [Sun95] compare trois algorithmes de résolution des systèmes tri-diagonaux.

Par définition, un système tri-diagonal est un système linéaire d'équations de la forme :

$$Ax = d$$

où $\quad x = (x_1, x_2, \ldots, x_n)^T$

$$d = (d_1, d_2, \ldots, d_n)^T$$

et A une matrice carrée tri-diagonale de dimension n :

$$A=\begin{bmatrix} b_0 & c_0 & & & & & & & \\ a_0 & b_1 & c_1 & & & & & & \\ & a_1 & \ddots & \ddots & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & \ddots & \ddots & \ddots & & \\ & & & & & \ddots & \ddots & c_{n-2} & \\ & & & & & & a_{n-2} & b_{n-2} & c_{n-1} \\ & & & & & & & a_{n-1} & b_{n-1} \end{bmatrix}$$

Les trois algorithmes de résolution d'un système tri-diagonal étudiés par Xian-He Sun sont les suivants :

- algorithme parallèle de partitionnement (**PPT**, Parallel Partition Algorithm) ;
- algorithme parallèle de la diagonale dominante (**PDD**, Parallel Diagonal Dominant Algorithm) ;
- algorithme parallèle réduit de la diagonale dominante (**Reduced PDD**, Reduced Parallel Diagonal Dominant Algorithm).

L'étude théorique montre que les algorithmes *PDD* et *PDD réduit* ont une évolutivité parfaite de 1 [Sun95]. Par contre, l'algorithme *PPT* n'est pas parfaitement évolutif et son évolutivité dépend de l'architecture du système parallèle. Ces différences sont dues à la structure des communications qui diffère d'un algorithme à l'autre. Ce résultat a été vérifié par l'expérience. Les résultats expérimentaux obtenus valident également les théorèmes et les corollaires énoncés plus hauts. Par exemple, les algorithmes *PDD* et *PDD réduit* ont théoriquement la même évolutivité : cela a permis d'appliquer le corollaire 2.

2.3.4 Synthèse

Alors que l'accélération parallèle permet de mesurer le gain entre un système séquentiel et un système parallèle, l'évolutivité évalue le gain d'un système parallèle en

fonction de sa taille. L'évolutivité mesure la capacité d'une combinaison algorithme/architecture à maintenir l'utilisation de chaque processeur [Kumar94].

L'évolutivité a fait l'objet de nombreuses études théoriques, mais le temps d'exécution reste le critère fondamental d'évolution des systèmes parallèles. L'étude de l'évolutivité aurait un intérêt pratique limité si elle ne donnait pas d'information importante sur les temps d'exécution d'un système parallèle [Sun95]. La métrique de l'iso-vitesse est basée sur cette idée. De plus, l'avantage de l'iso-vitesse est qu'une fois que la vitesse initiale a été choisie, la vitesse moyenne est indépendante de la taille du problème, du nombre de processeurs et de toute référence à un système séquentiel.

2.4 Autres métriques de l'évolutivité

Dans cette section, nous présentons quelques autres métriques d'évolutivité qui ont été développées. Elles ne font pas l'objet d'un développement complet comme l'iso-efficacité et l'iso-vitesse, car ce sont des extensions ou des sous-parties de ces deux métriques. Certains concepts sont cependant intéressants et méritent d'être évoqués dans cette revue de littérature.

2.4.1 Développements théoriques

Gustafson, Montry et Berner [Gustafson88, Gustafson88a] ont développé une métrique appelée *accélération d'échelle* (scaled speedup) pour évaluer expérimentalement les performances d'un système parallèle. Cette mesure est, par définition, l'accélération obtenue lorsqu'on augmente la taille du problème linéairement avec le nombre de processeurs. Si la courbe obtenue est bonne, c'est à dire proche de la linéarité, alors le système est dit évolutif. Pour des systèmes ayant une bonne évolutivité, les résultats de cette métrique et ceux de l'iso-efficacité sont très proches. Des généralisations de la notion « d'accélération d'échelle » ont été formulées par la suite : elles diffèrent par la manière dont la taille du problème augmente. Par exemple, dans une des approches, la taille du problème augmente de telle manière que toute la mémoire disponible est utilisée (on suppose que la quantité totale de mémoire augmente avec le

nombre de processeurs). D'autres méthodes prennent en compte le temps d'exécution et se rapprochent beaucoup de l'iso-vitesse.

Pour leurs expérimentations, Karp et Flatt ont défini la notion de « *fraction séquentielle* » f comme une métrique des performances d'un système parallèle. Si S est l'accélération parallèle d'un système à p processeurs, f est définie par :

$$f = \frac{1/S - 1/p}{1 - 1/p}$$

La valeur de f est exactement égale à la fraction séquentielle « s » si le seul facteur limitant l'accélération parallèle (S) est la partie séquentielle de l'algorithme (i.e. il n'y a pas de coût supplémentaire de parallélisme). De petites valeurs de f indiquent de bonnes performances. Si f augmente avec le nombre de processeurs, cela signifie que le surcoût de parallélisme augmente quand p croît : c'est le signe d'une mauvaise évolutivité.

L'équipe de Zorbas [Zorbas89] a développé une métrique de l'évolutivité basée sur l'évaluation des surcoûts de parallélisme. Un algorithme parallèle est décomposé en sa partie séquentielle (W_S) et sa partie parallélisable (W_P). Lorsqu'il est exécuté sur un seul processeur, le temps nécessaire est $t_c.(W_S + W_P)$. Idéalement, le temps d'exécution du même algorithme sur un système parallèle à p processeurs est $t_c.(W_S + \frac{W_P}{p})$. Mais en pratique, il faut prendre en compte les surcoûts de parallélisme qui augmentent le temps d'exécution, ce qui donne : $t_c.(W_S + \frac{W_P}{p}) \times \Phi(p)$. Ici, $\Phi(p)$ est appelée « *fonction surcoût de parallélisme* » : on dit alors par définition, qu'un système parallèle est évolutif avec un surcoût de parallélisme de $\Phi(p)$ si le temps d'exécution vérifie la condition :

$$T_P \leq t_c.(W_S + \frac{W_P}{p}) \times \Phi(p)$$

La plus petite fonction qui satisfait cette contrainte est appelée la *fonction surcoût de parallélisme du système* et est définie par :

$$\Phi(p) = \frac{T_P}{t_c.(W_S + \frac{W_P}{p})}$$

Un système parallèle a une évolutivité idéale si la fonction $\Phi(p)$ reste constante quand le nombre de processeurs augmente. Si la taille du problème augmente au moins aussi vite que la fonction iso-efficacité, $\Phi(p)$ est une constante et le système est idéalement évolutif selon la définition de Zorbas. Dans ce cas, ce critère ne donne aucune information sur le degré d'évolutivité du système. Un autre problème est que la fonction $\Phi(p)$ prend en compte uniquement le surcoût de communication. En revanche, cette métrique est intéressante lorsque le système n'est pas évolutif, car elle fournit des informations sur son degré de non-évolutivité.

Chandran et Davis [Chandran87] définissent la fonction d'efficacité du processeur (FEP) comme la borne supérieure du nombre de processeurs p qui peut être utilisé pour résoudre un problème de taille W, tel que le temps d'exécution sur le système parallèle soit de l'ordre du temps d'exécution sur un système séquentiel divisé par p, i.e. :

$$T_p = \Theta\left(\frac{W}{p}\right)$$

L'inverse de cette fonction est appelé fonction d'efficacité des données (FED) et est définie comme la taille minimale du problème vérifiant la condition ci-dessus lorsque le nombre de processeurs est fixé. Le concept de fonction d'efficacité des données est en fait assez proche de la fonction d'iso-efficacité.

L'équipe de Kruskal [Kruskal88] a travaillé sur une classe de problèmes appelée *problèmes parallèlement efficaces (PE)*. La classe des problèmes PE a la propriété suivante : la fonction iso-efficacité des algorithmes est polynomiale. Ils ont montré l'invariance de cette propriété sur plusieurs architectures parallèles : ainsi, un problème dont la fonction iso-efficacité est polynomiale sur une architecture donnée aura la même propriété sur une autre architecture. Il existe cependant des exceptions : par exemple, l'algorithme de FFT a une iso-efficacité polynomiale sur une topologie en hypercube,

alors qu'elle est exponentielle pour une topologie en grille. La conclusion de cette étude est que l'iso-efficacité varie d'une architecture parallèle à une autre, mais a aussi permis de définir une nouvelle classe de problèmes et de prévoir, dans une certaine mesure, le comportement de combinaisons algorithme/architecture parallèle.

Eager, Zahorajan et Lazowska [Eager89] utilisent le concept de « *parallélisme moyen* » pour caractériser l'évolutivité d'un système logiciel parallèle représenté sous la forme d'un graphe acyclique de tâches, pouvant inclure des règles de précédence. Le parallélisme moyen se définit comme le nombre moyen M de processeurs qui sont occupés pendant l'exécution du programme, en faisant l'hypothèse qu'un nombre infini de processeurs est disponible. Une fois M déterminé, ils ont démontré que l'accélération parallèle et l'efficacité avaient comme borne inférieure $\frac{pM}{(p+M-1)}$ et $\frac{M}{(p+M-1)}$ respectivement. Mais ces résultats sont utiles uniquement lorsqu'on néglige les surcoûts de communication. Dans le cas contraire, les résultats peuvent être bien plus mauvais.

Marinescu et Rice [Marinescu93] pensent qu'un seul paramètre qui dépend uniquement du logiciel utilisé ne suffit pas à analyser l'évolutivité d'un système. Par exemple, l'approche précédente du « parallélisme moyen » ne prend pas en compte les composants matériels qui influencent pourtant les performances du système. Ils ont développé un modèle pour décrire et analyser les calculs parallèles à travers le nombre de « processus légers » (n_P) impliqués et le nombre d'évènements ($g(n_P)$) (un évènement est une action de communication ou de synchronisation). À tout moment, un processus léger peut être soit en train de faire un calcul utile pour l'algorithme, soit en train de communiquer, soit bloqué. L'accélération parallèle peut être vue comme le nombre moyen de processus légers qui effectuent un calcul utile. La conclusion de cette étude est que si $g(n_P)=\Theta(n_P)$ alors l'accélération est asymptotiquement bornée. Si $g(n_P)=\Theta(n_P{}^m), m>1$, alors l'accélération parallèle optimale est obtenue pour une certaine valeur p_{opt} et tend asymptotiquement vers zéro quand le nombre de processeurs

augmente. La valeur de p_{opt} dépend de $g(n_P)$: $g(n_P)$ est donc caractéristique du système et p_{opt} fournit une mesure de l'évolutivité. Plus le nombre de processeurs utilisés dans les conditions optimales est grand, plus le système est évolutif. Lorsque le nombre d'évènements $g(n_P)$ est une fonction convexe de n_P, p_{opt} peut être calculé en résolvant l'équation suivante :

$$p = \frac{\left(\frac{W}{\theta} + g(n_P)\right)}{g'(n_P)}$$

où W est la taille du problème, et θ la durée associée à chaque événement (on la considère constante).

2.4.2 Performance des systèmes parallèles de grande échelle

Dans la pratique, l'accélération parallèle pour un problème de taille constante atteint une limite lorsqu'on augmente le nombre de processeurs : cela est dû au surcoût de parallélisme qui croît plus rapidement que le gain de performance. Nous allons aborder dans cette section quelques-unes des recherches sur ce phénomène.

Flatt et Kennedy [Flatt90, Flatt89] ont calculé les limites de performance d'un système parallèle fixé par les surcoûts de synchronisation et de communication. Quand la fonction surcoût obéit à certaines conditions, il existe une valeur p_0 unique du nombre de processeurs qui minimise le temps d'exécution, pour une taille de problème donnée. Mais cette valeur p_0 correspond à une efficacité très faible : c'est pourquoi, les auteurs proposent plutôt de minimiser le produit de l'efficacité et de l'accélération parallèle. Cela correspond à des contraintes plus réalistes et revient en fait à maximiser l'efficacité divisée par le temps d'exécution. Une des hypothèses importantes de cette étude est que le surcoût par processeur $t_0(W, p) = \frac{T_0(W, p)}{p}$ augmente plus rapidement que $\Theta(p)$, ce qui limite la portée de cette étude. Gupta et Kumar [Gupta93] montrent que plus t_0 est petit et plus la valeur de p_0 est grande (et meilleur est l'algorithme). Pour certains systèmes parallèles, cette valeur de p_0 est même supérieure au degré de parallélisme du système, ce qui rend le calcul de sa valeur inutile.

Flatt et Kennedy ont aussi étudié l'augmentation simultanée de la taille du problème et du nombre de processeurs. Ils définissent *« l'accélération d'échelle »* par :

$$A_e(k) = \frac{k.T_P(W, p)}{T_P(kW, kp)}$$

Ils ont démontré que sous l'hypothèse que t_0 dépend uniquement du nombre de processeurs, A_e est bornée par $\frac{k}{T_P(kW, kp)}$.

L'équipe d'Eager [Eager89] a utilisé le « parallélisme moyen » pour déterminer la position de l'*inflexion* dans le graphe du temps d'exécution en fonction de l'efficacité. L'inflexion a lieu quand le rapport de l'efficacité au temps d'exécution est maximum. Une des applications de cette recherche est de pouvoir partager de façon optimale les ressources de calcul d'un système entre plusieurs applications.

Tang et Li [Tang90] ont prouvé que maximiser le rapport E/T_P revient à minimiser $p(T_P)^2$. Ils proposent donc de minimiser un terme plus général : $p(T_P)^r$. Dans cette expression, r reflète l'importance qui est attribuée au temps d'exécution par rapport à l'efficacité. Une grande valeur de r signifie que le temps d'exécution est privilégié aux dépens de l'efficacité : le système sera ainsi composé de plus de processeurs mais qui ne seront pas exploités au maximum. Inversement, si on veut porter l'accent sur l'efficacité aux dépens du temps d'exécution, on choisira des petites valeurs de r. Kumar et Gupta [Gupta93] déterminent analytiquement le nombre optimal de processeurs qui minimise $p(T_P)^r$, pour une certaine classe de fonctions de surcoût. Ils démontrent alors que, pour un grand nombre de fonctions de surcoût, minimiser ce terme revient asymptotiquement à maintenir l'efficacité constante : cette valeur constante dépend alors de r et de la fonction de surcoût du système.

Zhou et Van Catledge [VanCatledge89, Zhou89] ont raffiné le modèle de Gustafson qui prédit les performances d'un système parallèle à partir de sa fraction séquentielle. Ils ont conclu qu'en augmentant la taille du problème, il est possible d'obtenir une accélération parallèle proche du nombre de processeurs. Mais l'augmentation de la taille du problème nécessaire pour atteindre cette accélération parallèle dépend de la fraction séquentielle de l'algorithme utilisé. Si cette fraction

séquentielle reste constante, il suffit d'augmenter linéairement la taille du problème avec le nombre de processeurs (la métrique de l'iso-efficacité arrive aux mêmes conclusions). Dans le cas où la fraction séquentielle augmente avec la taille du problème, celle-ci doit augmenter plus que linéairement. Cette approche ne prend pas en compte le surcoût de communication en tant que tel. En effet, la composante séquentielle W_S d'un algorithme coûte $W_S(p-1) \approx p.W_S$ et participe au surcoût de parallélisme total. Cela est dû au fait que lorsqu'un processeur est en train d'exécuter une partie de la fraction séquentielle, tous les autres sont inutilisés. Ainsi, la fraction séquentielle peut inclure le surcoût de communication dans le cas où ce dernier augmente linéairement avec la taille du problème. S'il n'est pas en $\Theta(p)$, ce modèle n'est plus adéquat.

2.5 Conception de systèmes parallèles évolutifs

Après s'être intéressé à la métrique de l'évolutivité, nous allons aborder les problèmes liés à la conception d'un système évolutif. Nous allons ainsi étudier les choix d'architectures possibles.

2.5.1 Influence du matériel et de l'architecture

Homogénéité du système

De nombreuses recherches se sont penchées sur l'influence de la vitesse des CPU et des vitesses des canaux de communications sur les performances des systèmes parallèles. Il est clair que si la vitesse des CPU est plus grande, les performances globales ne peuvent qu'être meilleures. Mais contrairement à un système séquentiel, dans un système parallèle, une amélioration d'un facteur k de la performance des CPU n'entraîne pas nécessairement une réduction de k du temps d'exécution.

Dans une étude sur la multiplication de matrices sur des architectures parallèles en grille, Gupta et Kumar [Gupta91] définissent t_w comme le temps nécessaire pour transmettre un « mot de donnée » entre deux processeurs connectés directement, et t_c comme le temps nécessaire à un processeur pour effectuer une « unité de calcul ». Pour la variante GK de l'algorithme DNS [Gupta91], ils obtiennent alors le résultat suivant :

la fonction iso-efficacité est proportionnelle à $\left(\frac{t_w}{t_c}\right)^3$. La conséquence est qu'en augmentant la vitesse des processeurs sans changer les canaux de communication, il faut augmenter la taille du problème d'un facteur 1000 pour maintenir la condition d'iso-efficacité. En revanche, si on augmente la vitesse de communication d'un facteur 10, la même efficacité est obtenue en augmentant la taille du problème de seulement $10\sqrt{10}$. Cet exemple, comme d'autres recherches, prouve que pour optimiser l'utilisation du système, il ne faut pas créer de déséquilibre dans l'architecture. Ainsi, il est peu rentable d'améliorer uniquement une composante matérielle du système. Pour ne pas créer d'incohérence dans le système et maximiser les performances globales, il est indispensable de tenir compte à la fois de la puissance des CPU, de la vitesse des canaux de communications, de la quantité de mémoire, des unités de stockage, etc. Dans les prochains paragraphes, nous allons étudier les choix matériels et architecturaux possibles, ainsi que leurs avantages et inconvénients.

Rapport performances/prix

L'un des avantages des grappes d'ordinateurs est leur rapport performance/prix très intéressant car il est possible d'utiliser des composants grand-public. Mais nous avons vu que le système doit rester homogène : le problème est qu'à l'heure actuelle, les processeurs voient leur prix chuter alors qu'ils atteignent les performances des stations de travail. Au contraire, les réseaux performants ou spécialisés pour les systèmes parallèles sont encore relativement coûteux. Par exemple, A. Barak et al. ont montré que le réseau spécialisé *Myrinet* permet d'améliorer les performances d'une grappe d'ordinateurs car il diminue les délais de transmission et possède une bande passante importante [Barak99a]. Mais c'est aussi une architecture environ dix fois plus onéreuse qu'un réseau Ethernet. Dans ce contexte, il est clair que le prix intervient dans le choix de l'architecture du réseau. L'optimisation du rapport performance/prix est donc très compliquée car les composants matériels ont des impacts sur les performances et des

coûts variables. Gupta et Kumar [Kumar94] proposent d'utiliser l'efficacité ramenée au prix $Ec = \frac{S}{Coût}$ pour comparer les systèmes parallèles entre eux.

2.5.2 Architecture des systèmes parallèles et grappes d'ordinateurs

Jusqu'alors, nous avons plutôt abordé l'évolutivité des grappes d'ordinateurs d'un point de vue théorique. En effet, de nombreuses recherches se sont intéressées à l'évolutivité des systèmes parallèles sans tenir compte de leur architecture. Dans un premier temps, le but était plutôt de s'assurer dans quelle mesure un système parallèle est intéressant par rapport à un système séquentiel. Ensuite, les recherches ont voulu modéliser l'évolutivité des systèmes parallèles en général. Mais, avec l'avènement des grappes d'ordinateurs, les prix des systèmes parallèles ont beaucoup chuté et l'accent peut être mis sur l'expérimentation permettant de valider les résultats théoriques.

Nécessité d'une architecture répartie

Dès le début des travaux sur le parallélisme, avec les recherches d'Amdahl, on a compris que la fraction séquentielle est une barrière infranchissable pour le parallélisme. Il faut donc à tout prix la minimiser ou même l'éliminer si possible. Ainsi, un contrôle central limite forcément l'évolutivité car il constitue en fait une partie séquentielle. Selon [Buyya99], un contrôle central est facile à mettre en place mais doit être évité car il crée une non-tolérance aux pannes et constitue un risque potentiel de congestion. Comme nous le verrons par la suite, l'absence de contrôle central pour gérer la distribution des tâches et le partage des autres ressources n'est pas un problème trivial. Dans ce sens, Barak et Kornatzky [Barak87] ont établi qu'une architecture ou des algorithmes qui dépendent de la topologie et de la taille système ne sont pas évolutifs.

L'une des règles fondamentales est donc d'utiliser une architecture répartie, aussi bien du point de vue du matériel que du logiciel. Dans le cas contraire, l'évolutivité du système est limitée car il existera des possibilités de goulots d'étranglement. Cette contrainte est indépendante de la performance des composants du système : c'est-à-dire

qu'un système qui utilise des composants très performants mais dont l'architecture est mauvaise ne sera pas évolutif.

Selon Mukherjee [Mukherjee99], la capacité du système de stockage à servir des documents, la finesse de la distribution possible des tâches et l'absence de goulot d'étranglement dans un système, sont des facteurs critiques pour l'évolutivité d'un système de grande taille. Dans ce qui suit, nous allons aborder les deux premiers points évoqués ci-dessus.

Distribution des tâches

Dans l'étude théorique précédente, nous avons toujours considéré des « algorithmes parallèles » : c'est l'algorithme lui-même qui effectue les calculs et se charge de distribuer les tâches aux différents processeurs. C'est le cas pour certains algorithmes parallèles, mais essentiellement ceux spécialisés de calcul scientifique. Nous avons vu dans l'introduction que les grappes d'ordinateurs se destinent de plus en plus à des applications commerciales et professionnelles. Elles doivent donc être polyvalentes et c'est le système d'exploitation de la grappe qui se charge alors de distribuer les tâches aux différents processeurs. Par exemple, l'outil MOSIX pour les systèmes d'exploitation Linux implémente des algorithmes de partage des ressources pour les grappes d'ordinateurs [Barak99].

Dans le cas de calculs scientifique, la quantité totale de calculs est souvent connue : ainsi, la distribution des tâches est statique. Puisque c'est relativement facile à mettre en œuvre, c'est souvent le programme de calcul qui s'en charge, en utilisant des API de programmation parallèle. Mais pour un système parallèle qui se veut polyvalent, la distribution des tâches doit être dynamique. L'objectif est alors d'optimiser l'utilisation de tous les processeurs en répartissant les tâches le plus judicieusement possible. De nombreuses recherches ont été menées dans ce domaine. Trois politiques d'attribution des tâches sont possibles : au hasard, en fixant un seuil d'utilisation des processeurs ou en choisissant le moins chargé. L'attribution au hasard est la plus simple et ne fait intervenir aucune coopération entre les processeurs, mais peut entraîner des

surcoûts importants lorsque le « hasard fait mal les choses ». Pour la politique avec un seuil, on procède de la manière suivante : un certain nombre de processeurs vont être sondés et la tâche est attribuée au premier processeur qui est en dessous d'un seuil de charge fixé. Enfin, la dernière possibilité consiste à sonder des processeurs et de choisir le moins utilisé. Cependant, les études précédentes ont montré qu'il n'y a pas de gain lorsqu'on choisit le meilleure processeur plutôt qu'un processeur peu chargé [Buyya99].

En réalité, il existe des politiques d'attribution des tâches bien plus élaborées, qui peuvent prendre en compte de nombreux facteurs en plus de la charge des processeurs : par exemple, la « proximité du processeur », le taux d'utilisation de diverses ressources par un processeur, la vitesse des canaux de communication, le type de tâche, etc. La répartition des tâches peut aussi s'occuper de migrer des processus qui sont en cours d'exécution parce qu'un processeur est surchargé.

L'une des grandes difficultés de la distribution dynamique des tâches est que chaque processeur doit avoir des informations sur l'état d'autres processeurs à tout moment, puisqu'il est exclu d'utiliser une architecture centralisée. Les surcoûts de communications et de traitement même de la distribution des tâches peuvent devenir importants, réduisant les performances utiles de la grappe.

Stockage des données et systèmes de fichiers

Mukherjee [Mukherjee99] a souligné l'importance du système de stockage des données dans l'évolutivité du système. Pour ce qui est des applications commerciales, la tendance est d'utiliser des fichiers toujours plus volumineux car les documents intègrent de plus en plus de médias différents. Par exemple, des grappes d'ordinateurs sont utilisées comme serveurs de vidéo à la demande, qui nécessitent un débit important et constant. Sans aller à ces extrêmes, les disques durs sont aujourd'hui de plus en plus performants et il faut donc que l'architecture des systèmes de fichiers puisse les exploiter.

Les systèmes de fichiers pour grappes d'ordinateurs sont en fait très liés à l'architecture du système de stockage : deux raisons l'expliquent. Tout d'abord, pour

être performant, le système de fichier se doit d'être optimisé et donc doit exploiter au maximum les avantages de telle ou telle architecture. D'autre part, le but est de fournir un certain degré de transparence de service : l'accès aux données ne doit pas dépendre du choix de l'architecture, bien qu'il soit clair qu'il devra profiter de ses spécificités en terme de performance.

Comme pour la distribution des tâches, le système de fichiers ne doit pas avoir d'architecture centralisée. Une architecture répartie permet d'obtenir de meilleures performances grâce au parallélisme. Mais avant tout, c'est une condition pour que le système soit évolutif et exempt de goulot d'étranglement.

Il existe trois méthodes de distribution des données : la réplication, le partage de données réparties et le « wide striping ». La réplication n'est efficace que pour des grappes de petite taille et lorsque les données ne sont pas modifiées trop fréquemment. En revanche, la réplication n'est pas une solution pour l'évolutivité [Mukherjee99]. Gérer une quantité importante de répliquas devient vite impossible et pose des problèmes de consistance des données.

Les serveurs de fichiers partagés sont une meilleure solution. Mais la manière dont les données sont réparties, comme illustré à la Figure 2.2, conduit forcément à des déséquilibres dans l'utilisation des disques durs : un disque dur contenant une donnée très demandée peut devenir surchargé et constituer un goulot d'étranglement. D'autre part, si un fichier volumineux se trouve sur une seule unité de stockage, l'accès au fichier va monopoliser le disque dur concerné. Cette solution a donc des limites, mais est

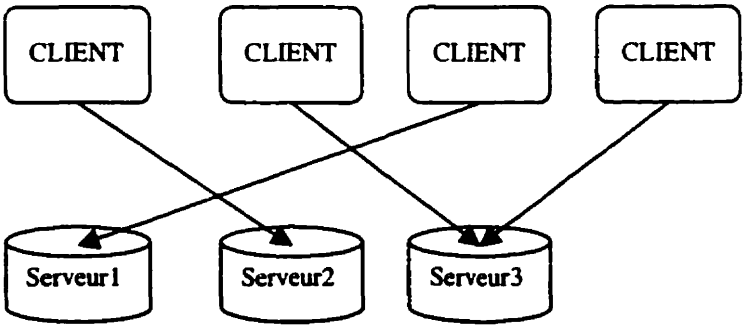

Figure 2.2 Partitionnement des données

néanmoins performante pour certaines applications : les fichiers doivent être de petite taille et les accès doivent être relativement bien répartis sur l'ensemble des disques.

Enfin le « wide striping » est la solution préconisée par Mukherjee [Mukherjee99] selon qui, pour les systèmes de fichiers, la véritable scalabilité ne peut pas être atteinte en utilisant la réplication et la distribution des données. Pour avoir une bande passante importante et respecter des contraintes temps réel, le « wide striping » est

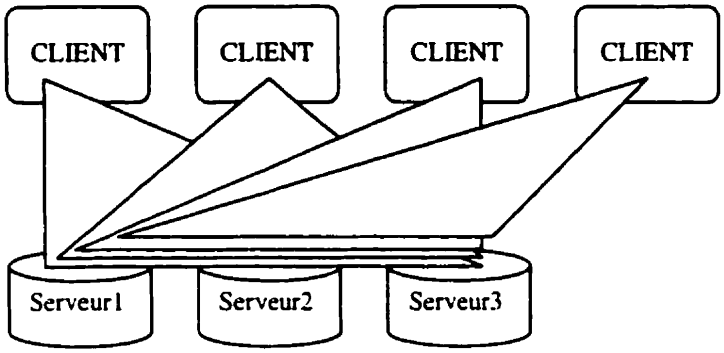

Figure 2.3 Wide Striping

un bon choix puisqu'il possède également de très bonnes propriétés de répartition de la charge. Le « wide striping » consiste en fait à distribuer les données, mais au niveau des blocs de données : les blocs qui appartiennent à un même fichier forment des groupes qui sont répartis sur plusieurs unités de stockage, comme illustré à la Figure 2.3. L'accès aux disques durs est ainsi réparti uniformément. D'autre part, il existe des techniques d'optimisation (par exemple le « graduated declustering » [Dusseau99], les files d'attentes distribuées) qui permettent de synchroniser les disques entre eux pour encore améliorer la coopération de ces derniers. Ce concept est né à l'Université de Berkeley dans les années 1980 avec les architectures RAID (Redundant Array of Inexpensive Disks) [Massiglia2000].

La Figure 2.4 explique l'idée du « wide striping » : le disque virtuel est en fait constitué de blocs réels, mais répartis régulièrement sur l'intégralité des disques. Dans cette figure, sont représentées deux variables importantes : la profondeur et la largeur du « stripe ». Leurs valeurs devront être choisies judicieusement pour optimiser les

performances du système. Outre le fait que les blocs d'un fichier sont répartis, on peut constater sur cette figure un autre avantage de cette architecture : la coopération entre les disques est accrue. Par exemple, la requête pour le fichier est traduite par le disque virtuel en :

1) Une requête au disque A pour les segments 000 et 001
2) Une requête au disque B pour les segments 002 à 005
3) Une requête au disque C pour les segments 006 à 009
4) Une requête au disque A pour les segments 010 à 013
5) Une requête au disque B pour les segments 014 à 017
6) Une requête au disque C pour le segment 018.

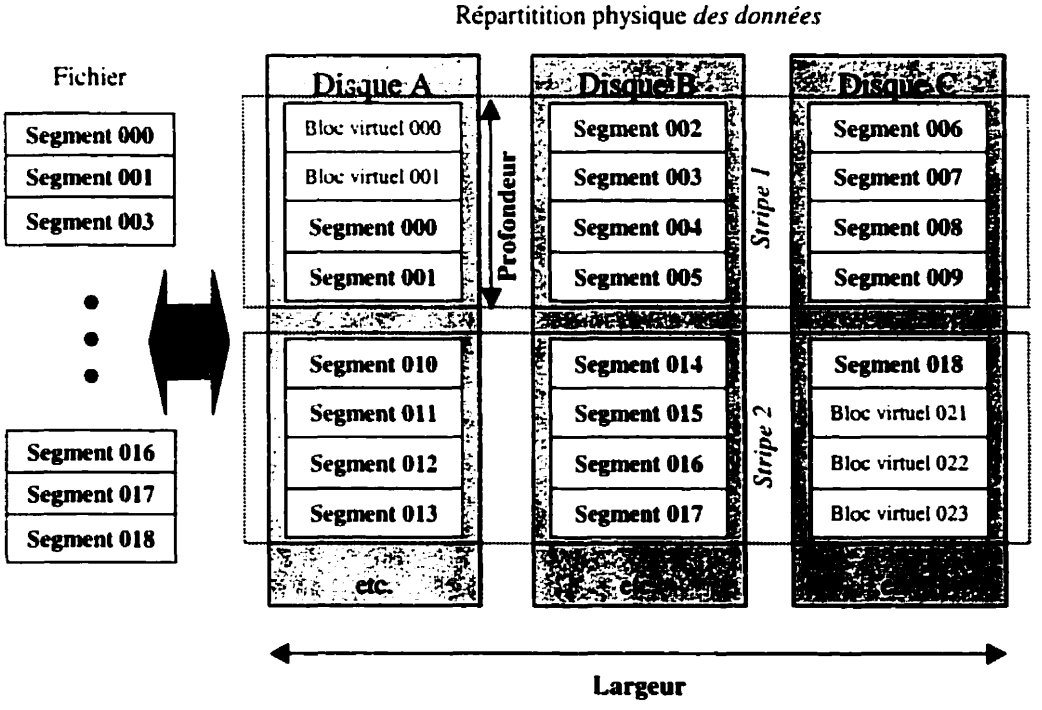

Figure 2.4 Principe du « wide striping »

Les trois premières requêtes concernent des disques différents et peuvent donc avoir lieu simultanément. De plus, lorsque la requête 1) a fini d'être traitée, la requête 4) peut débuter. En conclusion, les avantages du « wide striping » sont résumés à la Figure

2.5 : le « wide striping » est le plus avantageux dans le cas de fichier volumineux et dont les accès sont séquentiels.

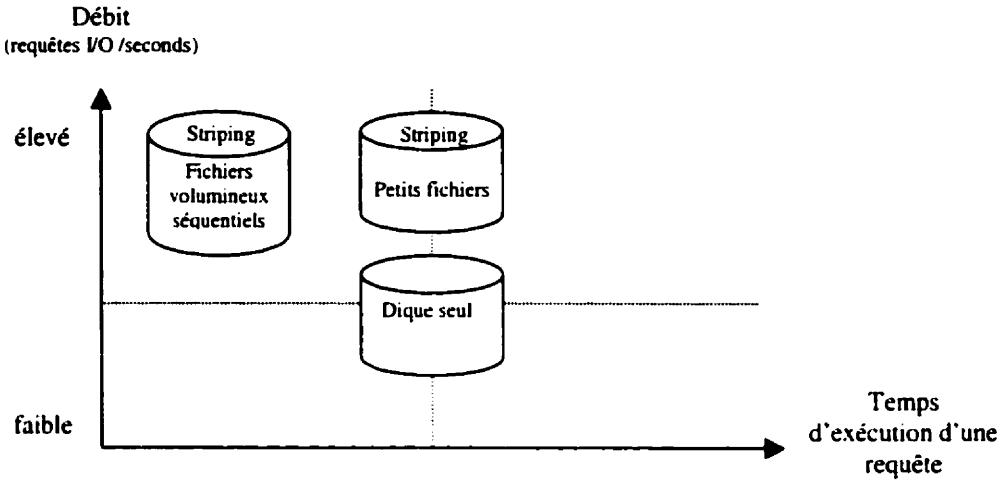

Figure 2.5 Avantages du « striping » des données

CHAPITRE 3
CONCEPTION D'UN MODÈLE DE GRAPPE D'ORDINATEURS

Les grappes d'ordinateurs constituent un type bien particulier de système parallèle. Le but de ce chapitre est d'élaborer un modèle théorique de grappe d'ordinateurs nous permettant d'étudier leur évolutivité. Dans un premier temps, nous allons décrire notre modèle et les raisons qui ont motivé nos choix. Puis, nous analyserons ce modèle pour évaluer la performance du système en fonction de ses caractéristiques. Enfin, nous élaborerons un modèle de simulation permettant de dégager des résultats quantitatifs qui seront analysés.

3.1 Description du modèle analytique

Pour modéliser les grappes d'ordinateurs, nous avons choisi une approche comportementale selon laquelle le système est décomposé en modules qui reproduisent la réalité. Cependant, le but ici étant d'obtenir des résultats surtout qualitatifs, nous n'avons pas modélisé chaque composante par son module correspondant. Nous avons plutôt choisi de regrouper les composantes qui ont des caractéristiques similaires. Une des raisons est que les calculs théoriques sont ainsi simplifiés. Pour obtenir des résultats quantitatifs, il convient mieux d'utiliser des méthodes numériques ou effectuer des simulations plus poussées : ces deux aspects seront abordés dans la dernière partie de ce chapitre.

L'un des objectifs de ce modèle est de partir d'une structure connue au niveau microscopique, pour en extraire un comportement général au niveau macroscopique. En d'autres termes, nous allons modéliser le comportement de chacun des processeurs pour étudier les performances globales du système parallèle. Nous allons ainsi utiliser les probabilités pour représenter le niveau microscopique : les temps d'exécution sont ainsi représentés par une variable aléatoire appropriée, sans chercher à simuler l'exécution d'un programme dans ses détails. Ainsi, nous ne nous intéresserons pas à simuler un programme au niveau de ses instructions : le but est d'arriver à un modèle plus général qui pourra être adapté à différentes situations.

Comme l'illustre la Figure 3.1, nous utilisons dans ce modèle trois types de composantes : les processeurs, les ressources partagées et un délai. Les processeurs font des calculs et effectuent des requêtes lorsqu'ils ont besoin d'accéder à une ressource non locale; ils constituent donc la seule ressource de calcul de notre modèle. Nous voulons déterminer la limite de performance, donc nous considérons qu'un processeur a toujours un processus à exécuter. Par contre, lors de l'accès à une ressource, le processeur devient inactif jusqu'à ce que la réponse à sa requête lui parvienne : **chaque processeur ne traite donc qu'un seul processus à la fois.**

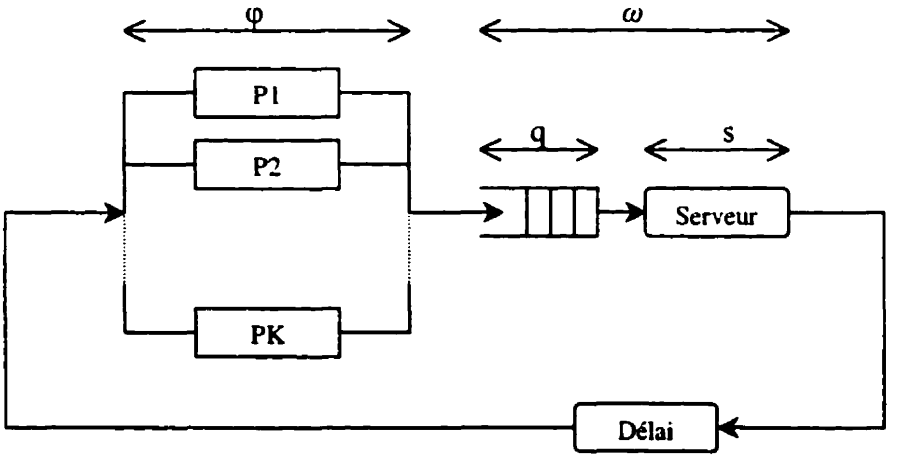

Figure 3.1 Schéma du modèle analytique

Le système est composé de K processeurs. De plus, toutes les requêtes partagent une ressource commune et subissent un même délai. La ressource partagée vise à modéliser une composante dont le temps de réponse dépend du nombre de requêtes. Si une requête arrive alors que la ressource partagée (représentée comme un serveur à la Figure 3.1) sert déjà une autre requête, alors elle est placée dans une file d'attente dotée d'une politique FIFO (premier arrivé, premier servi). Le délai représente une ressource qu'on suppose indépendante des deux autres composantes du système : il dépend uniquement de la topologie du système. Par exemple, le délai peut modéliser les communications ou les temps de traitement supplémentaire. Par définition, nous appelons φ le temps de calcul utile d'un processeur, w le temps d'attente totale, q le temps d'attente dans la file d'attente et s le temps de service.

Ce modèle repose sur l'hypothèse que **les processeurs ont un comportement cyclique**. Nous nous bornerons ainsi à une situation de calcul bien précise pour laquelle les caractéristiques d'utilisation des ressources du système ne varient pas. Si ce n'est pas le cas, il faudra considérer séparément toutes les phases par lesquelles passe le système. Pour modéliser le système, nous avons donc choisi les réseaux de Pétri : le principe est de représenter le système à travers les états et les transitions possibles. L'évolution de l'état d'un processeur est représentée par le parcours du réseau. Nous utilisons ici la classe des réseaux de Pétri stochastiques généralisés (GSPN, Generalised Stochastic Petri Net) qui permettent d'étudier à la fois le comportement mais aussi les performances du système. La Figure 3.2 représente le réseau de Pétri correspondant au modèle : il est composé d'états (représentés par des cercles) et de transitions (représentées par les lignes horizontales) reliés par des arcs. Chaque transition a lieu suivant une loi de probabilité déterminée.

Dans ce modèle, nous faisons aussi l'hypothèse que **tous les processeurs sont identiques**, ce qui permet de simplifier le réseau de Pétri en regroupant tous les processeurs sur le même graphe d'états. Les processeurs sont représentés par des jetons qui se déplacent d'état en état. Dans l'état p_1, un processeur exécute son programme : on considère que dans cet état, il fait du « calcul utile ». L'événement t_1 correspond à

l'émission d'une requête par un processeur : à partir du moment où une requête est émise, le processeur attend sans rien exécuter tant que la réponse à la requête ne lui est pas parvenue. Le serveur *s* modélise la ressource partagée : une seule requête peut être servie à la fois. Dans l'état p_2, les requêtes sont placées dans une file d'attente afin de pouvoir accéder à la ressource commune. Entre les événements t_2 (début du service) et t_3 (fin du service), l'état p_3 correspond à l'accès à la ressource partagée. Une fois servie, la requête subit encore un délai matérialisé par l'état p_4 et l'événement t_4 (fin du délai). Finalement, on peut remarquer que ce modèle constitue un système fermé de file d'attente : le comportement de chaque composante est cyclique et le nombre de requêtes est borné par le nombre de processeurs.

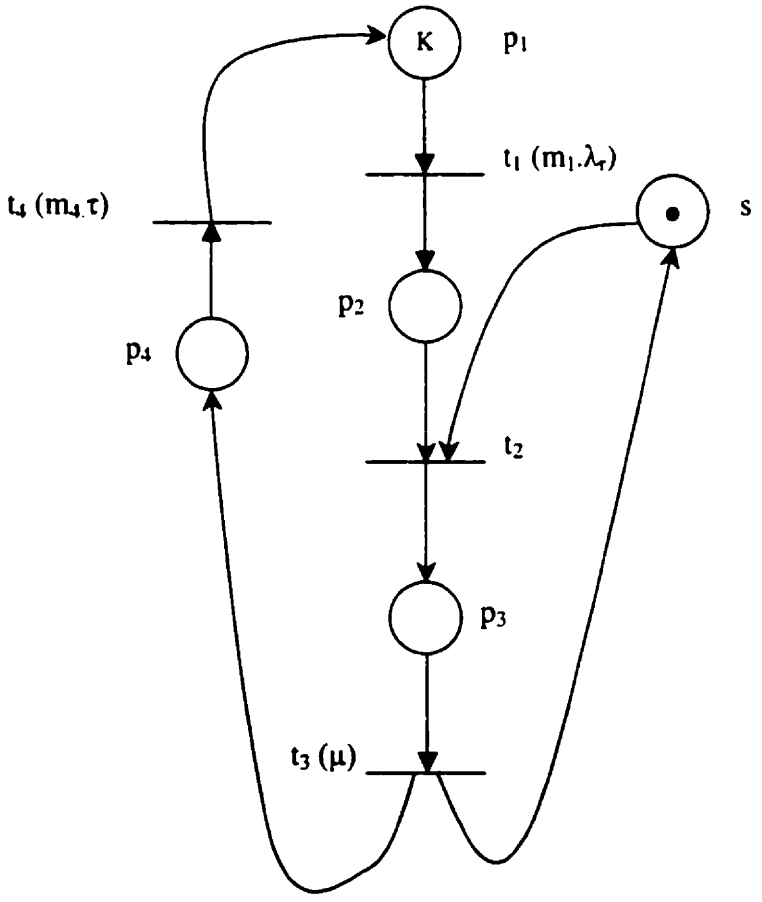

Figure 3.2 Représentation du modèle par un réseau de Pétri

Pour notre modèle de programme parallèle, nous avons choisi de ne pas prendre en compte dans le détail toutes les interactions entre les processeurs telles que les synchronisations, les échanges d'information, etc. Nous avons préféré regrouper ces interactions dans la composante de ressource partagée. Nous avons aussi choisi de modéliser l'évolution entre les états par des évènements qui suivent des distributions exponentielles. La raison de ce choix est que la distribution exponentielle est sans mémoire. Puisque nous avons déjà regroupé tous les processeurs sur un même graphe en considérant le système homogène, la propriété sans-mémoire nous permet d'aller plus loin et de ne plus différencier les processeurs entre-eux : les transitions deviennent ainsi complètement indépendantes des processeurs. On peut justifier le choix de la distribution exponentielle par l'hypothèse suivante : le programme exécuté par un processeur ne conserve pas de donnée sur son passé en tant que tel, c'est-à-dire que les seules informations qu'un processeur possède dépendent uniquement de l'état dans lequel il se trouve et les distributions qui régissent les transitions sont donc sans mémoire. Or la seule distribution sans mémoire est la distribution exponentielle.

Les taux d'arrivée et de service sont indiqués à la Figure 3.2. Chaque processeur émet des requêtes suivant une loi exponentielle de paramètre λ_r : étant donné qu'il y a m_l processeurs dans l'état p_l, les arrivées des requêtes au serveur suivent une loi exponentielle de paramètre $m_l.\lambda_r$. La politique de service de la ressource partagée suit aussi une distribution exponentielle de paramètre μ. Enfin, le délai suit aussi une distribution exponentielle de paramètre τ pour chaque processeur en attente. Le taux de transition de l'état p_4 est donc $m_4\tau$.

Cette modélisation est aussi équivalente à une chaîne de Markov à temps continu [Davies1994] et nous utiliserons les deux représentations pour notre étude théorique : le Tableau 3.1 indique les correspondances entre les paramètres. Les hypothèses que nous avons formulées ont pour conséquence que la chaîne de Markov représentant le système a les caractéristiques suivantes : elle est homogène, irréductible, récurrente non nulle et apériodique. Nous pouvons alors utiliser un théorème de la théorie des files d'attente [Kofman98] pour conclure qu'un état stationnaire existe. Cet état stationnaire est

indépendant des conditions initiales et c'est ce régime stationnaire qui fera l'objet de notre étude. Puisque l'état stationnaire existe, nous pouvons déterminer les probabilités stationnaires de chaque état : c'est la probabilité que le système se trouve dans un état donné.

Tableau 3.1 Correspondance entre le réseau de Pétri et le système de Markov

Paramètre	Chaîne de Markov	Réseau de Pétri
λ_r	Taux de requête d'un processeur	Taux de transition d'un jeton de p_1 à p_2
λ	Taux de requête total	$m_1\lambda_r$: taux de transition total de p_1 à p_2
q	Nombre de requêtes dans la file d'attente	Nombre de jeton en p_2 (m_2)
ω	Temps passé dans le serveur (temps d'attente et de service)	Latence d'un jeton en p_2 et p_3
s	Temps de service	Latence d'un jeton en p_3
T	Délai subi	Latence d'un jeton en p_4
N	Nombre total de requêtes	$m_2 + m_3 + m_4$

Le serveur de ressource partagée est équivalent au système de Markov M/M/1/K/K (notation de Kendall) : les arrivées et la politique de service sont markoviens, avec un seul serveur dont le nombre maximum de requêtes en attente et en train d'être servies est K, pour une population finie de K requêtes. Le système de file d'attente M/M/1/K/K a été étudié depuis les années 1950 et il a notamment été utilisé pour modéliser des requêtes d'entrée/sortie. Ce système peut aussi bien modéliser l'accès à une mémoire partagée : les processeurs effectuent du calcul utile tant qu'ils

n'ont pas besoin d'accéder à une donnée partagée. Pour accéder à une donnée partagée, un processeur doit effectuer une requête et attendre la réponse pour pouvoir continuer son calcul. Pendant ce temps, le processeur est inactif : l'efficacité du système dépend donc du temps de réponse aux requêtes. Ce système étant fermé, nous pouvons aussi utiliser la propriété de conservation du flot : elle nous sera utile pour l'étude analytique qui suit.

En ce qui concerne le délai, nous considérons qu'il ne dépend pas de la charge du système : il n'y a pas de concurrence, contrairement à la ressource partagée. De plus, le temps de service est exponentiel et **le délai est donc équivalent à une file d'attente M/M/∞** comportant une infinité de serveurs. Il faut aussi remarquer que le délai a été placé après le serveur de ressource partagée dans le réseau de Pétri. Mais son positionnement dans le temps n'a pas d'influence sur les caractéristiques et les performances puisque toutes les distributions sont sans mémoire.

3.2 Étude analytique

Pour étudier les propriétés générales du système, nous préférons utiliser la représentation du modèle sous la forme d'un réseau de Pétri plutôt qu'une représentation par chaîne de Markov. Car, un réseau de Pétri représente plus intuitivement la structure du système. En utilisant la propriété de conservation du flot, nous allons en déduire les relations entre le taux de requête, le taux de service, l'utilisation de la ressource partagée, les temps d'attente et le débit de requêtes. Nous notons ρ le taux d'utilisation de la ressource partagée, c'est à dire le taux d'utilisation du serveur dans notre modèle. Le débit du système sera noté D et mesure le nombre de requêtes traitées par seconde. Par conservation du débit, le débit du système est égal au nombre de requêtes arrivant de l'ensemble des processeurs par seconde. Quand le serveur est occupé, μ requêtes sont traitées par unité de temps. Le débit du système est donc :

$$D = \lambda = \rho.\mu \qquad (3.1)$$

La Figure 3.3 représente le cycle d'un processeur. En utilisant les notations introduites au Tableau 3.1, nous définissons pour un cycle les moyennes dans le temps suivantes :

$\overline{\omega}$: temps moyen passé dans le serveur de ressource partagée ;

$\overline{\varphi}$: temps moyen de calcul utile ;

$\overline{T}$: délai moyen.

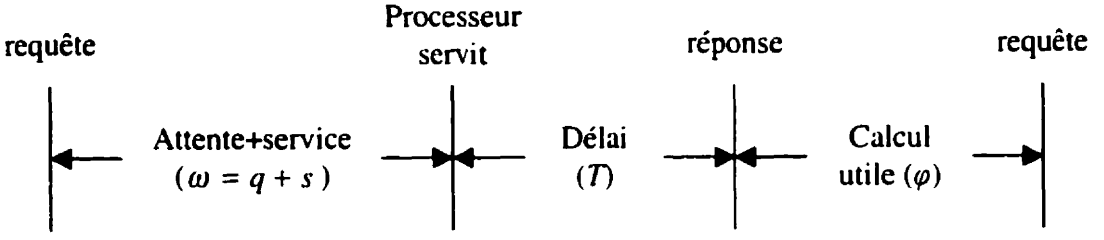

Figure 3.3 Cycle d'un processeur

On peut en déduire une expression du taux de requêtes pour un processeur comme l'inverse du temps moyen pour effectuer un cycle complet :

$$\lambda_r = \frac{1}{\overline{\omega} + \overline{T} + \overline{\varphi}}$$

Pour K processeurs, on a donc :

$$\lambda = \frac{K}{\overline{\omega} + \overline{T} + \overline{\varphi}}$$

Du fait de la conservation du débit, on peut donc écrire :

$$\rho.\mu = \frac{K}{\overline{\omega} + \overline{T} + \overline{\varphi}}$$

Avec les taux de transitions de la Figure 3.2, on obtient finalement :

$$\overline{\omega} = \frac{K}{\rho.\mu} - \frac{1}{\tau} - \frac{1}{\lambda_r} \qquad (3.2)$$

Il faut remarquer que ce résultat est complètement indépendant des distributions utilisées dans le modèle. L'équation (3.2) est donc valable pour tout système de file

d'attente de même architecture dont le temps de service moyen est $1/\mu$, le temps moyen entre deux requêtes $1/\lambda_r$ et le délai moyen $1/\tau$.

Nous allons maintenant nous intéresser aux propriétés liées à la distribution exponentielle des arrivées et du temps de service. Le système de Markov du système complet ressemble au système M/M/1/K/K que nous avons décrit précédemment, étendu par le fait qu'un processeur peut aussi être inactif car il subit un délai. Le système de Markov de naissance et de mort est ainsi triangulaire, comme le montre la Figure 3.4 représentant un système à trois processeurs. Pour un système à K processeurs, la représentation est identique, sauf qu'elle comporte plus d'états. Du fait de la propriété sans mémoire du système, chaque état est complètement déterminé par le nombre de processeurs en attente et le nombre de processeurs qui sont en train de subir le délai : c'est respectivement la signification des deux chiffres qui caractérisent un état dans le système triangulaire de Markov. Nous allons ainsi déterminer la forme générale de la probabilité de se trouver dans chaque état. Nous utilisons la notation suivante :

$P(x,y)$ désigne la probabilité de se trouver dans l'état (x,y)

où :

Nombre de processeurs qui attendent la réponse à une requête :

$x = m_2 + m_3 + m_4$

Nombre de processeurs dont la requête est en train de subir un délai :

$y = m_4$

Entre les différents états, les transitions qui sont considérées dans le système triangulaire de Markov sont :

- l'émission d'une requête par un processeur (flèche horizontale) ;
- la fin du traitement d'une requête par le serveur de ressource partagée (flèche verticale) ;
- la fin d'un délai pour une requête (flèche en diagonale).

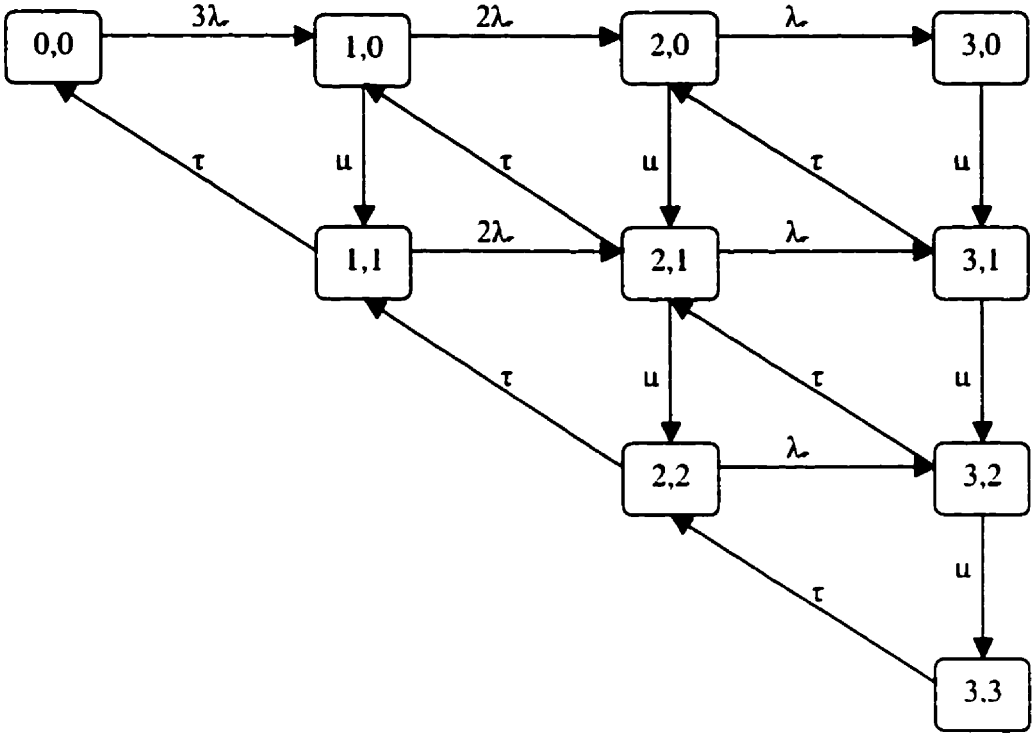

Figure 3.4 Système triangulaire de Markov (trois processeurs)

Le système de Markov permet d'écrire les égalités suivantes :

$$K.\lambda.P(0,0) = \tau.P(1,1)$$

$$(\mu + (K-1).\lambda).P(1,0) = K.\lambda P(0,0) + \tau.P(2,1)$$

$$(\tau + (K-1).\lambda).P(1,1) = \mu.P(1,0) + 2\tau.P(2,2)$$

etc.

On remarque donc que dans le cas général :

pour un nœud intérieur :

$$(\mu + j\tau + (K-i).\lambda).P(i,j) = (K-i+1)\lambda.P(i-1,j) + \mu.P(i,j-1) + (j+1)\tau.P(i+1,j+1)$$

pour un nœud du bord supérieur :

$$(\mu + (K-i).\lambda).P(i,j) = (K-i+1)\lambda.P(i-1,j) + (j+1)\tau.P(i+1,j+1)$$

pour un nœud du bord inférieur :

$$(K - i).\lambda_r.P(i, j) = \mu.P(i, j-1) + (j+1)\tau.P(i+1, j+1)$$

Nous remarquons également que les nœuds « sur les bords » sont des cas particuliers des nœuds intérieurs car certaines transitions ne leur sont par permises. En posant :

$$\delta(x) = \begin{cases} 1, \text{ si } x>0 \\ 0, \text{ sinon} \end{cases}$$

on peut écrire une formule générale :

$$[\delta(K-i)(K-i).\lambda_r + \delta(i-j)\mu + \delta(j).j].P(i, j) =$$
$$\delta(i)\delta(i-j)(K-i+1)\lambda_r.P(i-1, j)$$
$$+ \delta(j)\mu.P(i, j-1)$$
$$+ \delta(K-i)(j+1)\tau.P(i+1, j+1)$$

On peut alors démontrer [Davies1994] que la probabilité de se trouver dans un état donné est de la forme :

$$P(i, j) = \frac{K!}{(K-i)!\, j!} u^i d_r^j.P(0,0) \qquad (3.3)$$

où : $u = \frac{\lambda_r}{\mu}$ (charge) et $d_r = \frac{\mu}{\tau}$ (taux de délai).

La condition de normalisation donne :

$$P(0,0) = \left[\sum_{i=0}^{K} \sum_{j=0}^{K} \frac{K!}{(K-i)!\, j!} u^i d_r^j \right]^{-1}$$

3.3 Évaluation de performance

La charge du serveur (ressource partagée) est déterminée par le temps pendant lequel le serveur est inactif. Le serveur est inactif lorsqu'il n'y a aucune requête dans le système ou lorsque toutes les requêtes du système sont en train de subir un délai, ce qui donne :

$$\rho = 1 - \sum_{i=0}^{K} P(i,i) = 1 - \sum_{i=0}^{K} \frac{K!}{(K-i)!\,j!} u^i d_r^j \left[\sum_{i=0}^{K} \sum_{j=0}^{K} \frac{K!}{(K-i)!\,j!} u^i d_r^j \right]^{-1}$$

$$\rho = 1 - \frac{(1+u.d_r)^K}{\sum_{i=0}^{K} \sum_{j=0}^{K} \frac{K!}{(K-i)!\,j!} u_i d_r^j} \qquad (3.4)$$

Nous pouvons alors en déduire le débit total de requêtes à partir de l'équation (3.1) :

$$\lambda = \rho.\mu = \left[1 - \frac{(1+u.d_r)^K}{\sum_{i=0}^{K} \sum_{j=0}^{K} \frac{K!}{(K-i)!\,j!} u^i d_r^j} \right] \mu$$

À partir du temps moyen de réponse exprimé par l'équation (3.2) et de la formule de Little, on peut déduire la taille moyenne de la file d'attente du serveur de ressource partagée :

$$L = \lambda.\bar{\omega}$$

or $\lambda = \rho.\mu$ donc :

$$L = \rho\mu.\bar{\omega} = K - \frac{\rho\mu}{\tau} - \frac{\rho\mu}{\lambda_r}$$

$$L = K - \rho.(d_r + 1/u)$$

Temps passé dans la file d'attente :

$$\bar{\omega} = \bar{q} + \bar{s} \qquad \Rightarrow \bar{q} = \bar{\omega} - \bar{s} = \bar{\omega} - 1/\mu$$

$$\Rightarrow \bar{q} = \frac{K}{\rho\mu} - \frac{1}{\tau} - \frac{1}{\lambda_r} - \frac{1}{\mu}$$

La Figure 3.3 illustre le fait que, pour un cycle, un processeur est actif pendant $\bar{\varphi}$ en moyenne. Le reste du cycle, le processeur attend la réponse à une requête et demeure donc inactif. L'inactivité I d'un processeur s'exprime comme suit :

$$I = \frac{\overline{\omega} + \overline{T}}{\overline{\omega} + \overline{T} + \overline{\varphi}} = \frac{\overline{\omega} + 1/\tau}{\overline{\omega} + 1/\tau + 1/\lambda} = \frac{\lambda.(\overline{\omega}\tau + 1)}{\lambda(\overline{\omega}\tau + 1) + \tau}$$

On peut d'ailleurs dériver de cette équation la formule suivante :

$$\overline{\omega} = \frac{\lambda.I + \tau.I - \lambda}{\lambda.\tau.(1 - I)} = \frac{I.(\lambda + \tau) - \lambda}{\lambda.\tau.(1 - I)}$$

En utilisant (3.2), on peut écrire :

$$\frac{K}{\rho.\mu} - \frac{1}{\tau} - \frac{1}{\lambda} = \frac{I.(\lambda + \tau) - \lambda}{\lambda.\tau.(1 - I)}$$

$$\Rightarrow u = \frac{\rho}{K(1 - I)}$$

$$\Rightarrow u = \frac{1 - \sum_{i=0}^{K} P(i, i)}{K(1 - I)} \qquad (3.5)$$

Cette dernière équation (3.5) nous permet théoriquement d'obtenir la charge de la ressource partagée à partir de la mesure de l'inactivité des processeurs. En pratique, il faut résoudre une équation polynomiale d'ordre K et on se tournera vers une résolution numérique plutôt qu'une résolution analytique complète.

Une bonne mesure de performance d'un système parallèle est le temps total de calcul utile. Plus le système passe de temps à faire du calcul utile, plus il est capable de traiter un nombre important de requêtes. Nous allons commencer par faire ce calcul pour un processeur en déterminant le temps de calcul utile par cycle T_u (ramené au temps du cycle) :

$$T_u = \frac{\overline{\varphi}}{\overline{\omega} + \overline{T} + \overline{\varphi}} = \frac{1/\lambda}{\overline{\omega} + 1/\tau + 1/\lambda}$$

$$\Rightarrow T_u = \frac{\tau}{\lambda.\tau.\overline{\omega} + \lambda + \tau}$$

Pour K processeurs, le temps de calcul utile total s'exprime comme suit :

$$K.T_u = \frac{K.\tau}{\lambda.\tau.\overline{\omega} + \lambda + \tau}$$

En utilisant l'équation (3.2) pour substituer $\overline{\omega}$ dans la formule, on obtient :

$$K.T_u = \frac{K.\tau}{\lambda_r.\tau\left[\frac{K}{\rho.\mu} - \frac{1}{\tau} - \frac{1}{\lambda_r}\right] + \lambda_r + \tau} = \frac{K.\tau}{\frac{\lambda_r.\tau.K}{\rho.\mu} - \lambda_r - \tau + \lambda_r + \tau}$$

$$\Rightarrow K.T_u = \frac{K.\tau}{\frac{\lambda_r.\tau.K}{\rho.\mu}} = \frac{\rho}{u} \qquad (3.6)$$

Dans cette dernière expression, u est constante et fixée par les caractéristiques du système parallèle. Pour l'évaluation numérique, on utilise l'équation (3.4) pour substituer ρ :

$$K.T_u = \frac{1}{u} \times \left[1 - \frac{(1 + u.d_r)^K}{\sum_{i=0}^{K}\sum_{j=0}^{K} \frac{K!}{(K-i)!\,j!} u^i d_r^j}\right] \qquad (3.7)$$

La Figure 3.5 représente $K.T_u$ en fonction du nombre de processeurs, c'est à dire l'évolution du temps de calcul utile quand on augmente K. Pour ces tracés, u est fixée à la valeur 0,05 et chaque courbe représente une valeur de d_r différente. Nous pouvons remarquer qu'il existe une asymptote commune à toutes les courbes et on peut démontrer que c'est la droite d'équation $y = 1/u$. Le délai n'affecte plus le taux de temps de calcul utile au-delà d'une certaine valeur de K qui dépend de u. En effet, au-delà de cette valeur, c'est l'influence de u qui prend le dessus et qui limite l'évolutivité du système en constituant un goulot d'étranglement. La valeur du délai modifie seulement la vitesse à laquelle la fonction se rapproche de son asymptote : plus le délai est grand, moins la pente de la courbe est importante. Pour un délai important, il faut donc un plus grand nombre de processeurs pour atteindre une même valeur de temps de calcul utile. C'est ce que montre les valeurs du Tableau 3.2.

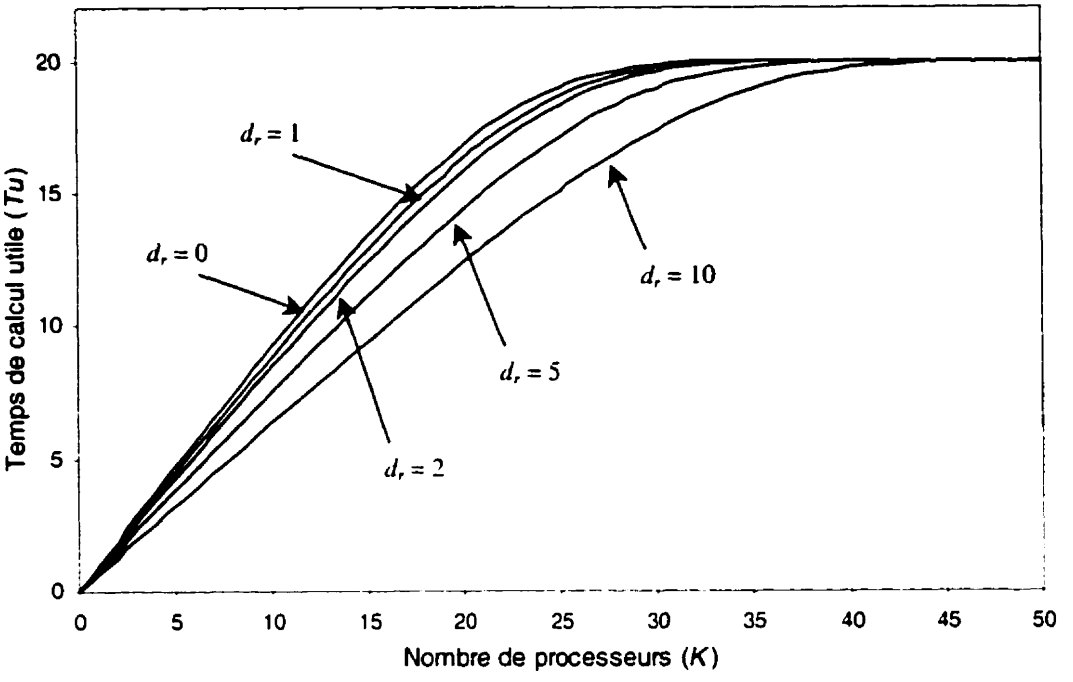

Figure 3.5 Variation du temps de calcul utile en fonction du nombre de processeurs quand d_r varie (u = 0,05)

Tableau 3.2 Valeur minimale du nombre de processeurs K pour que le temps de calcul dépasse 95 % de la valeur asymptotique

Nombre de processeurs nécessaires	Valeur de d_r	Pourcentage du maximum asymptotique
26	0	96,28 %
27	1	96,20 %
28	2	96,12 %
31	5	95,93 %
36	10	95,71 %

La Figure 3.6 démontre bien l'influence de u sur les performances absolues du système parallèle. Comme nous l'avons souligné, c'est la valeur de u qui détermine l'asymptote de la courbe et qui limite donc les performances.

Intéressons nous maintenant à l'accélération parallèle de ce système telle qu'elle est définie dans le chapitre 2. Reprenons l'équation (3.6) pour un processeur en substituant ρ grâce à l'équation (3.4). Nous obtenons alors :

$$T_u = \frac{\rho}{u} = \frac{1 - (P(0,0) + P(1,1))}{u} = \frac{1 - P(0,0).(1 + u.d_r)}{u}$$

$$(3.4) \Rightarrow T_u = \frac{1}{u} \times \left[1 - \frac{(1 + u.d_r)}{1 + u + u.d_r} \right]$$

$$\Rightarrow T_u = \frac{1}{1 + u + u.d_r} \qquad (3.8)$$

Dans le système avec un seul processeur, il n'y a plus de partage du serveur de ressource partagé, mais le délai s'applique toujours de la même manière puisqu'il représente un surcoût inhérent à l'architecture du système.

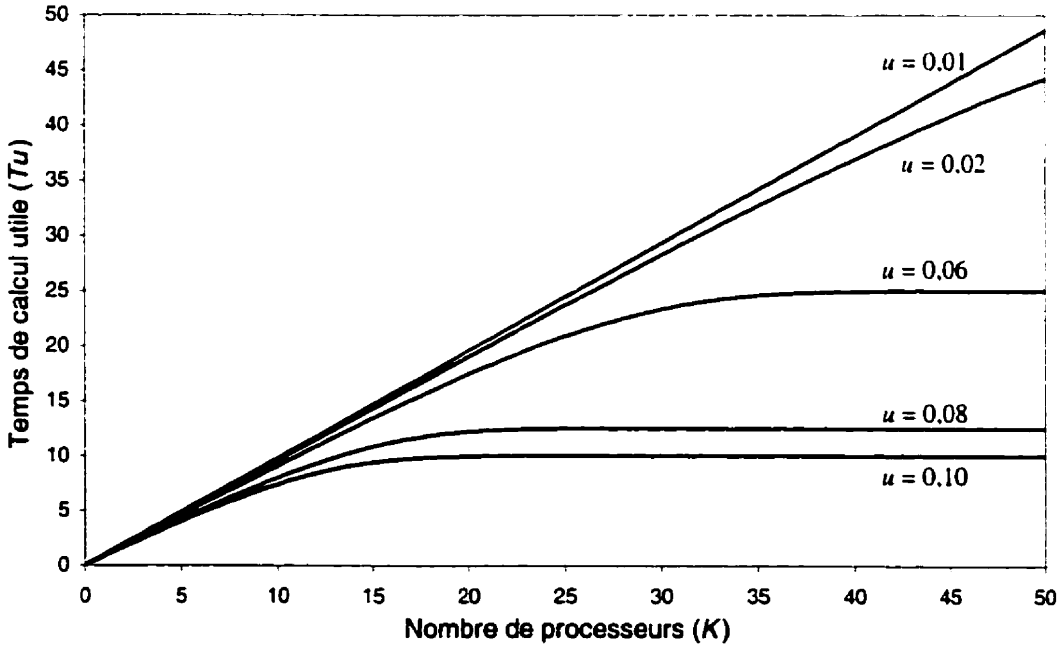

Figure 3.6 Variation du temps de calcul utile en fonction du nombre de processeurs quand u varie ($d_r = 1$)

À partir de l'équation (3.6), on peut donc en déduire l'accélération parallèle :

$$S = \frac{1 + u + u.d_r}{1} \times \frac{\rho}{u} . = \frac{1 + u + u.d_r}{u} \times \left[1 - \sum_{i=0}^{K} P(i, i) \right] \qquad (3.9)$$

La Figure 3.7 pour l'accélération parallèle a été tracée avec les mêmes paramètres que la Figure 3.5. Les équations (3.7) et (3.9) ne diffèrent que d'une constante puisque u et d_r sont fixés. Ainsi, chaque courbe de la Figure 3.7 correspond à une courbe de la Figure 3.5 multipliée par une constante. Cependant, ce facteur constant varie d'une courbe à l'autre, ce qui permet de mieux les différencier. Les courbes de l'accélération parallèle équivalentes à la Figure 3.6 n'ont pas été tracées, car pour les valeurs de u et d_r choisies, les courbes sont quasiment identiques. De plus, les courbes étant bien séparées sur la Figure 3.5, il ne nous a pas semblé pertinent de les retracer pour l'accélération parallèle.

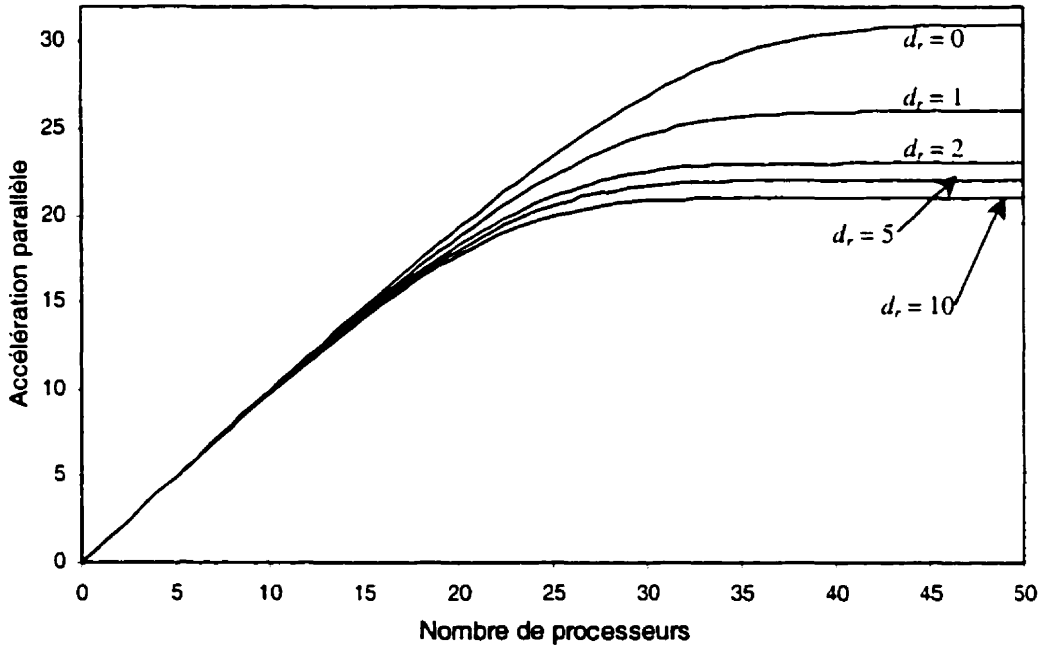

Figure 3.7 Variation de l'accélération parallèle en fonction du nombre de processeurs quand d_r varie (u = 0,05)

Dans ce modèle, l'accélération parallèle du système est proportionnelle à la charge du serveur de ressource partagée (équations (3.6) et (3.9)). La charge du serveur exprimée dans l'équation (3.4) est représentée à la Figure 3.8. Le cas de l'accélération parallèle linéaire a été représenté sur la même figure. Nous distinguons clairement sur cette figure la partie asymptotique qui correspond à la saturation de la ressource partagée.

Le modèle analytique que nous venons d'étudier avait pour but de dégager des résultats qualitatifs sur l'évolutivité des grappes d'ordinateurs. Nous l'avons élaboré en faisant le minimum d'hypothèses sur l'architecture pour que le modèle soit le plus générique possible. Ainsi, nous avons représenté deux composantes caractéristiques auxquelles les nœuds d'une grappe peuvent faire appel : une ressource partagée et une ressource introduisant un délai pur. Les courbes qui ont été tracées permettent de se rendre compte de l'influence de chaque composante.

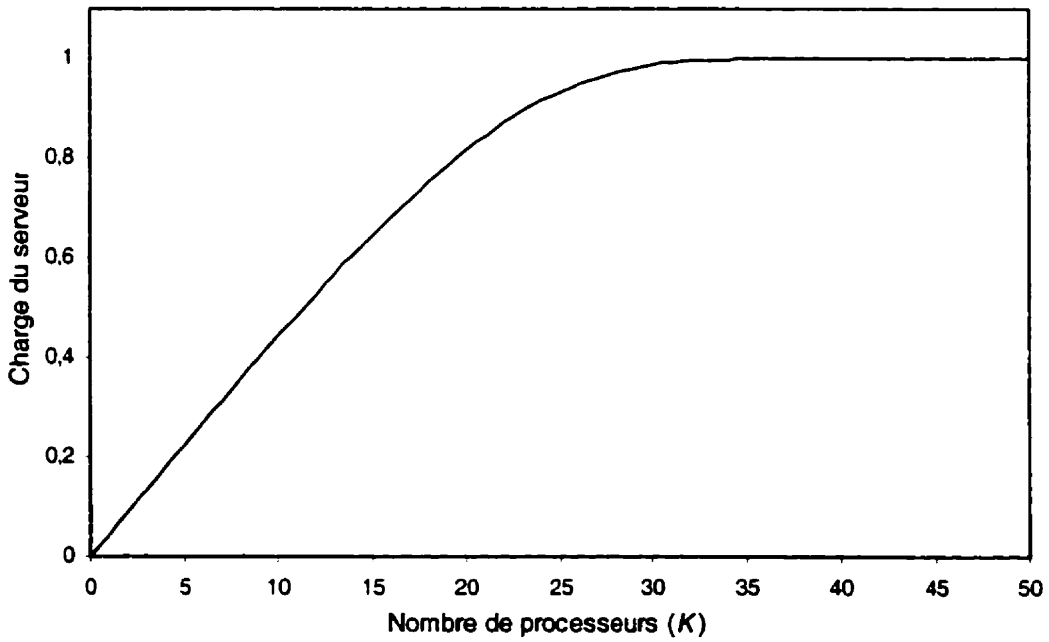

Figure 3.8 Charge du serveur de ressource partagée en fonction du nombre de processeurs ($u = 0{,}05$; $d_r = 1$)

CHAPITRE 4
MODÈLES ET RÉSULTATS DE SIMULATION

Dans ce chapitre, nous allons décrire les deux simulateurs qui ont été développés comme outils d'analyse supplémentaire de l'évolutivité des grappes d'ordinateurs. Ils ont été programmés en langage C++ et utilisent la librairie CSIM18. Nous allons d'abord donner un aperçu de cette librairie et préciser les raisons de son choix. Puis, nous présenterons le premier simulateur, les résultats obtenus et les conclusions que nous avons tirées. Par la suite, nous développerons un second simulateur qui représente un système de fichiers réparti dont l'architecture s'inspire du « wide striping » présenté au chapitre 2. Ce simulateur a été calibré à l'aide de mesures effectuées sur un système réel : les simulations nous ont alors permis de mesurer les performances du système actuel et d'analyser ses limites. Enfin, nous avons utilisé le simulateur pour valider une nouvelle technique de distribution redondante des données assurant l'évolutivité du système sans interruption de service.

4.1 Aperçu de CSIM 18

CSIM18 est une librairie commerciale, distribuée par Mesquite Software : c'est une librairie de simulation, orientée processus et événement, destinée à être utilisée dans des programmes en langage C ou C++ [CSIM18]. Elle se présente sous la forme de classes et procédures qui fournissent toutes les fonctionnalités liées à la simulation. La liste des classes disponibles est la suivante :

- Processus (Process) : Ce sont les entités qui effectuent des requêtes pour exploiter des ressources et attendent des événements. Elles s'apparentent à un processus UNIX, mais limité au cadre du simulateur.
- Ressources (Facility) : Elles sont composées de leur file d'attente et d'un ou de plusieurs serveurs de service.
- Réservoirs (Storage) : Ce sont des ressources qui peuvent être partiellement affectées à des processus.
- Évènements (Event) : Ils servent à synchroniser les processus entre-eux.
- Boite aux lettres (Mailbox) : Ils permettent aux processus de communiquer entre-eux

À cette liste, il faut rajouter des fonctions mathématiques et diverses structures utiles aux mesures et à la production de statistiques.

Nous avons donc choisi CSIM18 car il fournit tous les objets nécessaires à notre simulateur, tout en étant relativement simple pour que la prise en main soit rapide. CSIM a déjà été utilisé pour développer un simulateur de disques durs RAID dont les résultats sont très proches de la réalité [Ganger98]. Grâce à ses différentes classes, CSIM facilite également la transition du modèle analytique au modèle de simulation. Enfin, le langage de programmation C++ permet d'avoir une bonne souplesse dans l'implémentation. Les deux simulateurs ont été développés avec Borland C++ 5.0, sous Windows 98. Dans la suite de ce chapitre, nous allons détailler leurs architectures.

4.2 Modèle du premier simulateur

Notre premier simulateur se veut simple : son but est de simuler notre modèle analytique, puis d'étendre l'étude théorique par des résultats de simulation. Il est composé d'une classe unique qui a été rajoutée aux classes de CSIM18 : la classe « Processor ». Cette classe définit les objets processeurs de notre modèle analytique. Le programme principal peut se décomposer en trois parties. La première se charge de

l'initialisation de la simulation. Ainsi, elle propose à l'utilisateur le choix du mode parmi quatre possibilités :

- un temps de service et un délai constant ;
- un temps de service variant linéairement avec le nombre de processeurs et un délai constant ;
- un temps de service constant et un délai variant linéairement avec le nombre de processeurs ;
- un temps de service et un délai variant tous deux linéairement avec le nombre de processeurs.

Les coefficients constants du temps de service et du délai doivent être définis dans le fichier d'entrée *input.txt* dont le contenu est indiqué à la Figure 4.1. Dans le cas d'une variation linéaire, les coefficients lus dans le fichier d'entrée seront également pris en compte.

Temps de simulation
Temps moyen entre deux requêtes d'un processeur
Temps moyen de service
Coefficient de variation linéaire du temps de service moyen
Temps moyen du délai
Coefficient de variation linéaire du temps moyen du délai

Figure 4.1 Contenu du fichier de configuration « input.txt »

Le programme demandera alors à l'utilisateur de saisir le nombre de processeurs maximum N désiré. Puisque nous voulons étudier l'évolutivité du système, N simulations seront en fait effectuées, en faisant varier le nombre de processeurs de 1 à N avec un pas de 1. Le nombre maximum de processeurs est fixé à 200, mais il est facile d'augmenter cette limite en modifiant la taille du tableau des processeurs et en augmentant le nombre de processus supporté par CSIM.

Une fois que l'initialisation est terminée, la simulation débute. Le programme initialise et crée la ressource partagée et le nombre de processeurs nécessaires. Puis, chaque processeur est rendu actif par la commande *start()* qui démarre un nouveau processus par processeur. Ainsi, chaque processeur fonctionne en parallèle comme s'ils étaient indépendants; il recueille des statistiques sur son activité par lui-même pendant la simulation. Une fois la simulation terminée (quand le temps de fin de simulation est égal au temps d'horloge), la collecte des informations est donc simplifiée : le programme écrit les statistiques de chaque processeur dans le fichier *processors.txt* et les données recueillies par CSIM18 sur la ressource partagée sont écrites dans le fichier *modele0.txt*.

Comme le montre la Figure 4.2, les résultats obtenus sont très proches de ceux obtenus avec notre modèle analytique. Après avoir validé ce simulateur, nous l'avons utilisé pour étendre notre étude théorique du chapitre 3. Ainsi, nous avons envisagé le cas d'une ressource partagée dont la capacité de service se dégrade lorsque le nombre de

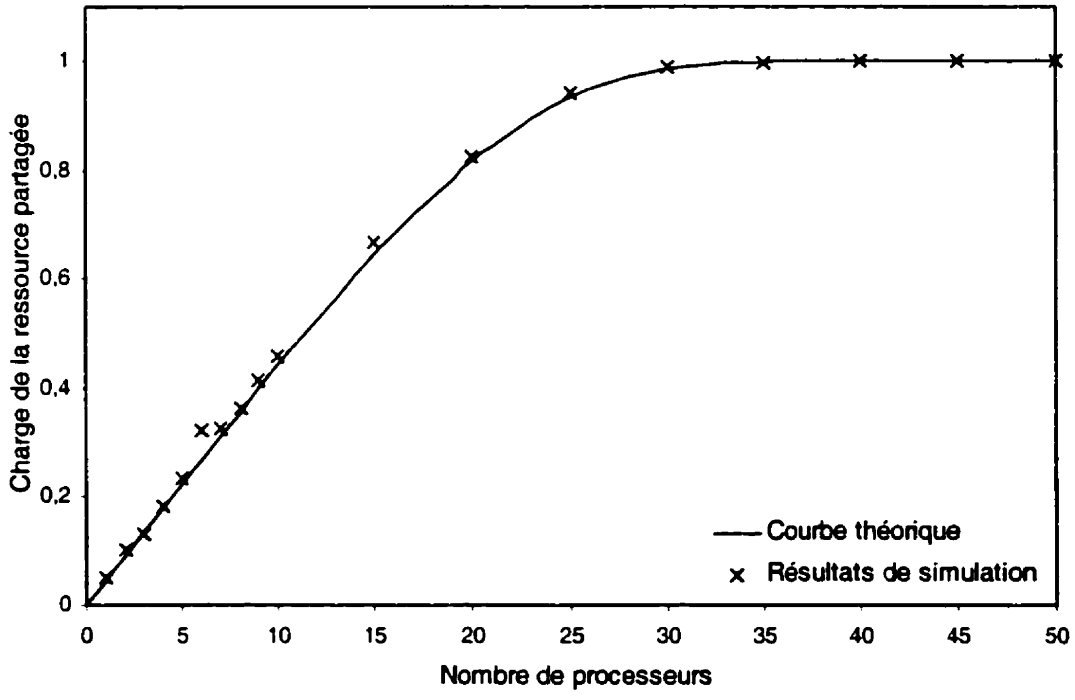

Figure 4.2 Résultats de simulation : charge du serveur de ressource partagée en fonction du nombre de processeurs ($u = 0{,}05$; $d_r = 1$)

processeurs augmente. À la Figure 4.3, nous constatons une dégradation des performances du système parallèle lorsque le nombre de nœuds dans la grappe augmente. C'est une situation que l'on veut à tout prix éviter puisqu'elle revient à augmenter les ressources du système parallèle pour en dégrader les performances globales. Dans la réalité, ce phénomène dépend de l'architecture du système. Par exemple, si les processeurs sont reliés par un bus Ethernet, la bande passante utile du réseau se dégrade lorsqu'il y a trop de nœuds, à cause des collisions entre les paquets Ethernet et les retransmissions. Dans le cas de nœuds reliés par un commutateur, le même phénomène se produit, mais pour un nombre de processeurs plus important. En effet, l'utilisation d'un commutateur évite un grand nombre de collisions, mais son cœur de chaîne a tout de même des performances limitées.

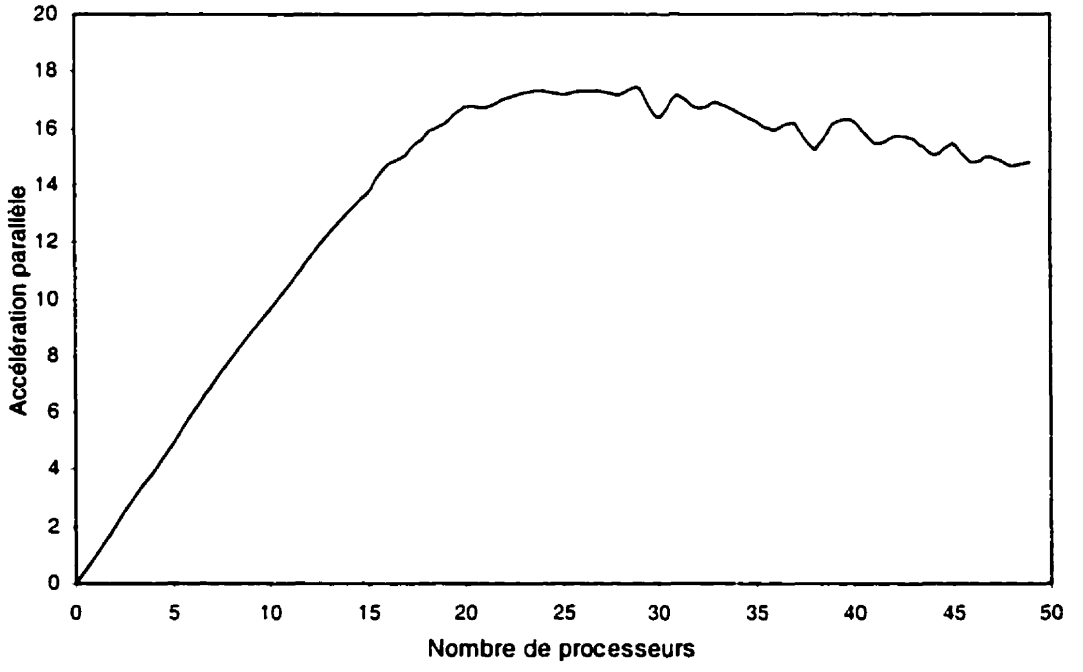

Figure 4.3 Variation de l'accélération parallèle en fonction du nombre de processeurs : $u = 0.05*(1+0.01*K)$ et $d_r = 1$

Ces résultats sur les systèmes parallèles ont orienté la suite de notre recherche. En effet, pour améliorer l'évolutivité des grappes d'ordinateurs, il y a deux approches possibles. La première consiste à supprimer les phénomènes qui limitent

l'évolutivité : à la Figure 4.3, cela revient à améliorer le comportement de la grappe quand le nombre de processeurs est supérieur à 20. Mais nous pensons que ces limitations sont fortement liées au matériel utilisé et elles constituent donc une contrainte inhérente aux grappes d'ordinateurs. L'autre approche consiste à mettre en place un système qui se maintient toujours dans la zone d'évolutivité quasi idéale : à la Figure 4.3, l'évolutivité est quasi linéaire pour un nombre de processeurs inférieur à 20. Nous allons donc mettre en application cette approche pour proposer des améliorations aux grappes d'ordinateurs. De plus, nous allons nous limiter à un problème bien précis : l'accès à des données partagées.

4.3 Simulateur de système de fichiers réparti

Dans la suite de nos travaux, nous nous sommes concentrés sur l'évolutivité des systèmes de fichiers parallèles pour grappes d'ordinateurs. Nous avons déjà établi au chapitre 2 qu'une architecture du type « wide striping » est particulièrement adaptée à la lecture séquentielle de fichiers volumineux. Cette architecture reprend l'idée du RAID 0 selon laquelle un fichier est découpé en segments qui sont répartis sur plusieurs disques durs [Massiglia2000]. Ainsi, lors de la lecture d'un fichier, les disques vont collaborer entre eux, ce qui permet d'améliorer les performances générales. La différence majeure avec un système RAID se trouve au niveau de la transmission des données, qui a lieu par l'intermédiaire d'un réseau Ethernet et de gestionnaires implémentés au niveau logiciel. Pour notre recherche, nous nous sommes basés sur PVFS (Parallel Virtual File System), un système de fichiers développé à l'Université de Clemson par P. Carns et W. Ligon [Carns2000], qui utilise le « wide striping » sur des grappes Linux. En prenant PVFS comme point de départ, nous allons utiliser ce simulateur dans le but de tester des améliorations que l'on pourrait apporter aux systèmes de fichiers parallèles actuels. Linux est déjà utilisé dans les grappes d'ordinateurs et possède de bonnes performances [Luecke2000, Brightwell99]. D'autre part, le code de Linux étant ouvert, il est un système d'exploitation prometteur et polyvalent qui est le support de nombreuses recherches sur les systèmes parallèles. Enfin PVFS supporte les API suivantes : une API

PVFS native, l'API UNIX/POSIX et l'API MPI-IO [Gropp99, MPI97]. Ces APIs sont souvent utilisées dans les systèmes parallèles, mais ne feront pas l'objet de développement supplémentaire puisque nous avons choisi de ne pas faire d'implémentation dans notre recherche.

4.3.1 Architecture matérielle et distribution de données

Pour illustrer les problèmes d'évolutivité que l'on peut rencontrer, nous allons considérer un serveur de vidéo sur demande (VoD), une des principales applications des systèmes de fichiers répartis utilisant le « wide striping ». En effet, les clients visionnent des séquences vidéo en lisant de manière séquentielle des fichiers de grande taille. Le but d'un tel système est donc de maximiser le nombre de clients pouvant être servis simultanément : la performance du système est évaluée par la bande passante totale délivrée. De plus, la capacité totale de stockage doit être élevée puisque les fichiers multimédias sont très volumineux. Du point de vue de l'évolutivité, un serveur de vidéo sur demande doit s'adapter au nombre de clients et à la quantité de données accessible. On peut imaginer que, lors de la mise en place du système, ses capacités sont réduites, pour augmenter progressivement en suivant la demande.

Avant de détailler les problèmes d'évolutivité à résoudre, nous allons décrire l'architecture de PVFS sur laquelle est basé notre simulateur. Les clients extérieurs au système accèdent aux données par des points d'entrée qui sont eux-mêmes des clients pour le système de fichiers, tel que représentés à la Figure 4.4. Dans la suite de notre étude, nous ne considérerons que les clients locaux en faisant l'hypothèse qu'un algorithme s'occupe de la répartition des requêtes de clients extérieurs. Lorsque le système est utilisé à pleine capacité, les clients internes sont ainsi toujours actifs. Le nombre de clients internes détermine le nombre d'accès simultanés au serveur de données. En plus du client interne à la grappe, la lecture d'un fichier fait intervenir tous les éléments de la grappe représentés à la Figure 4.4 :

- le réseau reliant le client au reste de la grappe ;
- le réseau reliant chacun des nœuds de la grappe entre eux ;

- les N nœuds de la grappe ;
- un nœud particulier de la grappe appelé « Management Daemon » (MNG) qui est le gestionnaire du système de fichiers.

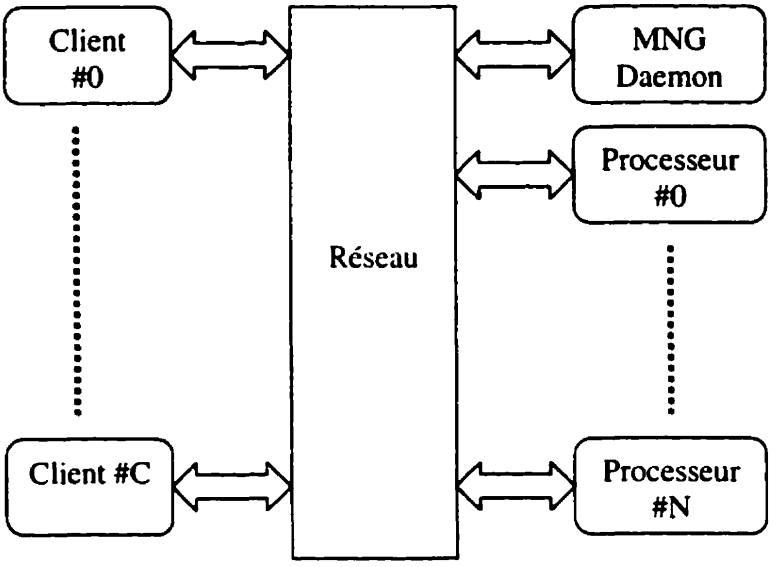

Figure 4.4 Architecture matérielle du système de fichiers

Chaque nœud de la grappe est en fait composé d'une unité de calcul, d'un disque dur pour le stockage des données et d'une interface réseau. Le « wide striping » de PVFS est identique à celui détaillé dans le chapitre 2 : un fichier est découpé en blocs de taille fixe et ces blocs sont répartis de manière séquentielle sur l'ensemble des nœuds de la grappe. La profondeur du « striping » est définie par la taille d'un bloc de donnée (valeur par défaut de 64 Ko). Selon la Figure 4.5, PVFS adresse les données en considérant directement les blocs de données. Le disque virtuel correspondant est un disque composé des blocs de données réordonnées. Ainsi, le k^{em} bloc du disque virtuel peut être repéré dans la grappe par des coordonnées (i,j), où i désigne le numéro du « stripe » et j le numéro du disque. Par exemple, le bloc 10 a pour coordonnées (3,1).

Nous allons maintenant énumérer les situations d'évolutivité qui peuvent être rencontrées :

- 1er cas : évolutivité de la capacité de stockage. Le nombre de clients pouvant être servi simultanément ne varie pas, mais la taille de l'ensemble des données accessibles par ces clients doit augmenter. Dans ce cas, il suffit d'accroître la capacité de stockage de chaque nœud. Cette opération est plus ou moins coûteuse en fonction de la taille de la grappe : au minimum, il faudra rajouter autant de disques qu'il y a de nœuds.
- 2e cas : augmentation du nombre de clients pouvant être servi simultanément. C'est la situation qui nous semble la plus complexe à résoudre. En effet, pour augmenter la capacité de service, il faut ajouter des nœuds. Une redistribution des données doit alors avoir lieu.

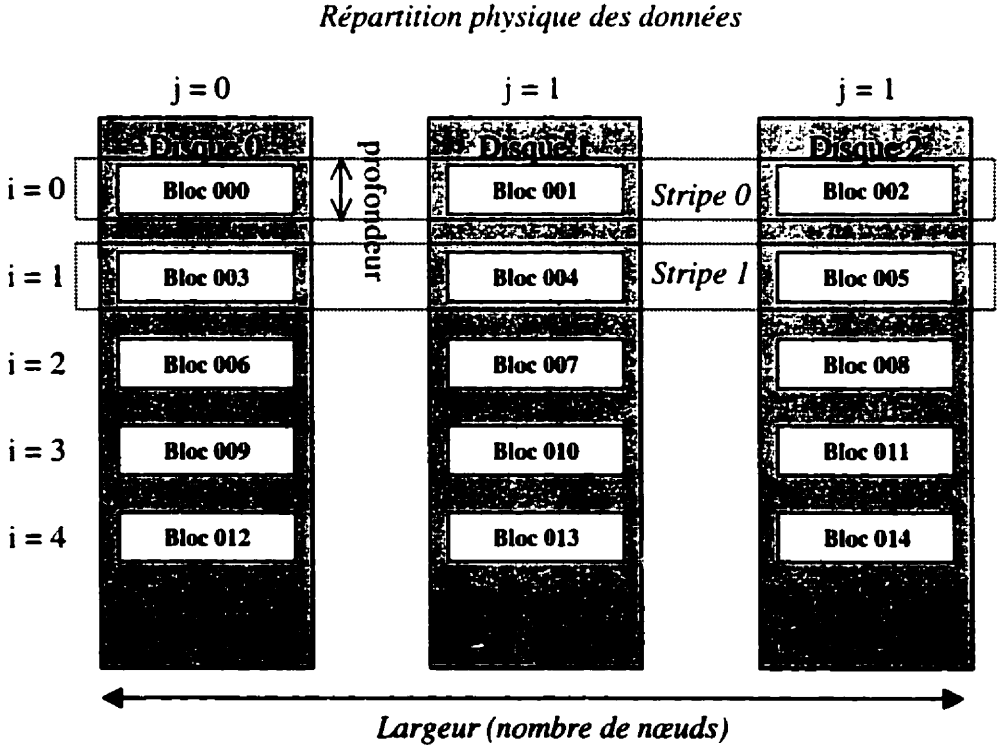

Figure 4.5 Répartition des données avec PVFS

- 3^{e} cas : augmentation simultanée de la capacité de stockage et du nombre de clients pouvant être servi simultanément. Dans cette situation, la solution peut être soit de combiner les solutions des deux premiers cas, soit de mettre en place une nouvelle grappe qui supportera une partie des requêtes des clients et contiendra les nouveaux fichiers.

Pour les différents cas, l'objectif est de faire évoluer le système en minimisant les coûts de mise à jour : il faut donc utiliser les ressources déjà existantes. Cela ne doit pas non plus entraîner d'arrêt ou de perturbation trop importante du service. Enfin, il faut rester dans la zone où l'évolutivité est quasi idéale. Notre recherche a donc pour but de proposer des mécanismes qui permettront de faire évoluer le système tout en répondant à l'ensemble de ces contraintes. Pour ce faire, nous avons développé un simulateur de système de fichiers en nous appuyant sur l'architecture de PVFS.

4.3.2 Architecture et configuration du simulateur

Les mécanismes de lecture d'un fichier sont présentés à la Figure 4.6. Afin de modéliser les différentes composantes du système de fichiers et de la grappe d'ordinateurs, six classes ont été utilisées dans notre simulateur :

- les instances de la classe *IODaemon* sont les entités qui gèrent la transmission des données entre un client et un nœud de la grappe ;
- la classe *Processor* permet de représenter les nœuds de la grappe ;
- la classe *MNGDaemon* correspond au processus qui gère le système de fichiers lui-même, il maintient ainsi les informations de localisation des fichiers, le nombre et le rôle des processeurs de la grappe ;
- la classe *Client* : l'ensemble des instances de cette classe représente les clients qui accèdent simultanément aux fichiers ;
- la classe *Request* est la structure de données d'une requête d'accès aux données ;
- La classe *Datafile* est la structure de données qui contient les informations sur la localisation des blocs d'un fichier.

Pour lire un fichier, un client doit procéder de la manière suivante :

- Il communique avec le MNG Daemon pour avoir des informations sur le fichier (taille, localisation) qu'il veut lire et savoir ou trouver les blocs correspondants.
- Avec toutes ces informations, il initialise un IO Daemons pour chacun des nœuds sur lesquels des blocs seront lus. Ces IO Daemons possèdent alors une liste des blocs à lire sur le nœud qui leur est attribué.
- Enfin, le client démarre les IO Daemons et se met en mode réception : les IO Daemons transmettent alors les blocs aux clients de manière séquentielle.

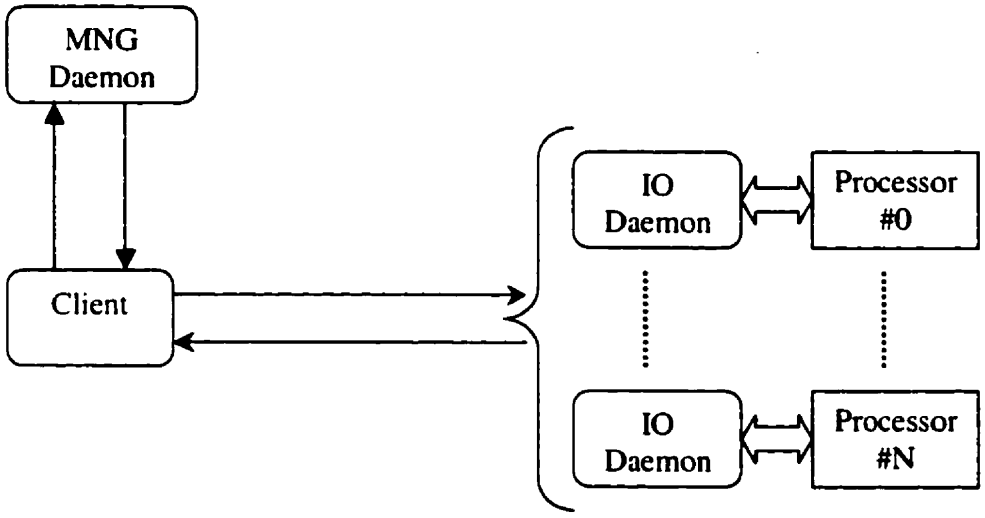

Figure 4.6 Description de la lecture d'un fichier

La plupart des paramètres de configuration sont lus dans le fichier de configuration *input.txt* dont le contenu est présenté à la Figure 4.7. Une option de compilation permet de choisir un mode manuel dans lequel le programme demande à l'usager le nombre de nœuds de la grappe, le nombre de nœuds additionnels et le nombre de clients. Les valeurs correspondantes dans le fichier de configuration sont alors ignorées. On peut remarquer que le MNG Daemon a un rôle central dans cette architecture et constitue ainsi un goulot d'étranglement potentiel. Un des objectifs de la prochaine version de PVFS est de permettre plusieurs MNG dans la même grappe

d'ordinateur : cela permet de s'affranchir de cette limite, mais également d'introduire une redondance et donc d'améliorer la tolérance aux pannes.

Nombre de nœuds de la grappe (nœuds d'origines uniquement)
Nombre de nœuds additionnels
Nombre de clients
Temps de simulation
Temps moyen d'inter-arrivée des requêtes clients (0 dans nos simulations)
Taille d'un bloc de donnée

Figure 4.7 Contenu du fichier de configuration « *input.txt* »

Le fichier *fileslist.txt* contient les informations sur les fichiers stockés dans la grappe : la première entrée indique le nombre de fichiers, puis en suivant l'ordre dans lequel les fichiers sont localisés dans le disque virtuel, chaque entrée donne sa taille. Dans le cas d'ajout de nœuds durant la simulation, le fichier *online.txt* doit contenir en première entrée le nombre de nœuds qui seront ajoutés puis, pour chaque nœud, le temps d'horloge auquel aura lieu l'ajout.

CSIM génère automatiquement un compte rendu de simulation dans le fichier *modele1.txt*. D'autres statistiques sont disponibles dans les fichiers texte suivants :

- bandwidth.txt : Ce fichier constitue un historique des simulations effectuées. La dernière entrée donne la bande passante moyenne du système. Selon le mode choisi à la compilation, le nombre de clients ou de nœuds additionnels est indiqué ;
- clients.txt : dans l'ordre chronologique, il contient les requêtes client complétées, le nombre de blocs et le numéro du fichier correspondant, ainsi que le temps qui a été nécessaire ;
- c_bandwidth.txt : mesure la bande passante consommée par les clients toutes les 10 secondes ;

- rt_bandwidth.txt : mesure la bande passante totale du système toutes les 10 secondes.

4.3.3 Calibrage du simulateur

Pour régler les paramètres internes du simulateur, nous allons utiliser des résultats de tests effectués par P. Carns et W. Ligon [Carns2000] sous PVFS. Le matériel utilisé est une grappe de PC reliés par un commutateur Ethernet à 100 Mbit/s. Les mesures ont été prises pour des tailles de grappe différentes et en faisant varier le nombre de clients : les clients lisent simultanément un fichier de taille fixée. Le Tableau 4.1 présente les différentes configurations utilisées.

Les mesures montrent que la bande passante du réseau est la ressource qui limite les performances globales. C'est donc le paramètre de notre simulateur qui sera ajusté avec le plus de soins. Les figures 4.8, 4.9 et 4.10 montrent que le système possède une bonne évolutivité tant que le nombre de clients ne dépasse pas le nombre de nœuds de la grappe : passé ce seuil, les requêtes des clients saturent la bande passante des liens de communication qui les relient aux différents nœuds de la grappe. Dans la zone d'évolutivité linéaire, la bande passante totale augmente d'environ 11 Mo/s par nœud supplémentaire. C'est aussi le domaine dans lequel notre simulateur reproduit le plus fidèlement le comportement du système réel. Quand la charge du réseau devient trop élevée, notre simulateur s'écarte des résultats mesurés car il ne prend pas en compte les phénomènes liés à une forte contention. Ceci n'aura pas d'incidence sur la suite de notre recherche puisque c'est le domaine d'évolutivité linéaire dans lequel le simulateur est valide qui nous intéresse.

Les figures 4.11 et 4.12 montrent que, pour des tailles de 24 et 32 nœuds, de nouvelles limites du réseau apparaissent : c'est la capacité du commutateur Ethernet qui constitue le goulot d'étranglement. Étant donné l'architecture matérielle utilisée, le MNG Daemon ne constitue pas un goulot d'étranglement : les performances du réseau empêchent de se retrouver dans une telle situation. En revanche, il est probable qu'en utilisant un réseau plus performant, le MNG Daemon peut devenir la ressource limitant

les performances globales : c'est pourquoi la possibilité d'avoir plusieurs MNG Daemon est un des objectifs pour la prochaine version de PVFS.

Tableau 4.1 Paramètres des bancs de test PVFS

Taille de la grappe	Nombre de clients	Taille d'un fichier
4 nœuds	0-48	8 Mo
8 nœuds	0-48	16 Mo
16 nœuds	0-45	32 Mo
24 nœuds	0-29	48 Mo
32 nœuds	0-28	64 Mo

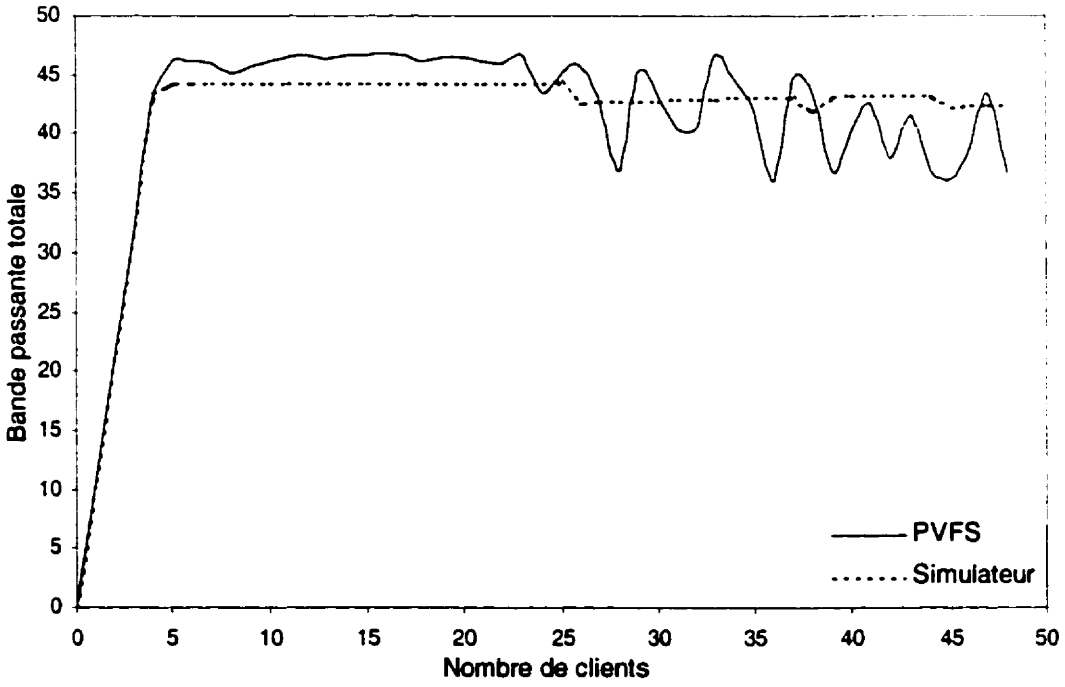

Figure 4.9 Résultats de simulation pour une grappe de 4 nœuds

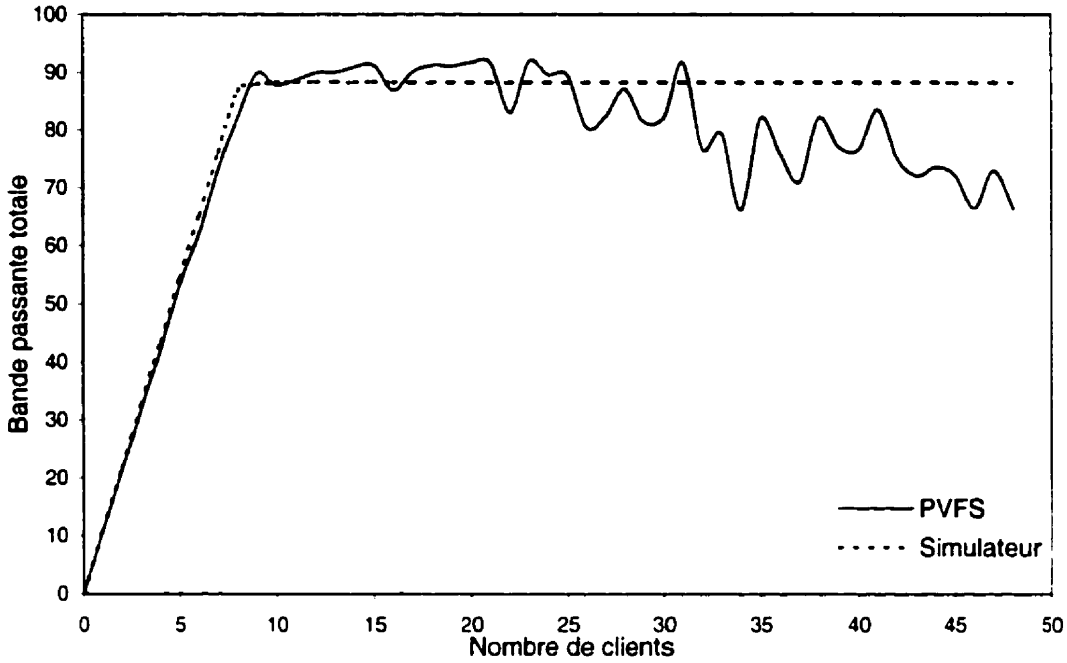

Figure 4.9 Résultats de simulation pour une grappe de 8 nœuds

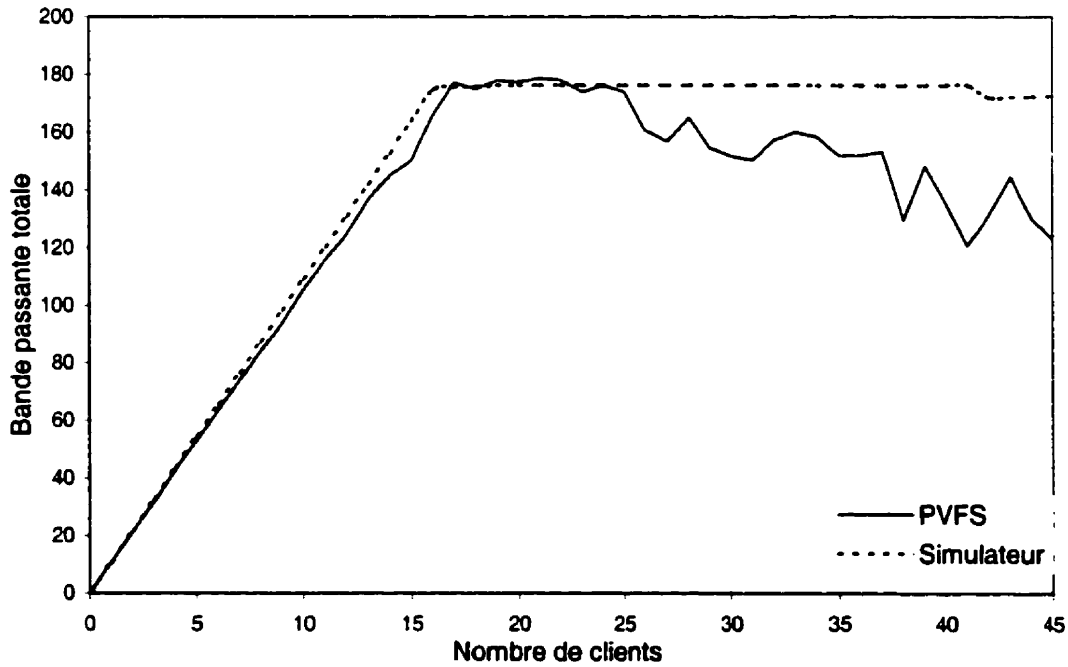

Figure 4.10 Résultats de simulation pour une grappe de 16 nœuds

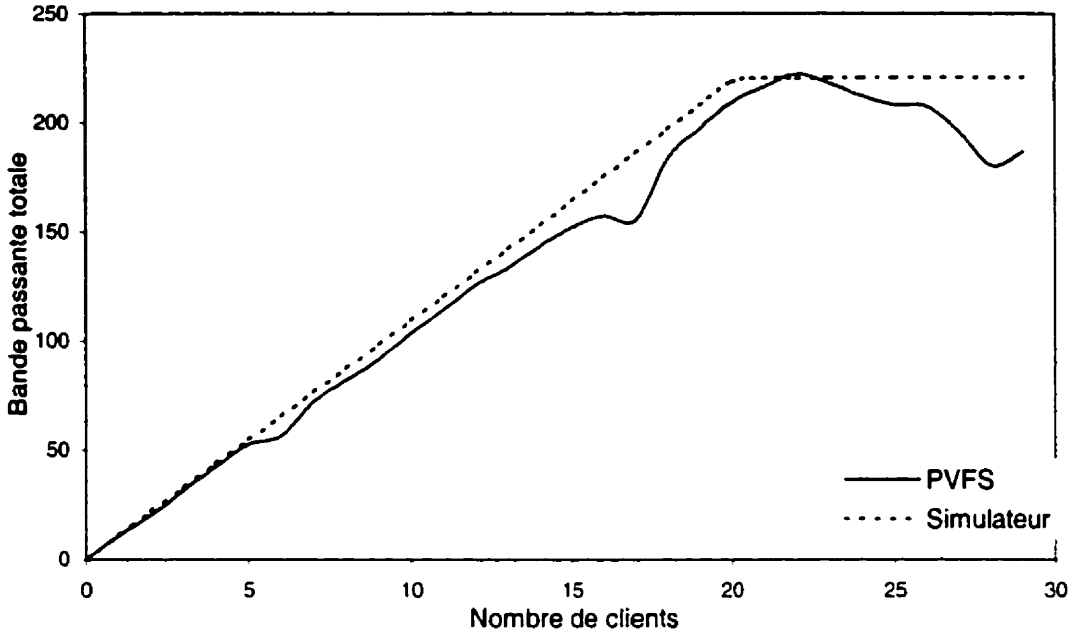

Figure 4.11 Résultats de simulation pour une grappe de 24 nœuds

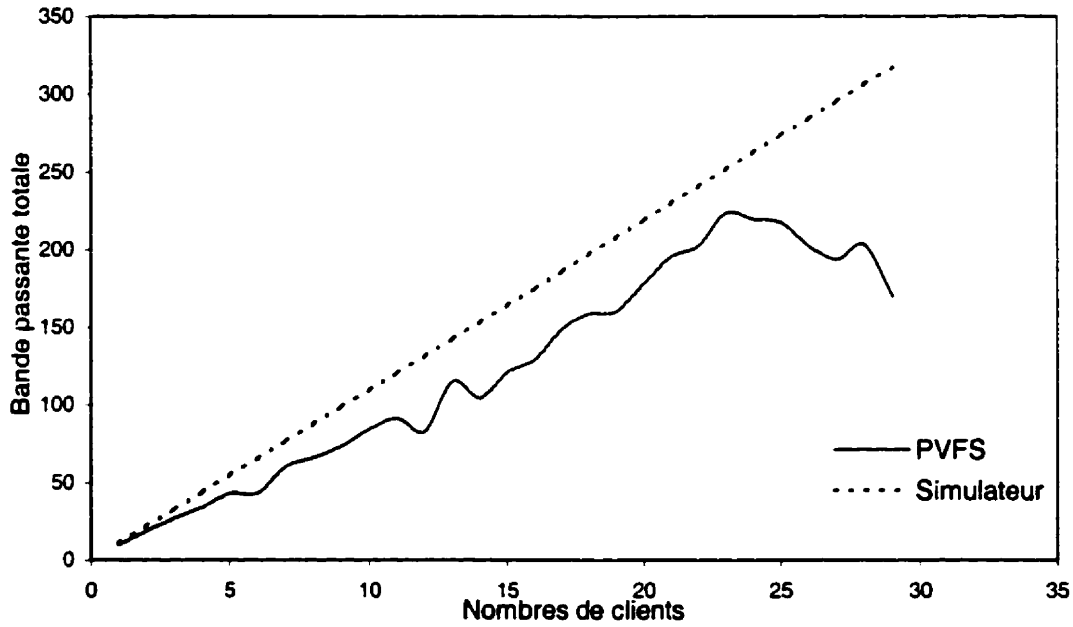

Figure 4.12 Résultats de simulation pour une grappe de 32 nœuds

4.3.4 Distribution des données

Comme prévu, la principale limite d'évolutivité est la capacité du réseau. Nos résultats de simulation confirment la très bonne évolutivité de PVFS dans la partie linéaire. Nous allons donc proposer une technique de placement des données qui permet d'augmenter facilement le nombre de clients pouvant être servis simultanément en ajoutant progressivement des nœuds, tout en restant dans la zone d'évolutivité linéaire. Un des objectifs est de diminuer autant que possible la granularité de cette évolutivité, le cas idéal étant une granularité d'un nœud. De cette manière, les performances du système pourront toujours être optimales par rapport aux besoins. La distribution des données doit également être redondante afin d'assurer la haute disponibilité du système et la tolérance aux pannes. Nous nous imposons cette contrainte, mais nous n'allons pas étudier les schémas de redondance en tant que tel car ils sortent du cadre de cette recherche.

Nous allons ainsi étudier une extension du « chained declustering » [Dusseau99]. Jusqu'à présent, ces méthodes de répartition redondante des données n'ont été utilisées que pour garantir un service continu en cas de pannes matérielles. Cependant, nous voulons montrer qu'il est également possible de les utiliser pour rendre les systèmes facilement évolutifs. Dans des applications comme les serveurs de vidéo sur demande, la combinaison de ces deux propriétés est primordiale. Nous avons choisi d'utiliser un algorithme statique de lecture des données. Un tel algorithme est plus difficile à optimiser qu'une version dynamique, car il ne prend pas en compte l'état courant du système. Cependant, une politique statique est plus simple à implémenter et minimise les coûts de gestion des transferts. De plus, de nombreuses méthodes dynamiques existent déjà et nous pensons qu'elles peuvent être adaptées à notre système (le « graduated declustering » [Dusseau99] par exemple).

Désignons par N le nombre de nœuds d'entrée/sortie d'origine, c'est à dire les nœuds qui constituent la grappe avant toute évolution et par M le nombre de nœuds d'entrée/sortie additionnels destinés à augmenter le nombre de clients que le système peu servir simultanément. Nous numérotons les nœuds d'origine de 0 à $N-1$ et les nœuds

additionnels de 0 à *M-1*. Notre solution consiste à augmenter les performances du système d'origine en ajoutant progressivement des nœuds jusqu'à atteindre la limite $M = N$. Lorsque la taille de la grappe a doublé, les nœuds d'origine et additionnels sont séparés pour former deux grappes indépendantes : cette solution nous permet de rester dans la zone d'évolutivité idéale en multipliant le nombre de grappes et en limitant la taille de chaque grappe. Un algorithme de répartition de la charge distribue alors les requêtes des clients sur les deux systèmes, ce qui permet de se rapprocher de l'évolutivité idéale.

Au maximum, un même bloc de donnée du disque virtuel pourra être dupliqué quatre fois dans la même grappe. Pour différencier les blocs de données en fonction de leur type, nous utilisons les notations suivantes :

- X désigne un bloc de donnée se trouvant sur un des nœuds d'origine et correspondant aux données primaires ;
- Y désigne un bloc de donnée se trouvant sur un des nœuds d'origine de la grappe et correspondant aux données redondantes ;
- U désigne un bloc de donnée se trouvant sur un nœud additionnel de la grappe et correspondant aux données primaires (i.e. un bloc n'étant pas un duplicata) ;
- V désigne un bloc de donnée se trouvant sur un nœud additionnel de la grappe et correspondant aux données redondantes.

Nous reprenons la notation introduite à la Figure 4.5 : un bloc est représenté par son numéro k dans le disque virtuel.

- X_k (respectivement Y_k, U_k, V_k) désigne alors les coordonnées du bloc k lorsqu'il est de type X (respectivement Y, U, V) ;
- $x(i,j)$ (respectivement $y(i,j)$) désigne le bloc de type X (resp. Y) qui se trouve sur le « stripe » i et le nœud d'origine j ;
- $u(i,j)$ (respectivement $v(i,j)$) désigne le bloc de type U (resp. V) qui se trouve sur le « stripe » i et le nœud additionnel j.

On peut donc écrire :

$$x(i, j) = x(X_k) \Leftrightarrow (i, j) = X_k \quad , avec\ 0 \le i \quad et \quad 0 \le j \le (N-1)$$

Les correspondances sont les même pour les autres types de blocs. Toutes les démonstrations concernant cet algorithme sont présentées en annexe [Annexe A].

1) Les blocs de type X (nœuds d'origine, données primaires) :

Ils sont distribués selon un « stripe » séquentiel simple. On bénéficie ainsi des avantages du « wide striping » évoqués précédemment. On peut formaliser les positions représentées à la Figure 4.13 (lignes notées X) par les formules suivantes :

$$X_k = \left(\frac{k - k[N]}{N}, k[N] \right)$$

$$x(i, j) = x(X_k) \Rightarrow k = i * N + j$$

2) Les blocs de type Y (nœuds d'origine, données redondantes) :

Distribution de type « multi-chained declustering » (redondance des blocs X). Le principe de la distribution que nous proposons est de répartir les blocs de redondance tel que les données primaires d'un disque soient uniformément réparties sur l'ensemble des autres disques. Cela revient à décaler le « stripe » séquentiel des blocs de redondance d'un bloc à chaque « stripe. » Cependant, il faut éliminer le « stripe » pour lequel les blocs primaires et les blocs de redondance sont sur les mêmes disques. Le décalage (D) d'un bloc Y_k par rapport à un bloc X_k s'exprime donc par la formule suivante :

$$D = (i[N-1] + 1)$$

On peut vérifier que D est bien périodique de période (N-1) et n'est jamais nul. Nous pouvons alors formaliser cette distribution :

$$y(i, j) = y(Y_k) \Leftrightarrow Y_k = (i, j) \text{ et } y(i, j) = x\left(i, \{j - D\}[N]\right)$$

$$\Rightarrow y(i, j) = x\left(i, \{j - (i[N-1] + 1)\}[N]\right)$$

$$\Rightarrow x(i, j) = y\left(i, \{j + i[N-1] + 1\}[N]\right)$$

Nous démontrons alors que :

$$Y_k = \left(\frac{k - k[N]}{N}, \left\{ k[N] + \left(\frac{k - k[N]}{k} \right) [N-1] + 1 \right\} [N] \right)$$

$$y(i, j) = y(Y_k) \Rightarrow \begin{cases} k = i * N + j - \left(i[N-1] + 1 \right) & si \quad j \geq i[N-1] + 1 \\ k = i * N + j - \left(i[N-1] + 1 \right) + N & si \quad j \leq i[N-1] \end{cases}$$

Cette distribution des données est représentée à la Figure 4.13 (lignes Y). On peut alors constater l'avantage d'un tel placement de la redondance : lorsqu'un nœud tombe en panne, sa charge est répartie sur l'ensemble des nœuds survivants. Par exemple, si le nœud 0 ne fonctionne plus, les blocs de données de type X qu'il contient sont répliqués sur tous les autres disques sous la forme de blocs de type Y (blocs en gras). Une telle répartition permet donc de ne pas perdre de données lorsqu'un nœud devient indisponible ou tombe en panne, mais également de maintenir une capacité de service optimale avec les nœuds survivants.

3) Les blocs de type U (nœuds additionnels, données primaires) :

C'est une distribution de type « extended multi-chained declustering » par rapport aux blocs X. Elle reprend l'idée de la distribution des blocs Y, mais les blocs sont répliqués sur des nœuds additionnels et il est possible d'avoir $X_k = U_k$. Le décalage s'exprime alors simplement comme suit :

$$D = i[N]$$

On peut alors formaliser cette distribution par les formules :

$$u(i, j) = u(U_k) \Leftrightarrow U_k = (i, j)$$

$$u(i, j) = x\left(i, \left\{ j + i[N] \right\} [N]\right)$$

$$x(i, j) = u\left(i, \left\{ j - i[N] \right\} [N]\right)$$

Ce qui nous permet de démontrer que :

$$U_k = \left(\frac{k - k[N]}{N}, \left\{ k[N] - \left(\frac{k - k[N]}{k} \right)[N] \right\}[N] \right)$$

$$u(i, j) = u(U_k) \Rightarrow \begin{cases} k = i * N + j + i[N] & si \quad j \leq N - i[N] - 1 \\ k = i * N + j + i[N] - N & si \quad j \geq N - i[N] \end{cases}$$

Cette distribution que nous proposons reprend l'idée du « multi-chained declustering », mais a pour but d'améliorer la bande passante totale du système en répliquant les blocs situés sur les nœuds d'origine. Ainsi, la lecture des blocs d'un même fichier peut avoir lieu sur l'ensemble des nœuds d'origine additionnels. Pour que la répartition de la charge soit idéale, il faut lire autant de blocs de données sur chacun des nœuds de la grappe. Chacun des nœuds supporte alors la même charge indépendamment de sa nature. Cette distribution nous permet également d'avoir une granularité de l'évolutivité idéale, puisque les performances de la grappe restent optimales indépendamment du nombre M de nœuds ajoutés à celle-ci.

4) Les blocs de type V (nœuds additionnels, données redondantes) :

Distribution de type « multi-chained declustering » par rapport aux blocs U. Cette distribution est équivalente à la distribution des blocs Y par rapport aux blocs X, c'est à dire que le décalage de V_k par rapport à U_k est le même que celui de Y_k par rapport à X_k. On peut donc écrire :

$$v(i, j) = v(Y_k) \Leftrightarrow V_k = (i, j)$$

$$v(i, j) = u\left(i, \{j - (i[N-1] + 1)\}[N]\right)$$

$$u(s, t) = v\left(s, \{t + s[N-1] + 1\}[N]\right)$$

or

$$u(i, j) = x\left(i, \{j + i[N]\}[N]\right)$$

$$x(p, q) = u\left(p, \{q - p[N]\}[N]\right)$$

d'où

$$v(i, j) = x\left(i, \left\langle \{j - (i[N-1]+1)\}[N] + i[N] \right\rangle [N]\right)$$

$$x(p, q) = v\left(p, \left\langle \{q - p[N])\}[N] + p[N-1] + 1 \right\rangle [N]\right)$$

On démontre alors que :

$$V_k = \left(\frac{k - k[N]}{N}, \left\langle \left\{ k[N] - \left(\frac{k - k[N]}{k} \right)[N] \right\}[N] + \left(\frac{k - k[N]}{k} \right)[N-1] + 1 \right\rangle [N] \right)$$

$$v(i, j) = v(V_k) \Rightarrow \begin{cases} k = i * N + j + i[N] - (i[N-1]+1) \\ \qquad si \quad i[N-1]+1-i[N] \le j \le N + i[N-1] - i[N] \\ k = i * N + j + i[N] - (i[N-1]+1+N) \\ \qquad si \quad i[N-1]+1+N-i[N] \le j \le 2N + i[N-1] - i[N] \\ k = i * N + j + i[N] + (N-1) - (i[N-1]+1) \\ \qquad si \quad i[N-1]-(N-1)-i[N] \le j \le i[N-1] - i[N] \\ k = i * N + j + i[N] - (i[N-1]+1) \\ \qquad si \quad i[N-1]+1-i[N] \le j \le N + i[N-1] - i[N] \end{cases}$$

À l'aide de ces formules, nous avons implémenté les algorithmes de placement des blocs de données lors de l'écriture d'un fichier. Les résultats de ces algorithmes sont présentés à la Figure 4.13. Chaque groupe de 2 lignes représente un « stripe » composé des données primaires et de la redondance correspondante. Les blocs X et Y se situent sur les nœuds de base de la grappe et les blocs U et V sur les nœuds additionnels. Nous avons représenté tous les nœuds additionnels possibles mais un système peut comporter un nombre de nœuds additionnels compris entre zéro et le nombre de nœuds du cluster d'origine.

6 nœuds d'origine *Nœuds supplémentaires*

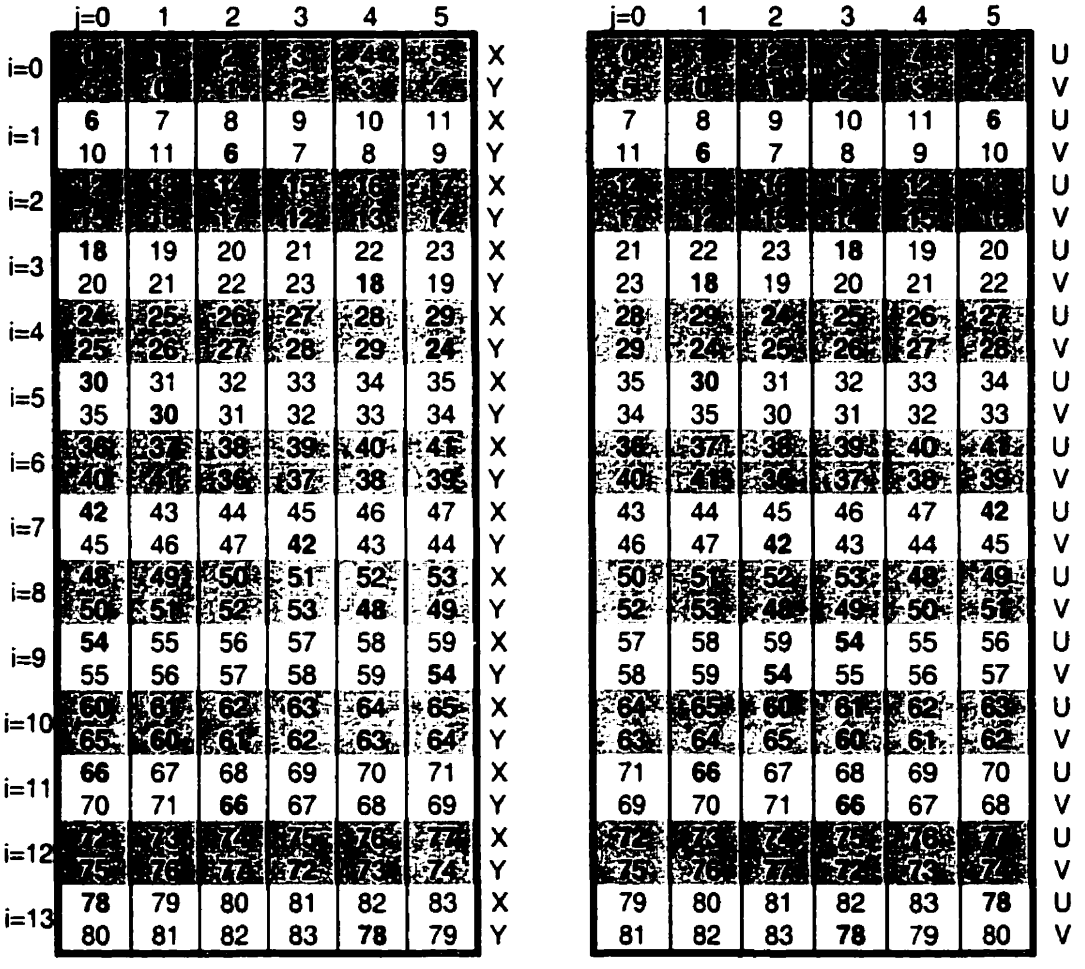

Figure 4.13 Distribution des données pour *N*=6

La distribution des données est telle qu'un nœud additionnel contient autant de blocs provenant de chaque nœud d'origine. Pour la lecture des données, il faut aussi que le nombre de blocs lus soit le même sur l'ensemble des nœuds. L'algorithme de lecture va donc faire intervenir les paramètres M et N. En remarquant que :

$x(i, j) = u(i, \{j - i[N]\}[N]) = u(i, e)$ où $e \equiv \{j - i[N]\}[N]$

on peut en déduire l'algorithme décrit à la Figure 4.14.

Initialisation : on veut lire le bloc k, $x(i, j) = x(X_k)$

Soit $e \equiv \{j - i[N]\}[N]$

Si $e < M$

Alors :

Si $(i - t)[M + N] = 0$ *pour* $0 \le t < M$, **alors** on lit $x(i, j)$.

Sinon, on lit $u(i, e)$ à la place de $x(i, j)$.

Fin si

Sinon on lit $x(i, j)$

Fin si

Figure 4.14 Algorithme de lecture

Cet algorithme répartit également les blocs à lire pour des groupes de $N*(N+M)$ blocs. On peut montrer qu'il est en fait nécessaire d'avoir un nombre suffisamment grand de blocs pour avoir de bonnes performances. Dans la pratique, ce n'est pas une limite car la taille des données dépasse largement ce seuil. La Figure 4.15 représente la lecture de blocs sur une grappe composée de 6 nœuds de base et 2 nœuds additionnels en reprenant le schématisme de la Figure 4.13. Cet algorithme combiné à notre méthode de distribution des données a de très bonnes performances : on peut démontrer que sur un nombre de requêtes suffisamment grand, la répartition de la charge est quasiment parfaite (le défaut de répartition est inférieur à 1%).

6 nœuds à l'origine

2 nœuds supplémentaires

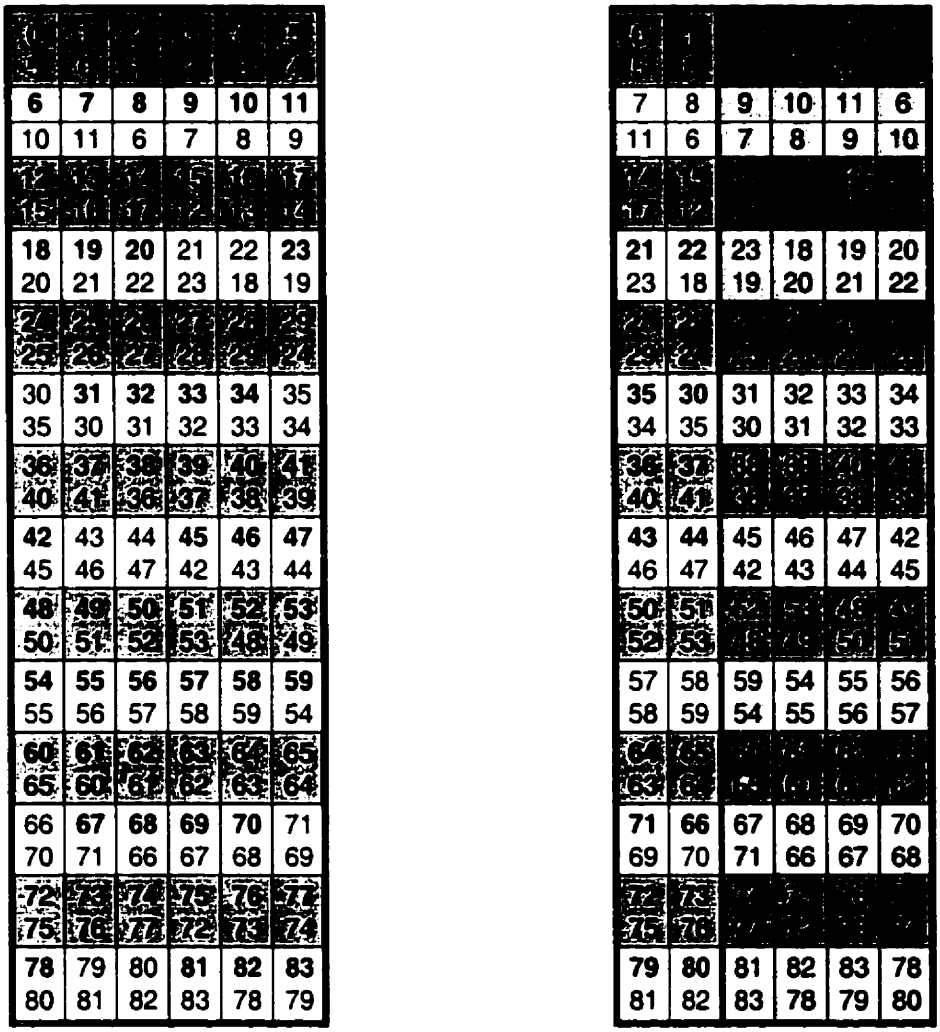

Figure 4.15 Exemple de lecture

4.4 Analyse des résultats

Dans cette section, nous allons présenter et analyser les résultats obtenus avec le simulateur de système de fichiers décrit précédemment. Ce simulateur nous a permis de valider la méthode de répartition des données que nous proposons et de mettre en place une procédure pour faire évoluer les performances du système de fichiers qui l'utilise.

4.4.1 Évolutivité de la capacité de service

Dans un premier temps, nous avons testé notre technique de répartition des données. Nous avons donc repris les configurations utilisées pour calibrer le simulateur et correspondant à son domaine de validité. La Figure 4.16 représente l'évolutivité d'une grappe de 8 nœuds. L'évolutivité du système de fichiers est proche du cas idéal : l'écart est inférieur à 1%. Ces résultats ont été obtenus pour 25 clients qui lisent simultanément des fichiers de 16 Mo. Le nombre de clients est toujours supérieur au nombre de nœuds

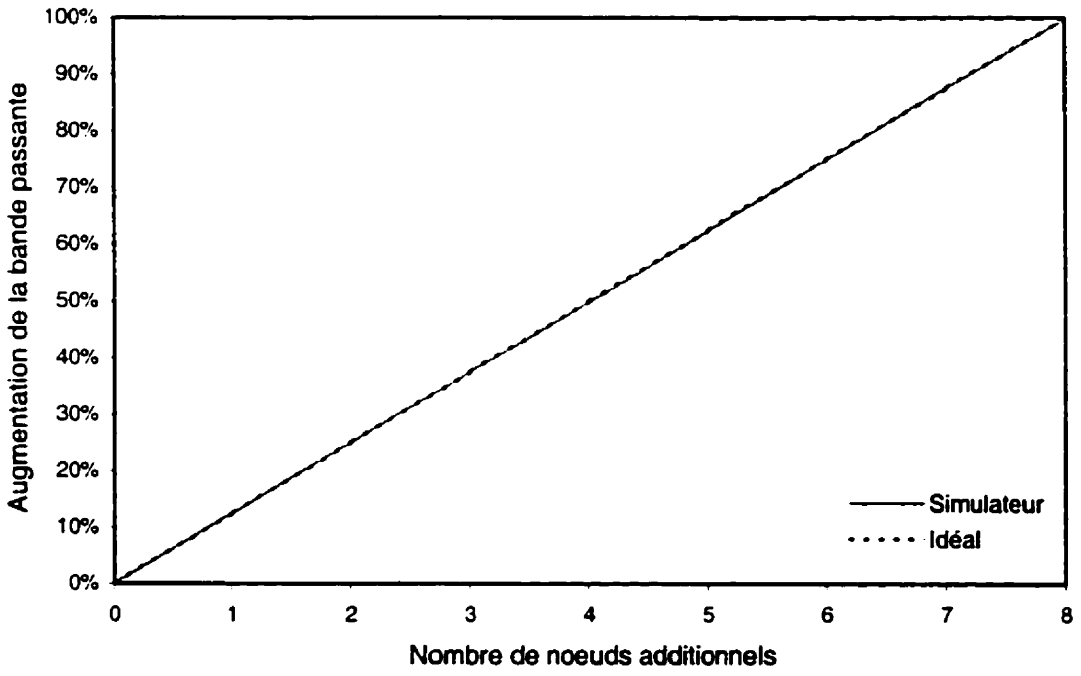

Figure 4.16 Évolutivité d'une grappe de 8 nœuds

de la grappe : on mesure ainsi la bande passante maximale que peut délivrer le système de fichiers. C'est le réseau qui limite la capacité de ce système : comme l'indique la Figure 4.16, notre solution se limite à 8 nœuds additionnels par grappe. Cela signifie qu'une même grappe contiendra au maximum 16 nœuds. Nous avons vu que la limite d'évolutivité se situe autour de 220 Mo/s, soit 20 nœuds. La solution consiste alors à utiliser plusieurs grappes en parallèle pour toujours se situer dans la partie d'évolutivité linéaire.

Pour une grappe de 8 nœuds, une fois que le nombre de nœuds supplémentaires est égal à 8, le système est décomposé en deux grappes contenant les mêmes données. Les requêtes des clients doivent alors être réparties sur les deux systèmes. Grâce à notre technique de répartition des données, les deux systèmes sont quasiment identiques : le placement des données est un peu différent pour la grappe issue des nœuds additionnels, mais le même type de redondance des données permet d'assurer la même tolérance aux pannes. À ce stade, on peut à nouveau faire évoluer le système en rajoutant des nœuds à la grappe d'origine. La Figure 4.17 illustre la procédure utilisée pour augmenter la capacité du système : il est préférable de toujours ajouter les nœuds à la même grappe au lieu de les ajouter à toutes les grappes existantes à la fois. En effet, même si la répartition de la charge sur les grappes devient plus complexe car inégale, l'avantage est

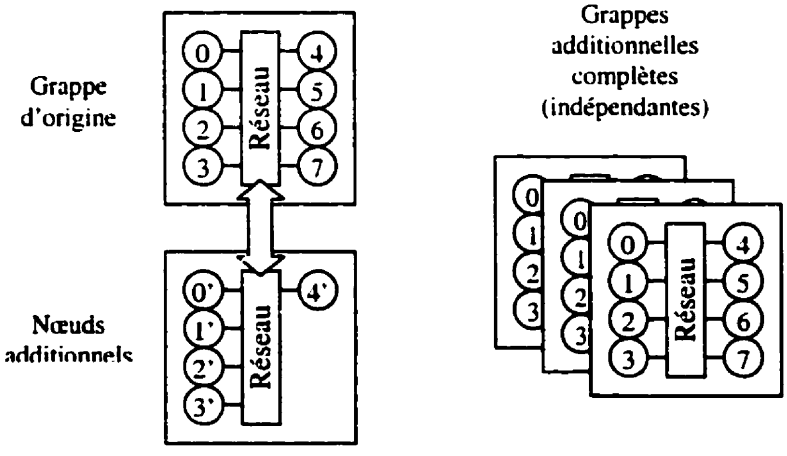

Figure 4.17 Méthode évolutive de développement

qu'à chaque fois que 8 nœuds ont été rajoutés, une nouvelle grappe indépendante devient opérationnelle.

La Figure 4.18 présente les résultats d'une simulation similaire, mais pour une grappe de 16 nœuds. On constate que la capacité limite du réseau est atteinte pour 4 nœuds additionnels. Cela correspond bien à une bande passante de 220 Mo/s. La Figure 4.19 montre qu'un tel système a une évolutivité en « marche d'escalier » : la granularité n'est pas de un nœud, puisque au-delà de 4 nœuds additionnels, les performances n'augmentent plus. Mais, grâce à notre méthode, on atteint des performances proches du cas idéal lorsqu'une grappe additionnelle est créée.

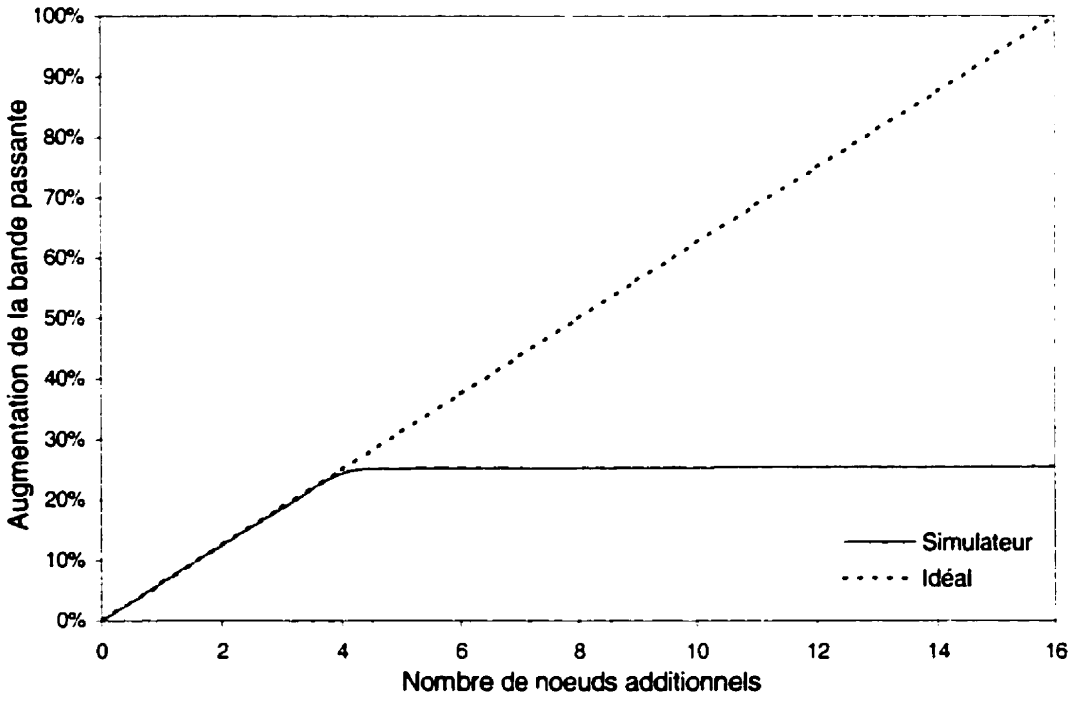

Figure 4.18 Évolutivité d'une grappe de 16 nœuds

À la Figure 4.19, nous n'avons pas pris en compte les défauts de répartition de la charge entre des grappes identiques, ni le coût même de cette tâche. Il est toujours préférable d'utiliser une grappe de base ayant la plus grande taille possible, étant donné les limitations du matériel utilisé. Dans notre cas, la bande passante maximale du réseau

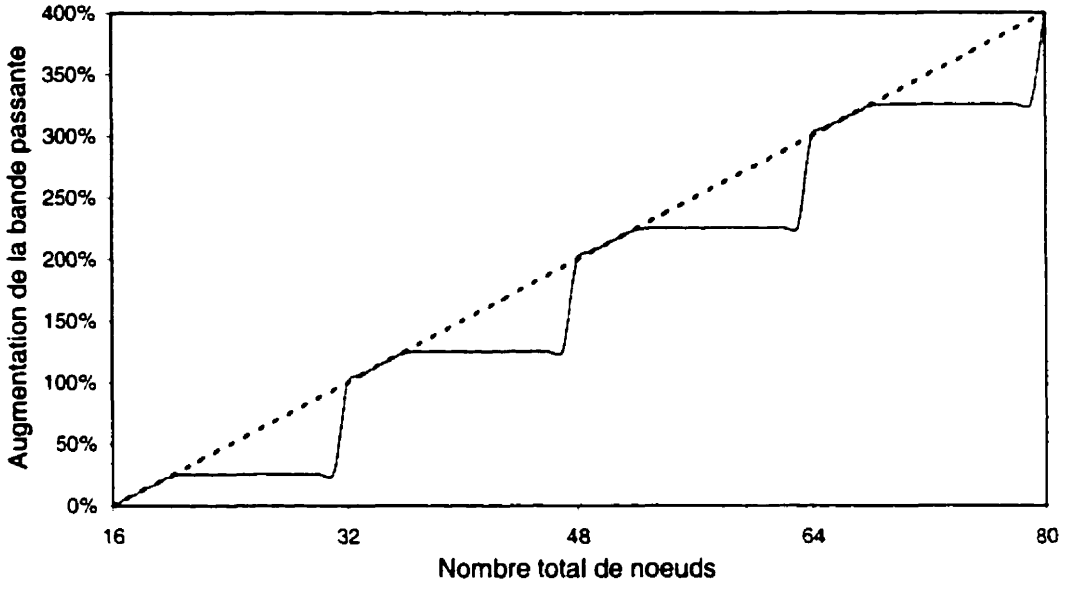

Figure 4.19 Évolutivité d'un système basé sur des grappes de 16 nœuds

est de 220 Mo/s. On peut en déduire que la taille optimale de la grappe d'origine est de 10 nœuds. Nous avons testé ce système avec 25 clients qui accèdent simultanément à des fichiers de 20 Mo. La Figure 4.20 montre que, lorsqu'on ajoute des nœuds à ce système, l'écart par rapport à l'évolutivité idéale reste inférieure à 1%. Pour 10 nœuds supplémentaires, la bande passante est de 219 Mo, ce qui est proche de la capacité maximale du réseau Ethernet utilisé. La Figure 4.21 montre que notre technique de répartition des données permet de très bien répartir la charge et d'optimiser l'utilisation des ressources : le défaut de répartition de la charge est de 0,22% et est dû au nombre fini de blocs lus.

Pour tester les performances pratiques de notre algorithme de distribution et de lecture des données, nous avons simulé l'évolutivité d'une grappe de 100 nœuds en éliminant la limite des 220 Mo/s imposée par le réseau Ethernet : avec 100 nœuds additionnels, l'écart par rapport au cas idéal est inférieur à 2%. Cela confirme que les performances de notre méthode de « declustering » sont indépendantes du nombre de nœuds utilisé. Les seules conditions pour que la charge soit bien répartie sont finalement :

- une probabilité de lecture égale des blocs virtuels de données ;
- une quantité de données suffisamment importante pour compenser les effets de bord de l'algorithme de lecture.

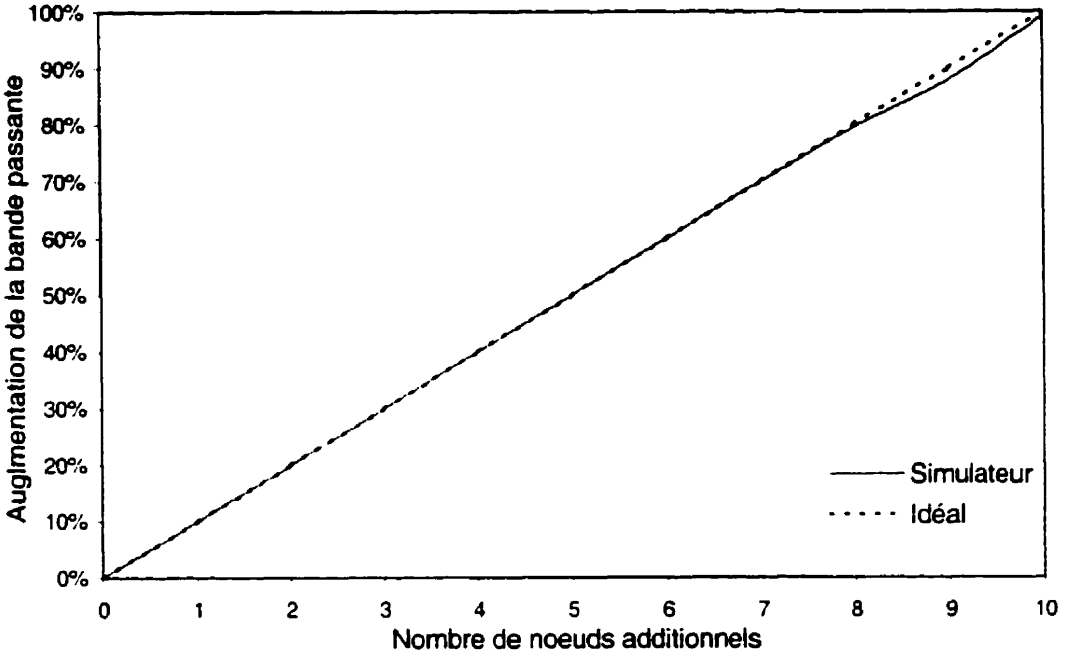

Figure 4.20 Evolutivité d'un cluster de 10 nœuds

L'un des avantages de notre système de fichier est de permettre l'évolutivité sans interruption de service et en limitant les perturbations occasionnées par une mise à jour du système. Lors de l'ajout d'un nœud additionnel, celui-ci doit lire les blocs de données qui lui sont attribués et les stocker localement : du fait de la redondance des données, un nœud additionnel doit mettre à jour le double de la quantité moyenne de donnée qui se trouve sur un nœud de la grappe d'origine. Mais, notre méthode de « declustering » permet de répartir cette charge sur tous les nœuds déjà opérationnels (y compris les nœuds additionnels). C'est pourquoi il est plus efficace d'ajouter les nœuds un par un.

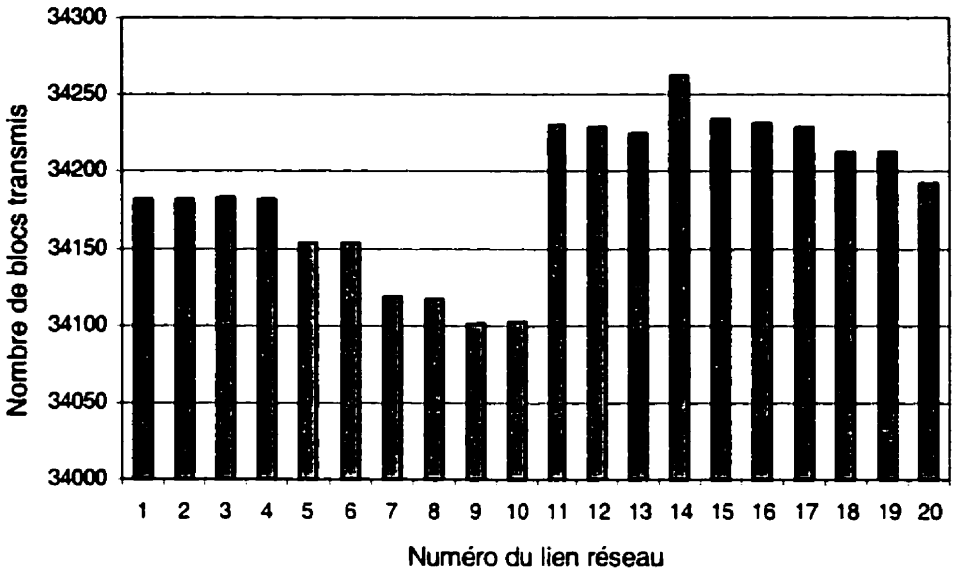

Figure 4.21 Répartition des blocs lus sur les nœuds de la grappe

Nous avons simulé l'ajout de nœuds à une grappe d'origine de 10 nœuds dans les conditions suivantes :

- 25 clients accèdent simultanément à des fichiers de 20 Mo.
- Le système de fichier comprend 1500 fichiers de 20 Mo (30 Go au total). Chaque nœud de la grappe d'origine stocke donc 3 Go de données. La taille d'un bloc de donnée est de 64 Ko.
- Les premier et deuxième nœuds additionnels sont ajoutés les uns après les autres au temps t=100 s et t=1500 s. Ainsi, le deuxième nœud est ajouté après que le premier eut fini sa mise à jour. Chaque nœud doit mettre à jour 93750 blocs, soit 6 Go de données.
- Pour lire les blocs à stocker localement, un nœud additionnel se comporte comme un client : sa priorité dans les files d'attente FIFO est identique aux clients ordinaires. Cela revient donc à rajouter la charge d'un client le temps de la mise à jour.

La Figure 4.22 représente l'évolution de la bande passante en fonction du temps : une courbe représente la bande passante totale que délivre la grappe et l'autre la bande passante qui est destinée au service des clients. Nous pouvons noter qu'il faut environ 50 min pour que deux nœuds deviennent opérationnels. À chaque fois qu'un nœud devient opérationnel, le gestionnaire du système de fichiers met à jour sa table des nœuds disponibles : cette opération est très simple car elle consiste à modifier un paramètre dans l'algorithme de lecture. La charge des nouvelles requêtes arrivant est alors supportée par l'ensemble des nœuds opérationnels. Pour la mise à jour des données d'un nœud additionnel, il faut faire un compromis entre le temps nécessaire pour que le nœud devienne opérationnel et la perturbation occasionnée sur le service des clients. Pour garantir une qualité de service minimale, on peut définir un seuil en dessous duquel la mise à jour est différée.

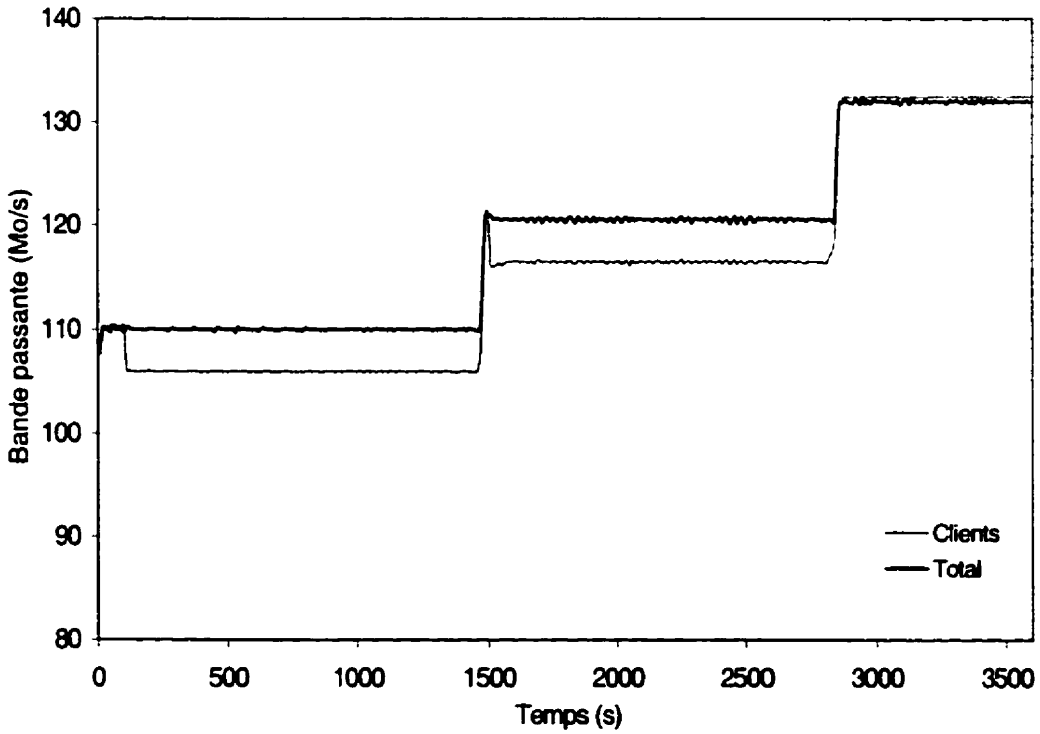

Figure 4.22 Scénario d'ajout de nœuds additionnels

CHAPITRE 5
CONCLUSION

Comparé au domaine de l'évolutivité des performances de calcul, l'évolutivité des systèmes de fichiers a fait l'objet de peu de recherches. Ce mémoire s'est donc concentré plus particulièrement sur l'utilisation d'une grappe d'ordinateurs pour des applications commerciales. La lecture de données est alors un problème central. Dans ce contexte, les indices de performance sont différents : l'objectif est de maximiser le nombre de clients pouvant être servis simultanément. Il faut aussi garantir un certain niveau de qualité de service et la tolérance aux pannes. Enfin, les contraintes de coûts étant primordiales pour une application commerciale, la granularité de l'évolutivité doit être faible afin de pouvoir adapter les performances du système aux besoins. Idéalement, il doit être possible d'augmenter les performances graduellement en ajoutant des nœuds un par un.

5.1 Synthèse des travaux et principales contributions

Ce mémoire propose un système de fichiers qui se base sur le « wide striping » combiné à des techniques de distribution des blocs de données. Nous avons choisi le *Parallel Virtual File System* (PVFS) comme base de départ : ce système de fichiers développé par l'Université de Clemson implémente le « wide striping » sur une grappe Linux. Nous avons développé un simulateur de PVFS puis nous l'avons calibré à l'aide de mesures prises sur un système réel afin d'en étudier le comportement lorsque le nombre de nœuds dans la grappe est grand. À travers l'étude théorique, l'étude des systèmes déjà existants et les simulations, nous avons montré que l'évolutivité d'un tel

système est limitée par les performances du réseau. Une fois que la bande passante maximale du réseau est atteinte, l'ajout d'un nœud peut même dégrader les performances globales. Puisque la taille d'une grappe est limitée par les performances du matériel, nous avons proposé d'utiliser plusieurs grappes en parallèle. Cela revient alors à constituer une grappe de grappes et permet de rester dans la zone où l'évolutivité de chaque grappe est quasiment idéale.

Pour améliorer l'évolutivité de PVFS, nous avons proposé une nouvelle technique de placement des blocs de données. Celle-ci permet l'ajout de nœuds un par un, avec une amélioration des performances très proche du cas idéal. À chaque fois que le nombre de nœuds additionnels est égal au nombre de nœuds de la grappe d'origine, une nouvelle grappe indépendante est formée. Cette architecture exploite à la fois les avantages de la réplication et du « wide stripping ». En effet, C. Chou et al. ont montré que le « wide striping » permet de répartir la charge de manière uniforme mais l'évolutivité est alors limitée par les performances du matériel [Chou99]. De même, comme nous l'avons vu dans le chapitre 2, la simple réplication ne permet qu'une évolutivité limitée. C'est la combinaison des deux techniques qui assure l'évolutivité de notre système. Dans le cas d'un « wide striping » simple, il faut redistribuer tous les blocs de données à chaque fois que le nombre de nœuds change : la reconfiguration est donc très coûteuse et nécessite en général l'interruption du service. Grâce à notre approche, la charge est également répartie sur l'ensemble de la grappe sans nécessiter de redistribution des données sur les nœuds déjà existants.

Pour assurer la tolérance aux pannes, nous utilisons une méthode de répartition des blocs de redondance appelée « multi-chained declustering ». Nous avons choisi cette technique car elle se rapproche de celle que nous avons utilisée pour assurer l'évolutivité. Elle possède également l'avantage de maintenir une meilleure qualité de service quand un nœud tombe en panne en répartissant la charge sur l'ensemble des nœuds survivants.

Par ailleurs, nous avons décrit les algorithmes utilisés dans notre système de fichiers. Nous avons implémenté les améliorations proposées dans notre simulateur pour

comparer notre système de fichiers à PVFS et valider les résultats théoriques. Nous avons ainsi constaté que l'implémentation est relativement simple car les algorithmes utilisés sont statiques et ne dépendent pas de la charge du système. Les tests montrent que notre technique de placement des données garantit une évolutivité proche du cas idéal : l'ajout d'un nœud fait augmenter linéairement la bande passante délivrée par le système de fichiers. Nous avons également simulé des scénarios d'ajout de nœuds sans interruption de service pour mesurer les perturbations dues à la configuration des nouveaux nœuds de la grappe. Dans le cas de files d'attente sans priorité, les mises à jour nécessaires lorsqu'un nœud est rajouté consomment les ressources équivalentes à un client de la grappe. Le nombre de blocs accédé est égal au double du nombre de blocs moyens sur chaque nœud de la grappe, du fait des blocs de redondance.

5.2 Limitations des travaux

Le but de notre système de fichiers est de fournir un espace de stockage accessible à un nombre important de clients. Une application typique est un serveur multimédia qui propose des images, des fichiers son ou des fichiers vidéo : les requêtes correspondent à la lecture de données. Un tel système doit être évolutif pour s'adapter à un nombre croissant de clients. En dehors du cas de la mise à jour d'un nouveau nœud, nous n'avons pas étudié l'écriture de données en tant que tel. Cependant, cette situation est similaire à une écriture pour une architecture de type « wide stripping » avec des miroirs pour la redondance des données : par rapport à la lecture, l'écriture est plus coûteuse en ressource à cause de la redondance des données.

La tolérance aux pannes de notre système de fichiers n'a pas fait l'objet de simulation car cela déborde du cadre de ce mémoire. De plus, des méthodes similaires de distribution des blocs de redondance ont déjà fait l'objet d'articles. Cependant, la distribution des blocs de données que nous proposons sert également à améliorer l'évolutivité du système de fichiers réparti. Il serait donc intéressant d'étudier la tolérance aux pannes de notre système, en particulier lors de la panne d'un des nœuds

additionnels, si le nombre de nœuds additionnels est strictement inférieur au nombre de nœuds de la grappe d'origine.

Dans ce mémoire, nous avons également fait l'hypothèse que le matériel utilisé est homogène. Puisque c'est la bande passante du réseau qui constitue la principale limitation, cette hypothèse revient à utiliser des commutateurs et des liens de communications tous identiques. Ainsi, nous n'avons pas pris en compte d'éventuels coûts de répartition dynamique de la charge. Nous avons également considéré que la répartition des requêtes des clients entre des grappes identiques était parfaite. Enfin, tous les clients possèdent le même niveau de priorité et ils sont servis selon la politique du « meilleur effort ».

5.3 Recommandations pour des travaux futurs

À notre connaissance, l'utilisation d'une distribution de blocs de données dans le but de garantir l'évolutivité d'une grappe d'ordinateurs n'avait pas encore été proposée. Bien que nous ayons testé notre système de fichiers à travers des simulations, il reste à développer un prototype : par exemple en modifiant le système de fichier PVFS. L'évolutivité de ce prototype pourra alors être testée sur une grappe comportant un grand nombre de nœuds. Il serait également intéressant d'utiliser un réseau offrant une plus grande bande passante pour pouvoir augmenter le nombre de nœuds de la grappe d'origine. Les nouvelles technologies « Fast Ethernet » sont par exemple un bon compromis entre le coût et les performances [Mache99].

Un prototype permettrait également de tester les optimisations possibles. Par exemple, dans les architectures réparties, les mémoires tampons et les mémoires caches peuvent améliorer les performances. Comme nous l'avons déjà mentionné, il peut également être intéressant d'utiliser des algorithmes dynamiques de lecture des données afin d'améliorer la qualité de service offerte aux clients, en particulier lorsque les contraintes temps réel sont fortes. Notre solution se base sur la distribution des données et de la redondance sur l'ensemble de la grappe d'ordinateurs : des techniques comme le « graduated declustering » [Dusseau99] pourraient donc être adaptées pour garantir une

meilleure qualité de service. Enfin, il serait intéressant d'approfondir la technique de placement des blocs de redondance pour garantir une qualité de service optimale en cas de panne sur une grappe comportant des nœuds additionnels.

Pour rendre le système adapté à des écritures de petite taille, on peut envisager l'utilisation d'un système hybride. Ce système utiliserait deux systèmes de fichiers d'architecture différente : le système de fichiers que nous avons développé pour la lecture de fichiers volumineux et un système de fichiers plus adapté à des petites écritures.

BIBLIOGRAPHIE

[Amdahl67] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *In AFIPS Conference Proceedings*, Vol. 30, 1967, pp.483-485.

[Barak87] A. Barak et Y. Kornatzky, "Design Principle of Operating Systems for Large Scale Multicomputers", *Proceedings of the International Workshop on Experience with Distributed Systems*, Kaiserslautem, LNCS 309, Springer-Verlag, Septembre 1987, pp. 104-123.

[Barak99] A. Barak, O. La'adan et A. Shiloh, "Scalable Cluster Computing with MOSIX for LINUX", *In Proceedings of the 5-th Annual Linux Expo*, Raleigh, N.C., Mai 1999, pp. 95-100.

[Barak99a] A. Barak, I. Gilderman et I. Metrik., "Performance of the Communication Layers of TCP/IP with the Myrinet Gigabit LAN", *Computer Communications*, Vol. 22, No. 11, Juillet 1999.

[Brightwell99] R. Brightwell et Steve Plimpton, "Scalability and Performance of a Large Linux Cluster", Sandia National Laboratories, *the Special Issue of the Journal of Parallel and Distributed Computing on Cluster and Network-Based Computing*, Décembre 1999.

[Buyya99] R. Buyya (ed.), *High Performance Cluster Computing: Architectures and Systems*. Vol.1, Prentice Hall PTR, NJ, USA, 1999, 845 pages

[Carns2000] P. H. Carns, W. B. Ligon III, R. B. Ross, et R. Thakur, "PVFS: A Parallel File System For Linux Clusters", *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, Octobre 2000, pp. 317-327.

[Chandran87] S. Chandran and L. S. Davis, "An approach to parallel vision algorithms", In R. Porth (ed.), *Parallel Processing, SIAM*, Philadelphia, PA, 1987.

[Chou99] C. Chou, L. Golubchik et J.C.S. Lui, "Striping Doesn't Scale: How to Achieve Scalability", Technical Report, CS-TR-1999-03, University of Maryland, Septembre 1999, 34 pages.

[CSIM] Mesquite CSIM 18 - *A Development Toolkit for Simulation and Modeling*, http://www.mesquite.com/csim18page.htm.

[Davies1994] N. J. Davies, *The performance and scalability of parallel systems*, Mémoire de thèse, Université de Bristol, Angleterre, Décembre1994.

[Dusseau99] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, K. Yelick. "Cluster I/O with River: Making the Fast Case Common", *In Sixth Workshop on I/O in Parallel and Distributed Systems*, Mai 1999. http://now.cs.berkeley.edu/River/

[Eager89] D. L. Eager, J. Zahorjan, et E. D. Lazowska, "Speedup versus efficiency in parallel systems", *IEEE Transactions on Computers*, Vol. 38, No. 3, 1989, pp. 408-423.

[Flatt89] H. P. Flatt et K. Kennedy, "Performance of parallel processors", *Parallel Computing*, Vol. 12, 1989, pp.1-20.

[Flatt90] H. P. Flatt, *Further applications of the overhead model for parallel systems*, Technical Report G320-3540, IBM Corporation, Palo Alto Scientific Center, Palo Alto, CA, 1990.

[Ganger98] G. Ganger, B. Worthington et Y. Patt, *The DiskSim Simulation Environment Version 1.0 Reference Manual*, Technical Report CSE-TR-358-98, Dept of Electrical Engineering and Computer Science, The University of Michigan, Février 1998, 53 pages.

[Grama93] A. Grama, A. Gupta, et V. Kumar, "Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures", *IEEE Parallel and Distributed Technology*, Special Issue on Parallel and Distributed Systems: From Theory to Practice, Vol. 1, No. 3, 1993, pp. 12-21.

[Gropp99] W. Gropp, E. Lusk, et R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge, MA, 1999, 382 pages.

[Gupta91] A. Gupta et V. Kumar, "The scalability of matrix multiplication algorithms on parallel computers", Technical Report TR 91-54, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1991. A short version appears in *Proceedings of 1993 International Conference on Parallel Processing*, pp. III115-III119, 1993.

[Gupta93] A. Gupta et V. Kumar, "Performance properties of large scale parallel systems", *Journal of Parallel and Distributed Computing*, Vol. 19, 1993, pp. 234-244.

[Gustafson88] J. L. Gustafson, "Reevaluating Amdahl's law", *Communications of the ACM*, Vol. 31, No. 5, 1988, pp. 532-533.

[Gustafson88a] J. L. Gustafson, G. R. Montry, et R. E. Benner, "Development of parallel methods for a 1024processor hypercube", *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 4, 1988, pp. 609-638.

[Kofman98] D. Kofman, H Korezlioglu et S. Tomé, *Élément de théorie des files d'attente*, Support de cours de l'École Nationale Supérieure des Télécommunications, version 1.3, Septembre 1998, 178 pages

[Kruskal88] C. P. Kruskal, L. Rudolph et M. Snir, *A complexity theory of efficient parallel algorithms*, Technical Report RC13572, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[Kumar94] V. Kumar et A. Gupta, "Analyzing scalability of parallel algorithms and architectures", *Journal of Parallel and Distributed Computing*, Vol.22, No.3, 1994, pp.379-391.

[Luecke2000] Glenn R. Luecke, Bruno Raffin et J. J. Coyle, "Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600", *Journal of Performance Evaluation and Modeling for Computer Systems (PEMCS)*, Iowa Sate University, Iowa, USA, 2000. http://hpc-journals.ecs.soton.ac.uk/PEMCS/Papers/Paper13/

[Mache99] J. Mache, "An Assessment of Gigabit Ethernet as Cluster Interconnect", *1st IEEE Computer Society International Workshop on Cluster Computing*, Melbourne, Australia, Décembre 1999.

[Marinescu93] D. C. Marinescu et J. R. Rice, *On high level characterization of parallelism*, Technical Report CSD-TR-1011, CAPO Report CER-90-32, Computer Science Department, Purdue University, West Lafayette, IN, Revised June 1991.

[Massiglia2000] P. Massiglia, *RAID for Enterprise Computing*, A Technology White Paper from VERITAS Software Corporation, Janvier 2000, 45 pages.

[MPI97] "MPI-2: Extension to the message-passing interface", *Message Passing Interface Forum*, Juillet 1997. http://www.mpi-forum.org/docs/docs.html

[Mukherjee99] R. Mukherjee, "A scalable and Highly Available Clustered Web Server", *High Performance Cluster Computing: Architectures and Systems*, Vol.1, Prentice Hall PTR, NJ, USA, 1999, pp. 811-839

[Sun91] X. Sun et D. T. Rover, *Scalability of parallel algorithmmachine combinat ions*, Technical Report IS-5057, Ames Laboratory, Iowa State University, Ames, IA, 1991.

[Sun95] X. Sun, *The Relation of Scalability and Execution Time*, Department of Computer Science, Louisiana State University, 1995.

[Sun98] K. Li et X. Sun, *AverageCase Analysis of Isospeed Scalability of Parallel Computations on Multiprocessors*, State University of New York at New Paltz, Department of Mathematics and Computer Science, Technical Report #98-108, 1998.

[Tang90] Z. Tang and G. Li, "Optimal granularity of grid iteration problems", *In Proceedings of the 1990 International Conference on Parallel Processing*, 1990, pp. I111-I118.

[VanCatledge89] F. A. VanCatledge, "Towards a general model for evaluating the relative performance of computer systems", *International Journal of Supercomputer Applications*, Vol. 3, No. 2, 1989, pp. 100-108.

[Zhou89] X. Zhou, "Bridging the gap between Amdahl's law and Sandia laboratory's result", *Communications of the ACM*, Vol. 32, No. 8, 1989, pp.1014-1015.

[Zorbas89] J. R. Zorbas, D. J. Reble, et R. E. VanKooten, "Measuring the scalability of parallel computer systems", *In Supercomputing '89 Proceedings*, 1989, pp. 832-841.

ftp://ftp.cs.sandia.gov/pub/papers/bright/cplant-tflops.pdf

http://computer.org/proceedings/iwcc/0343/03430036abs.htm

ANNEXE A
DISTRIBUTION DES DONNÉES

Nous reprenons les notations introduites au chapitre 4 :

- X désigne un bloc de données se trouvant sur un des nœuds d'origine et correspondant aux données primaires ;
- Y désigne un bloc de données se trouvant sur un des nœuds d'origine de la grappe et correspondant aux données redondantes ;
- U désigne un bloc de données se trouvant sur un nœud additionnel de la grappe et correspondant aux données primaires (i.e. un bloc n'étant pas un duplicata) :
- V désigne un bloc de données se trouvant sur un nœud additionnel de la grappe et correspondant aux données redondantes.

1) Les blocs de type X (nœuds d'origine, données primaires) :

Ils sont distribués selon un « stripe » séquentiel simple :

$$X_k = \left(\frac{k - k[N]}{N}, k[N] \right) \qquad \text{(A.1)}$$

$$x(i, j) = x(X_k) \Rightarrow k = i * N + j$$

2) Les blocs de type Y (nœuds d'origine, données redondantes) :

Distribution de type « multi-chained declustering » (redondance des blocs X).

Cette distribution se définie par :

$$y(i, j) = y(Y_k) \Leftrightarrow Y_k = (i, j)$$

$$y(i, j) = x\left(i, \{j - (i[N-1]+1)\}[N]\right)$$

$$x(p, q) = y\left(p, \{q + p[N-1]+1\}[N]\right) \qquad \text{(A.2)}$$

Nous fixons k, ce qui entraîne :

$$y(i, j) = x(p, q)$$

En substituant dans (A.2) à l'aide de (A.1), on obtient :

$$\begin{cases} i = \dfrac{k - k[N]}{N} \\ j = \left\{ k[N] + \left(\dfrac{k - k[N]}{k} \right)[N-1] + 1 \right\}[N] \end{cases} \qquad \text{(A.3)}$$

Ce qui démontre que :

$$Y_k = \left(\frac{k - k[N]}{N}, \left\{ k[N] + \left(\frac{k - k[N]}{k} \right)[N-1] + 1 \right\}[N] \right)$$

Déterminons maintenant k en fonction de i et j :

$$\text{(A.3)} \Leftrightarrow \begin{cases} i = \dfrac{k - k[N]}{N} \\ j = \{k[N] + i[N-1] + 1\}[N] \end{cases}$$

$$\text{(A.3)} \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = Z[N] \quad où\ Z \equiv k[N] + i[N-1] + 1 \end{cases}$$

Or $0 \le k[N] + i[N-1] + 1 \le 2.(N-1)$ par définition du modulo. On peut alors envisager 2 cas possibles :

1^e cas : $0 \le Z \le (N-1) \Rightarrow j = Z$

$$\Rightarrow k[N] = j - \{i[N-1] + 1\}$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j - \{i[N-1] + 1\} \le (N-1)$$

$$\Rightarrow j \ge i[N-1] + 1$$

2^e cas : $N \le Z \le 2.(N-1) \Rightarrow j = Z - N$

$$\Rightarrow k[N] = j - \{i[N-1] + 1 - N\}$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j - \{i[N-1] + 1 - N\} \le (N-1)$$

$$\Rightarrow j \le i[N-1]$$

En conclusion, nous avons démontré que :

$$y(i,j) = y(Y_k) \Leftrightarrow \begin{cases} k = i * N + j - \big(i[N-1]+1\big) & si \quad j \geq i[N-1]+1 \\ k = i * N + j - \big(i[N-1]+1\big) + N & si \quad j \leq i[N-1] \end{cases}$$

3) Les blocs de type U (nœuds additionnels, données primaires) :

Distribution de type « extended multi-chained declustering » par rapport aux blocs X :

$$u(i,j) = u(U_k) \Leftrightarrow U_k = (i,j)$$

$$u(i,j) = x\big(i, \{j + i[N]\}[N]\big)$$

$$x(p,q) = u\big(p, \{q - p[N]\}[N]\big) \qquad \text{(A.4)}$$

Nous fixons k, ce qui entraîne :

$$u(i,j) = x(p,q)$$

Comme précédemment, en substituant dans (A.4) à l'aide de (A.1), on obtient :

$$\begin{cases} i = \dfrac{k - k[N]}{N} \\ j = \left\{k[N] - \left(\dfrac{k - k[N]}{k}\right)[N]\right\}[N] \end{cases} \qquad \text{(A.5)}$$

Ce qui démontre que :

$$U_k = \left(\frac{k - k[N]}{N}, \left\{k[N] - \left(\frac{k - k[N]}{k}\right)[N]\right\}[N]\right)$$

Déterminons maintenant k en fonction de i et j :

$$\text{(A.5)} \Leftrightarrow \begin{cases} i = \dfrac{k - k[N]}{N} \\ j = \{k[N] - i[N]\}[N] \end{cases}$$

$$\text{(A.5)} \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = W[N] \quad où\ W \equiv k[N] - i[N] \end{cases}$$

Or $-(N-1) \leq k[N] - i[N-1] + 1 \leq (N-1)$ par définition du modulo. On peut alors envisager 2 cas possibles :

1^e cas : $0 \leq Z \leq (N-1) \Rightarrow j = Z$

$\Rightarrow k[N] = j + i[N]$

Or $0 \leq k[N] \leq (N-1)$ impose :

$0 \leq j + i[N] \leq (N-1)$

$\Rightarrow j \leq N - i[N] - 1$

2^e cas : $-(N-1) \leq Z \leq 0 \Rightarrow j = Z + N$

$\Rightarrow k[N] = j + i[N] - N$

Or $0 \leq k[N] \leq (N-1)$ impose :

$0 \leq j + i[N] - N \leq (N-1)$

$\Rightarrow j \geq N - i[N]$

En conclusion, nous avons démontré que :

$$u(i, j) = u(U_k) \Leftrightarrow \begin{cases} k = i * N + j + i[N] & si \quad j \leq N - i[N] - 1 \\ k = i * N + j + i[N] - N & si \quad j \geq N - i[N] \end{cases}$$

4) Les blocs de type V (nœuds additionnels, données redondantes) :

Distribution de type « multi-chained declustering » par rapport aux blocs U. Cette distribution est équivalente à la distribution des blocs Y par rapport aux blocs X :

$v(i, j) = v(Y_k) \Leftrightarrow V_k = (i, j)$

$v(i, j) = u\left(i, \{j - (i[N-1] + 1)\}[N]\right)$

$u(s, t) = v\left(s, \{t + s[N-1] + 1\}[N]\right)$

or

$$u(i, j) = x\big(i, \{j + i[N]\}[N]\big)$$

$$x(p, q) = u\big(p, \{q - p[N]\}[N]\big)$$

d'où

$$v(i, j) = x\Big(i, \big\langle \{j - (i[N-1] + 1)\}[N] + i[N] \big\rangle [N]\Big)$$

$$x(p, q) = v\Big(p, \big\langle \{q - p[N])\}[N] + p[N-1] + 1 \big\rangle [N]\Big) \qquad \text{(A.6)}$$

Nous fixons k, ce qui entraîne :

$$v(i, j) = x(p, q)$$

Comme précédemment, en substituant dans (A.6) à l'aide de (A.1), on démontre que :

$$V_k = \left(\frac{k - k[N]}{N}, \left\langle \left\{ k[N] - \left(\frac{k - k[N]}{k} \right)[N] \right\} [N] + \left(\frac{k - k[N]}{k} \right)[N-1] + 1 \right\rangle [N] \right)$$

Déterminons maintenant k en fonction de i et j :

$$\text{(A.6)} \Leftrightarrow \begin{cases} i = \dfrac{k - k[N]}{N} \\ j = \big\langle \{k[N] - i[N]\}[N] + i[N-1] + 1 \big\rangle [N] \end{cases}$$

$$\text{(A.6)} \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = K[N] \quad \text{où } K \equiv \{k[N] - i[N]\}[N] + i[N-1] + 1 \end{cases}$$

Posons aussi : $K \equiv L[N] + i[N-1] + 1 \quad avec\ L \equiv k[N] - i[N]$

Or $1 \le L[N] + i[N-1] + 1 \le 2.(N-1)$ par définition du modulo. On peut alors envisager 2 cas possibles :

1) 1^{e} cas : $0 \le K \le (N-1) \Rightarrow j = K$

2) 2^{e} cas : $N \le K \le 2.(N-1) \Rightarrow j = K - N$

De plus, Or $-(N-1) \le k[N] - i[N] \le (N-1)$ par définition du modulo. On peut alors envisager 2 cas possibles :

a) 1^{e} cas : $0 \le L \le (N-1) \Rightarrow K = L + i[N-1] + 1$

b) 2^{e} cas : $-(N-1) \le L \le 0 \Rightarrow K = L + N + i[N-1] + 1$

Envisageons maintenant la combinaison des différents cas possibles :

1a) $0 \le K \le (N-1)$ **et** $0 \le L \le (N-1)$

$$(A.6) \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = k[N] - i[N] + i[N-1] + 1 \end{cases}$$

$$\Rightarrow k[N] = j + i[N] - (i[N-1] + 1)$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j + i[N] - (i[N-1] + 1) \le (N-1)$$

$$\Rightarrow i[N-1] + 1 - i[N] \le j \le i[N-1] + N - i[N]$$

1b) $0 \le K \le (N-1)$ **et** $-(N-1) \le L \le 0$

$$(A.6) \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = k[N] - i[N] + N + i[N-1] + 1 \end{cases}$$

$$\Rightarrow k[N] = j + i[N] - (i[N-1] + N + 1)$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j + i[N] - (i[N-1] + N + 1) \le (N-1)$$

$$\Rightarrow i[N-1] + N + 1 - i[N] \le j \le i[N-1] + 2.N - i[N]$$

2a) $N \le K \le 2.(N-1)$ **et** $0 \le L \le (N-1)$

$$(A.6) \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = k[N] - i[N] + i[N-1] + 1 - N \end{cases}$$

$$\Rightarrow k[N] = j + i[N] + N - (i[N-1] + 1)$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j + i[N] + N - (i[N-1]+1) \le (N-1)$$

$$\Rightarrow i[N-1]+1-i[N]-N \le j \le i[N-1]-i[N]$$

2b) $N \le K \le 2.(N-1)$ **et** $-(N-1) \le L \le 0$

$$(A.6) \Leftrightarrow \begin{cases} k = i.N + k[N] \\ j = k[N] - i[N] + N + i[N-1] + 1 - N \end{cases}$$

$$\Rightarrow k[N] = j + i[N] - (i[N-1]+1)$$

Or $0 \le k[N] \le (N-1)$ impose :

$$0 \le j + i[N] - (i[N-1]+1) \le (N-1)$$

$$\Rightarrow i[N-1]+1-i[N] \le j \le i[N-1]+N-i[N]$$

En conclusion, nous avons démontré que :

$$v(i,j) = v(V_k) \Leftrightarrow \begin{cases} k = i*N + j + i[N] - (i[N-1]+1) \\ \qquad si \quad i[N-1]+1-i[N] \le j \le N + i[N-1] - i[N] \\ k = i*N + j + i[N] - (i[N-1]+1+N) \\ \qquad si \quad i[N-1]+1+N-i[N] \le j \le 2N + i[N-1] - i[N] \\ k = i*N + j + i[N] + (N-1) - (i[N-1]+1) \\ \qquad si \quad i[N-1]-(N-1)-i[N] \le j \le i[N-1] - i[N] \\ k = i*N + j + i[N] - (i[N-1]+1) \\ \qquad si \quad i[N-1]+1-i[N] \le j \le N + i[N-1] - i[N] \end{cases}$$