Étude quantitative des mécanismes d'équilibrage de charge dans les systèmes de programmation pour le calcul parallèle

Martha Rosa Castaneda Retiz

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : "Informatique: systèmes et communications"

présentée et soutenue publiquement

par

Martha Rosa CASTAÑEDA RETIZ

le 12 novembre 1999

Étude quantitative des mécanismes d'équilibrage de charge dans les systèmes de programmation pour le calcul parallèle

---

Directeur de thèse

Mme. Brigitte PLATEAU

Composition du jury :

Président :	M. Andrzej	DUDA
Rapporteurs :	M. Jean-Michel	FOURNEAU
	Mme. Noufissa	MIKOU
Examinateurs :	Mme. Brigitte	PLATEAU, Directeur de thèse
	M. Jean-Louis	ROCH
	M. Jean-Marc	VINCENT

Thèse préparée au sein du
Laboratoire de Modélisation et Calcul
et soutenue au **Laboratoire Informatique et Distribution**
dans le cadre de l'Ecole Doctorale
"Mathématiques, Sciences et Technologie de l'Information"

à **Caterina**,
petit bout de vie
qui m’a accompagnée
tout au long
de l’écriture de ce document.

Remerciements

Je tiens à remercier Andrzej Duda qui m'a fait l'honneur de présider ce jury de thèse.

Un grand merci à Brigitte Plateau qui m'a aidée à dégager les grandes orientations de cette thèse. Les nombreuses discussions que nous avons eu ensemble ont été pour moi d'une très grande valeur.

Un grand merci également à Noufissa Mikou et Jean-Michel Fourneau d'avoir accepté de rapporter sur cette thèse malgré leurs contraintes d'emploi du temps. Leurs remarques ont été très utiles.

Je remercie Jean-Louis Roch et Jean-Marc Vincent, non seulement d'avoir accepté de participer à mon jury de thèse mais aussi pour les nombreuses discussions très enrichissantes tout au long de mon travail.

Merci à tous ceux qui ont lu le manuscrit de ma thèse, me faisant part de leurs commentaires et suggestions. Je pense spécialement à Philippe Augerat, qui a fait un travail énorme, pour corriger la première version du manuscrit, à François Galilee pour la lecture attentive du chapitre sur Athapascan, aux autres re-lecteurs de choc : Gregory Mounie, Olivier Briat, Christophe Rapine, Jean-Guillaume Dumas.

A tous les " APACHES " et " annexes " pour avoir sû créer et animer un communauté exemplaire, un groupe dans lequel j'ai trouve un soutien et une motivation pour continuer : Alain, Alexandre, Alfredo, Andréa, Benjur, Denis T., Denis N., Ekbel, Eric M., Fred, Gerson, Ilan, Jacques B., Jacques C., João Paulo, Joelle, Luiz-Gustavo, Mathias, Marcelo, Michel C., Natalie, Nicolas, Pierre-Eric, Roberta, Thierry.

Merci à Nhan-Duc, Lydie, Gerardo, Mary et à tous les amis.

Je tiens tout particulièrement à remercier mes parents qui m'ont toujours encouragé à continuer mes études. Je leur dois beaucoup. Ils m'ont soutenu avec leurs lettres depuis le Mexique.

Je voudrais remercier ma nouvelle famille " italienne ", Fernando et Marcela, pour leur soutien et la " bonne cuisine ".

Finalement j'exprime toute ma gratitude à Gabriele pour sa présence, ses encouragements, ses conseils, sa patience et si j'ai réussi à finir c'est grâce à toi.

Martha Rosa Castañeda Retiz

Table des matières

Table des figures

Liste des tableaux

Présentation

Dans tout système complexe, il est difficile d'établir avec précision les influences des différents éléments qui le composent. L'évaluation de performances de ces systèmes s'avère une tâche complexe et sans une méthodologie rigoureuse il est difficile d'arriver à obtenir des conclusions valides.

Dans le domaine de la programmation parallèle, comme dans tous les domaines, les systèmes sont de plus en plus complexes. La machine parallèle a des éléments matériels et logiciels qui ont un impact sur les performances. Comment approcher un véritable système et rationaliser les moyens pour mieux utiliser les ressources ?

La constatation générale est que dans le cadre du parallélisme, on a besoin de machines parallèles disposant d'environnements exécutifs qui exploitent véritablement leurs performances.

L'objectif de la régulation de la charge est la minimisation du temps d'exécution de l'application parallèle en cherchant une utilisation maximale des ressources du système. Il existe des applications parallèles qui ont une structure irrégulière, dynamique et imprévisible, et l'unique façon de les ordonnancer est de le faire dynamiquement au cours de l'exécution. Ce sont ces applications et les heuristiques qui réalisent ce travail d'ordonnancement que nous allons étudier.

Lors des études traditionnelles sur l'évaluation de performances des algorithmes d'équilibrage de charge, les algorithmes sont étudiés en faisant varier un seul paramètre à la fois, généralement parce que l'étude de plusieurs paramètres s'avère très complexe et d'interprétation difficile. Cependant, dans le cadre des stratégies dynamiques, il est clair que l'influence simultanée de plusieurs facteurs est importante. Nous considérons une nouvelle approche pour avoir une vision globale qui doit permettre d'établir des conclusions plus globales.

L'étude de l'influence simultanée de plusieurs facteurs dans le cadre de stratégies dynamiques pour les applications parallèles irrégulières est le but principal de notre recherche.

Le cadre du projet APACHE

Un environnement de programmation pour les applications parallèles est développé au sein du projet APACHE[Pa94] : ce système est nommé ATHAPASCAN. Le système ATHAPASCAN offre une plate-forme portable ATHAPASCAN0, avec des communications performantes et proposant les fonctionnalités de la multiprogrammation légère. D'autre part,

il offre une interface applicative ATHAPASCAN1, qui facilite l'écriture du programme et qui offre la gestion dynamique des ressources de façon transparente pour le programmeur.

Dans ce contexte, nous avons participé à la construction de l'ordonnanceur du système. L'ordonnanceur a une structure modulaire (il est écrit en C++). Il peut-être décrit comme une boîte à outil qui permet l'écriture de nouveaux algorithmes d'équilibrage de charge par dérivation des classes de base de l'ordonnanceur. De cette façon, le système ATHAPASCAN peut offrir une bibliothèque d'algorithmes d'équilibrage de charge mais aussi permettre à l'utilisateur la construction (si nécessaire) d'une nouvelle stratégie ou l'adaptation d'un algorithme déjà existant.

C'est sur cet environnement que nous avons développé notre plate-forme expérimentale.

Les objectif de la thèse

L'approche pour l'évaluation des performances des stratégies d'équilibrage de charge que nous avons choisie est basée sur la technique des mesures réelles sur une machine parallèle. L'évaluation de performances par prise de mesures est un problème délicat et demande la mise au point de plans d'expériences rigoureux. Sans méthodologie expérimentale, la validité des résultats est sujette à caution.

Dans cette thèse nous ne cherchons pas à proposer un nouvel algorithme. Dans la littérature nous en trouvons un éventail important. Nous voulons " rationaliser " les performances et proposer une méthodologie pour l'évaluation des algorithmes d'équilibrage de charge et les comparer.

Nous avons fait appel à une technique générique de modélisation quantitative d'algorithmes parallèles pour la création de notre charge applicative. Il s'agit de programmes parallèles qui utilisent les ressources de la machine mais qui ne résolvent pas un vrai problème de calcul. Nos modèles de programmes parallèles synthétiques sont basés sur les travaux de Kitajima [Kit94].

Organisation de ce document

Ce document suit le plan suivant :

I État de l'art sur l'équilibrage de charge pour les applications parallèles.

II La régulation de charge au sein d'ATHAPASCAN

III Évaluation des performances

La première partie est composée de trois chapitres. Le premier chapitre est une introduction générale au domaine du parallélisme. Le problème général de l'équilibrage de charge est présenté. Le deuxième chapitre décrit les grandes lignes de l'ordonnancement d'une application parallèle : un survol sur l'ordonnancement statique et sur l'ordonnancement dynamique est présenté. Le troisième chapitre est l'étude des fonctionnalités d'un mécanisme de répartition dynamique de charge : l'ordonnanceur. Nous avons déterminé les activités de base utilisées pour les stratégies de répartition dynamique de charge (des

aspects les plus généraux aux aspects les plus spécifiques pour arriver à leur implémentation).

La deuxième partie décrit le système ATHAPASCAN. Il est formé de deux chapitres. Le chapitre quatre est la description des environnements pour le calcul parallèle, avec la description du système ATHAPASCAN. Plus précisément nous présentons le modèle de programmation proposé par l'interface applicative du système : la couche ATHAPASCAN1. Dans le cinquième chapitre nous présentons l'implémentation faite de l'ordonnanceur modulaire dans le système ATHAPASCAN. Cet ordonnanceur répond aux fonctionnalités décrites dans le chapitre trois.

La troisième partie correspond au cœur de cette thèse, l'évaluation de performances des algorithmes pour l'équilibrage de charge. Il est composé de deux chapitres, le sixième décrit la planification expérimentale et le septième l'analyse des performances. Ces deux chapitres décrivent notre méthodologie pour l'évaluation quantitative des algorithmes d'équilibrage de charge.

Les expériences qui ont été faites, ont été réalisées sur une machine SP1 d'IBM.

Par convention, les mots ou expressions écrits en **gras** représentent des passages clés pour le sujet étudié dans le chapitre ou dans le paragraphe. Pour certaines notions le terme anglo-saxon est d'un usage plus pratique ou plus explicite que son équivalent français. Il lui est alors substitué et est typographié en *italique*.

Première partie

État de l’art sur la régulation de charge pour les applications parallèles

Chapitre 1

Le parallélisme et le problème général de la régulation de charge

Dans ce chapitre, nous présentons les domaines du parallélisme de façon très générale : les ordinateurs parallèles et la programmation parallèle, puis la problématique qui nous intéresse sur le contrôle des applications parallèles ; ensuite nous préciserons le problème de base de cette thèse qu'est la régulation de la charge dans un système parallèle.

1.1 Introduction à la thèse

Devant les limites physiques de la vitesse de calcul des microprocesseurs et les besoins de puissance pour résoudre certains problèmes (dans les applications industrielles, la simulation de systèmes dynamiques, la résolution d'équations différentielles, le diagnostic médical, les systèmes financiers, la recherche opérationnelle, le traitement d'images, etc.), le parallélisme apparaît comme une solution pour augmenter les vitesses de calcul.

Dans l'histoire du parallélisme, la conception des algorithmes parallèles va avec l'évolution des ordinateurs parallèles. Il existe des techniques pour la conception des algorithmes parallèles mais cette conception, dans beaucoup de cas, ne peut pas assurer l'efficacité de l'exécution sur une machine parallèle réelle. Une des raisons est que beaucoup de ces algorithmes ont été conçus sur un modèle de machine idéale, comme la machine PRAM (*Parallel Random Access Memory*). Il est connu que le modèle PRAM cache la plupart des contraintes architecturales permettant au concepteur de concentrer son attention sur la parallélisation de l'algorithme sans se soucier de la future implémentation sur la machine réelle (Cray, T3D, IBM_SP, ...). Cependant l'implémentation de ces algorithmes sur une machine réelle est rarement simple à réaliser, puisque dans beaucoup de cas sont nécessaires des adaptations ou modifications qui vont dégénérer l'algorithme parallèle originel.

Un aspect à prendre en considération lors de l'exécution de l'application parallèle est la gestion des ressources de la machine cible. La stratégie utilisée pour l'allocation des tâches composant un programme parallèle sur les divers processeurs d'une architecture

a un impact critique sur les performances globales du système. Pour certaines architectures de machines, des contraintes supplémentaires devront être considérées pour que, lors de l'exécution d'une application, les ressources soient mieux utilisées (cas des machines synchrones [Fon94], des machines avec une taille mémoire très limitée [Tal91], ou les caractéristiques du réseau de connexion). Aujourd'hui, dans la plupart des cas, les problèmes de gestion des ressources de la machine parallèle sont pris en charge, soit au moment de la compilation, soit par l'environnement lors de l'exécution de l'application ; dans les autres cas cette tâche est laissée au programmeur. *Cette thèse étudie l'allocation de tâches aux processeurs pour un environnement d'exécution de l'application*.

1.2 Parallélisme

Deux besoins amènent plus particulièrement le concepteur d'algorithmes au parallélisme: il a besoin de résoudre le problème en moins de temps, il cherche à résoudre un problème plus grand. L'apparition des machines massivement parallèles avait pour principale motivation la vitesse de calcul, mais vite on a réalisé que les résultats recherchés n'étaient pas simples à atteindre et que leur programmation était très complexe. Il y a eu un incroyable effort dans les mathématiques appliquées comme dans la technologie informatique pour essayer de maîtriser ces algorithmes et ces machines et un nouveau spécialiste est apparu : le programmeur des machines parallèles.

1.2.1 Les machines parallèles

On définit un système parallèle comme un système composé d'un ensemble de nœuds interconnectés, où un nœud correspond à une entité de calcul. Un des principaux objectifs des constructeurs d'ordinateurs est que les machines aillent toujours plus vite. La proposition d'utiliser plusieurs ordinateurs séquentiels connectés pour résoudre un même problème se présente comme une possibilité et la machine parallèle apparaît. Ces ordinateurs se répartissent en plusieurs classes suivant des critères différents: l'architecture d'un nœud, les différents types d'interconnexion, la gestion des flux d'instructions et de données et l'architecture de la mémoire.

Évolution du support matériel

Chaque nœud d'une machine parallèle correspond à une entité ou processeur de calcul. Le processeur séquentiel, tel que décrit par Von-Neuman dans les années quarante, est composé d'une unité de traitement, d'une mémoire et de périphériques. Le modèle reste valide bien que les processeurs actuels soient beaucoup plus complexes. Ils sont subdivisés en plusieurs unités qui résultent de la recherche de performances au moindre coût, ces unités spécialisées fonctionnant en parallèle - unité de traitement des nombres en virgule flottante, unité de traitement des nombres entiers, unité de traitement des références mémoire, par exemple.

Parmi les évolutions technologiques, les constructeurs ont conçu les **pipelines**. La notion de pipeline consiste à découper une unité fonctionnelle (traitement des instructions,

traitement des références mémoires, opérations en arithmétique flottante) en plusieurs étages de manière à ce que plusieurs données à traiter puissent l'être en même temps. Ce type de machines appelées **vectorielles** a formé la première famille des super-ordinateurs puisque grâce aux pipelines les opérations d'algèbre linéaire étaient calculées plus rapidement (comme dans les cas de la Cray1 et de la Cyber 206). Il existe aujourd'hui des compilateurs spécifiques qui permettent d'exploiter au mieux les performances de ces microprocesseurs avec une grande facilité d'utilisation. Bien que la puissance de ces ordinateurs soit très attirante, son prix reste trop élevé de sorte que peu d'utilisateurs peuvent y avoir accès. La puissance des ordinateurs scalaires classiques est de plus en plus proche des performances des ordinateurs vectoriels et petit à petit les constructeurs de super-ordinateurs choisissent d'utiliser les processeurs scalaires, moins chers, à la place des ordinateurs vectoriels.

La mémoire a vécu aussi une autre évolution. L'accès aux données est accéléré par la présence des différents types de mémoires: la **mémoire principale** et les **mémoires caches**. Les mémoires d'accès rapide, nommées caches permettent un accès très performant aux données mais elles sont chères et les constructeurs ont choisi d'utiliser des mémoires à niveaux : les mémoires caches d'accès rapide (de petite taille) gardent les données les plus utilisées jouant le rôle d'un tampon entre l'unité de traitement et la mémoire principale qui est plus lente mais moins chère.

Les architectures matérielles cibles

On s'intéresse aux architectures parallèles vues comme des ordinateurs composés de plusieurs processeurs interconnectés pouvant calculer simultanément. Il existe actuellement de nombreuses machines parallèles. Flynn a proposé dans [Fly72] une taxinomie des ordinateurs qui permet de classifier le mode de fonctionnement de ces nœuds selon deux critères: les flots d'instructions et les flots de données. Dans cette classification deux principales classes sont établies : les machines **SIMD** (*Single Instruction Multiple Data*) où les processeurs exécutent la même opération de façon synchrone sur des données différentes, exemples de cette architecture la CM2 (*Connection Machine 2*) et la Maspar ; et les machines **MIMD** (*Multiple Instruction Multiple Data*) où les processeurs de calcul participent à l'exécution de la même application mais pas forcement de la même opération dans le même instant, c'est-à-dire que les processeurs sont autonomes. La plupart des machines de la précédente génération (Paragon d'*Intel*, CM5 de *Thinking Machine*, CS2 de *Meiko*, SP2 d'*IBM*) fonctionnent aujourd'hui en mode MIMD. Les deux autres types d'architectures, SISD et MISD, sont peu répandues aujourd'hui.

Une autre caractéristique utilisée pour classifier les ordinateurs parallèles est le **mode de synchronisation**, soit les noeuds sont **synchronisés**, soit les noeuds sont complètement **asynchrones**.

Considérant ces deux classifications, sur le flot et sur la synchronisation, le type d'architecture que nous allons considérer est de type MIMD asynchrone. Un raffinement des architectures MIMD asynchrones se base sur l'organisation de la mémoire et l'interconnexion du réseau. On distingue deux types de machines : les **ordinateurs à mémoire partagée**, et les **ordinateurs à mémoire distribuée**.

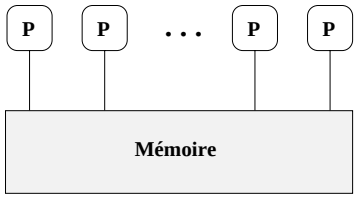


FIG. 1.1 – Ordinateur à mémoire partagée.

Dans les architectures à mémoire partagée ou **multiprocesseurs** il existe une mémoire contiguë unique accessible directement et uniformément par chacun des microprocesseurs (voir Figure 1.1), toute opération effectuée sur la mémoire d'un processeur est immédiatement visible par l'ensemble des autres processeurs. Cette mémoire globalement partagée correspond au médium de communication utilisé par les processeurs pour s'échanger des informations. Il est possible de maintenir un temps d'accès uniforme des processus à la mémoire pour quelques dizaines de processeurs. On parle alors de SMP (*Symetric Multi Processing*) ou machine UMA (*Uniform Memory Access*), au-delà de ce nombre, il est nécessaire de simplifier l'architecture de la mémoire au détriment de l'accès uniforme.

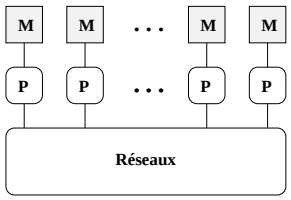


FIG. 1.2 – Ordinateur à mémoire distribuée.

Pour les ordinateurs à mémoire distribuée ou **multi-ordinateurs** (voir figure 1.2), où chacun des processeurs possède sa propre zone mémoire, les différents processeurs avec leur mémoire locale sont reliés entre eux par un réseau de communication. Dans ce type d'ordinateurs, la coopération (échange des données) et la synchronisation entre les processeurs ne peut alors se faire que par l'intermédiaire du réseau de communication. On parle alors de machine NUMA (*Non Uniform Memory Access*) où les accès à la mémoire locale du nœud sont plus rapides que les accès aux mémoires des autres noeuds. D'une manière générale, une telle machine parallèle est caractérisée par le nombre de ses processeurs et les propriétés du réseau d'interconnexion. Indépendamment des débits et latences, c'est l'uniformité de ces caractéristiques qui est importante, c'est dire que tout se passe comme si le réseau était complètement maillé et que tous les processeurs étaient à la même distance. Dans les premières machines, comme la série d'ordinateurs parallèles à base de transputers, l'accès n'était pas uniforme. Ces propriétés aujourd'hui sont généralement assurées dans les machines MPP (*Massively Parallel Processing*).

L'avantage des machines parallèles à mémoire partagée est que l'existence de cette mémoire commune entre les processeurs simplifie le travail de parallélisation des algorithmes et parce que le coût d'accès aux données est de type UMA. Cependant son principal inconvénient réside dans l'accès à la mémoire qui est un goulet d'étranglement, limitant le nombre de processeurs.

Les réseaux des communications

Dans un ordinateur parallèle à mémoire distribuée, les noeuds de calcul communiquent entre eux par des échanges de messages à travers le réseau de communication. Deux grandes familles de réseaux peuvent être distinguées: les **réseaux point-à-point** où les processeurs de calcul constituent aussi les noeuds du réseau de communication (réseaux hypercube) et les **réseaux multi-étages** où les processeurs de calcul sont les entrées et sorties du réseau (SP2, CS2, ...).

Les **réseaux point-à-point**, dont on retrouve différentes topologies, ont des contraintes technologiques et économiques qui font que le nombre de liens d'interconnexion de chaque processeur est faible. Une autre technologie, appelée machines à réseau reconfigurable est très utilisée, elle consiste à placer des *crossbar switchs* entres les différents processeurs et obtenir différentes topologies en accord avec les besoins recherchés. La communication entre deux noeuds qui ne sont pas directement connectés demande des mécanismes de routage logiciel.

Dans les **réseaux multi-étages**, les noeuds constituant le réseau diffèrent des unités de traitement. Certains réseaux, tels que le *switch* (commutateur) rapide équipant la SP2 d'IBM forcent chaque communication à passer par un nombre fixe de liens pour une taille de machine donnée. D'autres tels que le *fat-tree* de la CM5 de TMC font que la durée des communications dépend de la distance dans le réseau.

La nouvelle génération

De nombreux progrès technologiques ont changé l'aspect des machines parallèles, surtout celui des architectures à mémoire distribuée MIMD. L'arrivée des nouvelles générations de processeurs ou des réseaux à très haut débit basés sur des commutateurs a provoqué une nouvelle vague. Un nœud peut être composé par plusieurs processeurs qui partagent une même mémoire. C'est le cas pour des machines parallèles comme la CM-5, la Paragon (2 processeurs I860) ou l'IBM_SP3 (un nœud peut-être composé de 8 processeurs Power PC). L'utilisation des réseaux de commutation multi-étages permettent d'offrir un réseau d'interconnexion capable d'assurer N liaisons indépendantes en temps constant.

Dans la présente thèse, seules les machines parallèles MIMD à mémoire distribuée seront considérées [HCAL89]. Dans notre travail nous cherchons à étudier le problème indépendamment d'une topologie d'ordinateur spécifique, notre justification est que les machines parallèles ayant une topologie très contraignante forcent à une programmation spécifique [Fon94]. De plus, aujourd'hui, grâce aux avancées techniques, un réseau de stations ou grappes de calcul peut être utilisé comme une machine parallèle très puissante (évolution des unités de calcul, des réseaux de communication et de la mémoire). Dans notre cas, la machine IBM SP1 (machine disponible pour le projet APACHE) est un système réparti faiblement couplé (sans mémoire commune), avec des processeurs

homogènes. Grâce aux commutateurs qui gèrent les communications, le temps de communication entre nœuds est homogène.

1.2.2 Algorithmes et programmes parallèles

Tout d'abord pour procéder à la parallélisation d'un algorithme il faut connaître ses caractéristiques. L'écriture d'un programme parallèle portable qui calcule la solution d'un problème donné passe par la construction d'un algorithme dans un modèle de machine parallèle abstraite. Plusieurs modèles ont été proposés dans la littérature, parmi eux le modèle PRAM (*Parallel Random Access Memory*) et ses variantes sont les modèles les plus utilisés pour la construction d'algorithmes parallèles. De plus ce modèle permet d'évaluer et comparer les algorithmes. Une PRAM est une machine synchrone composée d'une infinité de processeurs reliés entre eux par une mémoire globale.

Normalement quand on modélise une machine les processeurs sont: **identiques** si les processeurs sont capables d'exécuter chaque tâche et que la durée d'exécution d'une tâche T est la même durée quel que soit le processeur, **uniformes** si les vitesses d'exécution des processeurs sont proportionnels (si un processeur est k fois plus rapide qu'un autre pour exécuter une tâche, il sera aussi k fois plus rapide pour toutes les autres tâches) et finalement **sans relation** si les processeurs ne peuvent pas exécuter indifféremment chacune des tâches ou que la durée relative d'exécution dépend des tâches considérées.

Un programme parallèle est amené à échanger des informations. Ces communications peuvent s'effectuer soit par une mémoire partagée par l'ensemble des processeurs comme dans une PRAM soit par un réseau d'interconnexion. La durée d'accès à un élément de la mémoire dans le modèle PRAM est considerée constante (accès **uniforme**) ; dans d'autres modèles, les accès sont considérés **non uniformes**. Il y a des modèles qui prennent en compte la localité des données en n'autorisant que les communications entre voisins, ce qui signifie que ce modèle prend en compte la topologie du réseau d'interconnexion.

Parallélisme de données et de contrôle

On identifie deux types de source de parallélisme: le **parallélisme de données** et le **parallélisme de contrôle**.

> On parlera de **parallélisme de données** lorsque la même opération ou la même fonction **SPMD** (*Single Program Multiple Data*) est effectuée sur des entrées différentes. Dans une machine SIMD chaque unité de calcul d'un ordinateur qui ne met en œuvre que ce mode de parallélisme est soit inactive, soit exécutant la même instruction que les autres unités sur des données qui lui sont propre [Fon94]. Le programme parallèle est une succession de phases de calculs et de phases de communications, et les phases de communications sont utilisées principalement pour la distribution des données. Il est possible de programmer une machine MIMD dans le mode SPMD, dans ce cas des mécanismes de synchronisation devront être implémentés.

La notion de **parallélisme de contrôle** est très générale. Elle consiste à découper un problème en tâches indépendantes. Ce type de parallélisme est particulièrement utilisé lorsque des opérations différentes s'exécutent sur les mêmes données ou bien sur des données différentes. Il peut aussi implanter le parallélisme de données. Les calculateurs parallèles qui permettent de choisir librement les opérations à exécuter sont de type MIMD.

Représentation d'un programme parallèle

En décrivant de façon simpliste le processus de parallélisation d'un problème, l'application est analysée, puis partitionée en un ensemble des tâches. Chrétienne [Chr94] définit " une tâche modélisera un processus de calcul qui reçoit (éventuellement) des données, exécute un calcul puis transmet (éventuellement) des résultats ". Une partie d'un programme qui peut être exécutée de façon indépendante et en parallèle, est une tâche. Il faut aussi procéder à l'analyse des dépendances, puisque les tâches peuvent être liées entre elles soit par des relations de précédence, soit par des échanges de données, soit par des relations de flot de données. La structure ainsi obtenue est un graphe qui peut être complètement connu avant l'exécution de l'application, partiellement connu ou complètement inconnu et se développer de façon dynamique au moment de son exécution. Quelques modèles de graphes sont:

Graphe de précédence: une technique qui permet de modéliser un algorithme parallèle est l'utilisation des **graphes orientés sans circuit** (*DAG - Directed Acyclic Graphs*) aussi nommés **graphe de précédence**. Ce graphe est constitué d'ensemble de tâches élémentaires (sommets) reliées par des contraintes de précédence (arcs) [CT95, Kit94]. Si la tâche i transmet un résultat à la tâche j, le graphe de précédence G contient l'arc (i,j). Si l'on a accès à toutes les informations sur l'application, chaque tâche est caractérisée par son temps d'exécution et chaque arc par un coût de communication qui modélise la quantité d'informations à échanger entre les différentes tâches.

Graphe des tâches: il existe une autre manière de modéliser un algorithme en ignorant les relations de précédence entre les tâches, c'est le **graphe de tâches**, dont les nœuds sont des tâches et les arêtes modélisent des canaux de communication entre ces tâches sans se soucier de la date d'occurrence dans le programme de ces communications. Cela revient à ignorer les précédences relatives des tâches. À ces nœuds et arêtes peuvent être associées des nombres représentant respectivement des coûts de calcul et de communication.

Graphe de flot de données: la précédence entre tâches est induite par les circulations de données. Typiquement, les tâches correspondent à l'évaluation d'une instruction et les précédences aux accès en lecture ou écriture des opérandes.

Le modèle du graphe de précédence n'est pas général, car tous les algorithmes ne s'y réduisent pas. Certains algorithmes possèdent une partie complètement dynamique. C'est-à-dire que la génération des tâches va dépendre, entre autres, des données en entrée et on se retrouve en face d'un **graphe de précédence dynamique**. Dans ce document

la représentation utilisée sera le graphe de précédence dynamique. Une définition plus formelle du graphe des précédence s'exprime ainsi:

Définition 1.1 *un programme parallèle peut être vu comme un ensemble de tâches $T = \{T_1, \ldots, T_n\}$ liées par des contraintes de précédence. L'ensemble des tâches correspond à une partition du programme et chaque précédence $T_i \prec T_j$ indique que T_j doit accéder en entrée à des données calculées par la tâche T_i. On dit alors que T_i est un prédécesseur de T_j. Il est possible de partitionner l'application pour que la relation $\prec$ définisse un ordre partiel sur l'ensemble des tâches [Sar89].*

L'application est alors représentée par un graphe orienté sans cycle $G = (T, E, (a_i), (c_{ij}))$, où E est l'ensemble des arcs $T_i \rightarrow T_j$ induit par $T_i \prec T_j$, et c_{ij} dénote le coût de transfert des données de T_i vers T_j. La quantité a_i désigne le temps d'exécution séquentiel de la tâche T_i.

Ces deux coûts c_{ij} et a_i seront établis à la compilation dans le cas statique, par contre dans le cas dynamique ces valeurs sont inconnues avant l'exécution. Cependant dans certains cas on peut estimer un coût (à partir de la complexité de l'algorithme) et ces valeurs pourrons être utilisées au moment de l'exécution pour établir une meilleure allocation.

La topologie (ou structure) de ce graphe (arbre, hypercube, grille, etc.) est une des caractéristiques importantes de l'algorithme. Typiquement, l'exécution d'un tel algorithme sera telle que les processus exécutent en séquence les opérations sur les chemins du graphe [CT95].

Régularité ou Irrégularité

Les applications parallèles seront classées selon leur comportement. L'application vue comme un graphe peut être statique ou dynamique, peut avoir une structure (topologie) régulière ou irrégulière et les tâches qui la forment avoir un coût équivalent ou non.

Nous appellerons **application parallèle régulière** une application dont nous pouvons prédire le comportement: si le graphe de tâches et la durée des tâches sont connus ou prévisibles et s'ils ne changent pas d'une exécution à l'autre. Alors qu'une **application parallèle irrégulière** est une application pour laquelle on ne peut pas prédire le comportement indépendamment d'une instance particulière. Cette impossibilité de prédiction est due soit à ce que le nombre de tâches est variable en fonction du problème, soit que le coût des tâches n'est pas prévisible, ou encore à l'indéterminisme des dépendances entre les différentes tâches (cas de Prolog [Kan96], B&B [Bou98]). Dans ce contexte d'irrégularité, le contrôle de l'application est plus complexe.

1.3 Les modèles de programmation

Pour une machine donnée, une programmation adaptée est une prémisse indispensable pour assurer l'utilisation des capacités du calculateur parallèle. Un des buts principaux du parallélisme est d'arriver à dégager un modèle de programmation qui, pour un large spectre de programmes, puisse être traduit automatiquement dans le langage de

la machine et exécuté efficacement pour une large classe de machines [Val90]. Ce modèle est loin d'être trouvé, bien que le modèle de programmation BSP (*Bulk Synchronous Program*, introduit par Valiant [Val90]) soit une proposition dans cette direction [McC95, Val90]. Ce modèle définit un ensemble d'unités indépendantes de calcul qui peuvent communiquer grâce un routeur de messages, chaque phase de calcul est suivie par une phase de communication de données qui leur permet d'itérer une nouvelle phase de calcul. Les processeurs peuvent être synchronisés après un nombre (établi par le programmeur) de pas de calcul, chaque séquence de calcul peut se découper en sous-calculs en collectant à la fin les résultats de ceux-ci. La phase de synchronisation est globale à tous les processeurs et elle est effectuée pour déterminer si tous les pas de calculs sont terminés par toutes les unités [McC95]. Chaque phase de synchronisation peut être exprimée par des schémas structurés de communication (diffusion, réduction, permutation, etc.).

Cependant un modèle de programmation universel n'existe pas pour le calcul parallèle. Tous les modèles architecturaux et les modèles théoriques associés définissent autant de modèles de programmation. Il existe aussi des compilateurs qui traduisent automatiquement des programmes séquentiels en programmes parallèles, pour un certain type de programmes, cette approche du parallélisme implicite permet d'obtenir de bons résultats.

1.3.1 Modèle du parallélisme de données

Le modèle de programmation du parallélisme de données, est basé sur la possibilité d'effectuer le même calcul sur des données différentes. Cette approche fonctionne correctement lorsque les données manipulées sont régulières (matrice dense, matrice structurée par bande, etc.). Un langage est basé sur ce modèle HPF [Hig93] (*High Performance Fortran*). Le principe de HPF est d'utiliser le parallélisme engendré par une découpe des données et leur distribution sur plusieurs processeurs.

1.3.2 Modèle à processus communiquants

Ce modèle, l'un des plus populaire, est basé sur la notion de processus communicants [Bal91]. Dans ce paradigme, un programme parallèle est un ensemble de processus qui concourent à la résolution d'un problème unique. Chacun des processus possède une zone de mémoire privée, et la coopération s'exprime par des communications explicites entre processus. Les bibliothèques de communication PVM [Sun90] et MPI [Mes95] sont basés sur ce modèle. Elles offrent un ensemble de primitives qui structurent les schémas de communication (envoi, réception, diffusion) et qui peuvent être adaptées efficacement pour une machine parallèle particulière.

1.3.3 Modèle par appel de procédure

Dans ce modèle l'unité de base est la procédure (ou fonction). Un processus appelant (le client) peut appeler une procédure exécutée par un autre processus. La procédure s'exécute " à distance " et ses résultats sont transmis à l'appelant. Les appels peuvent

être synchrones, si le processus appelant est bloqué jusqu'au retour des résultats et asynchrones si le processus appelant continue son exécution et peut récupérer les résultats en se mettant explicitement en attente du résultat. La première maquette du projet Apache ATHAPASCAN0a [Chr96] a été construite sur cette vision.

1.4 Le contrôle de l'application

Le noyau exécutif, ou environnement d'exécution, est responsable de l'exécution de l'application. Dans le cas parallèle, différentes contraintes sont à prendre en compte, comme la localité des données, les communications entre les tâches, la mémoire disponible. Un des problèmes principaux lors de l'exécution d'un algorithme sur une architecture de type MIMD à mémoire distribuée est celui du contrôle de la charge des noeuds en terme de tâches à y exécuter. En effet le temps global d'exécution d'un algorithme parallèle peut être très significativement dégradé par une mauvaise utilisation des ressources (réseau, mémoire et capacité de calcul).

Le graphe résultant de l'algorithme (graphe de précédence statique ou dynamique) lors de l'exécution est distribué entre les processeurs de la machine. L'idéal serait que cette opération soit à la charge d'un outil d'ordonnancement ou **ordonnanceur** (*scheduling*) de façon automatique et transparente. Dans des systèmes parallèles ou distribués existants, ce traitement intervient à deux moments différents: soit à la compilation si le graphe a été complètement déterminé statiquement (**ordonnanceur statique**) ou soit au moment de l'exécution (**ordonnanceur dynamique**); différentes techniques sont utilisées dans chaque cas comme on verra dans le chapitre deux.

Quand l'utilisateur construit son application, il serait intéressant qu'il ait la possibilité de donner certaines informations qui pourront être utilisées par le noyau exécutif pour essayer de faire un meilleur choix pour le placement des tâches, comme le coût estimé de la fonction de calcul et le coût estimé des communications.

1.4.1 Granularité

Lors de la description d'un algorithme parallèle sous la forme d'un graphe de tâches, choisir la **granularité** consiste à définir plus précisément ce qu'est une tâche. Empiriquement, on peut définir la granularité comme le rapport entre la quantité de calcul mise en parallèle sur le coût de sa mise en marche ou **latence d'allocation**. La latence d'allocation est le coût à payer pour installer la tâche avec ses données, elle est fortement liée au volume des communications, c'est pourquoi la granularité est définie comme le rapport entre le temps de calcul de la tâche divisé par le temps des communications nécessaires à son exécution [Ber97].

Il peut être intéressant de diminuer la granularité en découpant les tâches en sous-tâches plus petites. De cette façon on augmente le parallélisme potentiel, mais si les séquences de calcul sont trop courtes et si le temps de latence nécessaire pour démarrer une tâche est trop important, on est en présence d'un grain de parallélisme trop fin. A l'inverse si l'on regroupe les tâches pour accroître la granularité, on parle de grain plus gros,

la latence d'allocation peut-être négligeable, le nombre de tâches est diminué et le parallélisme potentiel est limité. Un grain trop fin entraîne souvent trop de communications entre les processeurs et un grain trop gros restreint le parallélisme.

Le choix de la granularité est un compromis entre le maximum de parallélisme potentiel, un surcoût de gestion à minimiser, et un bon équilibre de charge entre les noeuds de calcul.

Il peut être intéressant de créer un mécanisme qui permettre de réguler la granularité de l'application à partir des informations sur l'état de charge du système. Quand les processeurs sont trop chargés, on cherche à obtenir un gros grain et quand on a beaucoup de processeurs libres, on cherche à produire un grain fin. Cela implique d'adapter le parallélisme potentiel de l'application à la charge du système.

1.4.2 Communications et localité des données

Les besoins de mécanismes de communication sont une des principales différences entre les ordinateurs parallèles et séquentiels. Les développeurs d'applications parallèles ont besoin d'outils de communication efficaces et portables. Efficaces pour les performances et portables afin de proposer leurs applications sur des architectures différentes et d'en garantir la pérennité quand les plates-formes évoluent. [McB94]. Quand entre deux tâches, localisées dans deux processeurs différents d'une machine à mémoire distribuée, s'établit une communication, tout un mécanisme de routage est développé en passant par les différentes couches logicielles et physiques qui assurent cette communication. Quand une application parallèle est exécutée sur une machine à mémoire partagée, les communications de l'application sont transformées par des lectures et écritures dans la mémoire commune.

Pour un message passé entre deux tâches, on appelle **latence** le temps qui s'écoule pour installer la communication, sans compter le temps de transfert des données. Dans le modèle classique où la durée d'une communication est une fonction affine de la taille des données envoyées, la latence est la durée de l'envoi d'un message de taille nulle. Le **débit** est la quantité de données pouvant être transférée par unité de temps par une communication. Le débit est souvent mesuré en octets par seconde et peut varier suivant les types de communication et la taille des messages envoyés. Des bibliothèques de communication pour le calcul parallèle ont été développées, deux de ces bibliothèques les plus utilisées sont le standard MPI (*Message Passing Interface*) et la bibliothèque PVM (*Parallel Virtual Machine*).

Généralement, lors de l'exécution d'une application parallèle, le surcoût dû aux communications est lié au problème de la **localité des données** (*locality*). La localité des données d'un algorithme parallèle correspond au rapport des coûts en calcul (travail parallèle) et du coût en communication (volume total de communications). Deux types de localité existent pour les données, la localité temporelle (dans le temps) et la localité spatiale (partage entre les nœuds de calcul), dépendant du comportement du programme.

Plus la localité d'un programme est importante, moins un processeur communique des données. Le placement d'un graphe de tâches quelconque sur les processeurs prenant en compte la localité des données permet de réduire les coûts des communications entre

les processeurs. Le jeu consiste à placer les tâches qui partagent les données sur le même processeur. Si cette propriété ne peut pas être respectée, le coût des communications se mesure par accès aux différentes mémoires où sont localisées les données. Si les accès se font à une mémoire locale, leur coût sera plus faible que s'ils doivent se faire à mémoire distante.

La programmation des algorithmes non locaux nécessite des réseaux de communication à hautes performances. Aussi longtemps que le surcoût des communications sera significatif sur les architectures parallèles actuelles, l'obtention d'algorithmes performants dépendra en grand partie de la maîtrise de la localité [Bri96]. Pour les systèmes avec mémoire partagée, l'échange de données est transparent, puisqu'il consiste à lire directement dans la mémoire pour récupérer les données, en principe il n'existe pas de surcoût, par contre, le nombre de processeurs disponibles dans ce type d'architecture est limité.

1.4.3 Performance et efficacité

La **performance** et l'**efficacité** sont deux critères recherchés lors de l'exécution d'une application en parallèle. Dans le premier cas, le but est d'obtenir un temps d'exécution minimal de l'application [Ber91], dans le second, il s'agit de maximiser l'utilisation des ressources du système parallèle. L'allocation d'une application vise à ce que les tâches composant l'application soient exécutées en maximisant l'utilisation des processeurs (critère de distribution de charge) tout en minimisant le coût de communication (critère de localité), pour obtenir un temps de réponse minimal [Tal94]. Cependant la recherche d'une meilleure efficacité peut diminuer les performances en temps d'exécution.

Pour connaître l'efficacité d'un algorithme parallèle par rapport à un algorithme séquentiel, il faut faire des mesures de performances. L'accélération mesurée d'un algorithme sur *p* processeurs (A_p) est définie comme le rapport du temps d'exécution séquentielle de l'algorithme, T_{seq}, divisé par le temps d'exécution parallèle de l'algorithme sur *p* processeurs, T_p.

$$A_p = \frac{T_{seq}}{T_p}$$

où T_p est le maximum des temps d'exécution mesurés sur chacun des processeurs.

L'efficacité d'un algorithme (E_p) est définie comme le rapport de l'accélération de l'algorithme sur p processeurs divisé par le nombre de processeurs. Il est souvent exprimé par un pourcentage qui représente l'utilisation moyenne des processeurs par rapport une parallélisation parfaite. Une parallélisation parfaite implique que tous les processeurs sont en permanence utilisés pour effectuer des opérations utiles pour l'application.

$$E_p = \frac{A_p}{p} = \frac{T_{seq}}{p\,T_p}$$

1.5 La régulation de la charge

Centrant notre attention sur l'étude des mécanismes de partage de charge dans les systèmes de programmation parallèle nous avons été confrontés à une volumineuse litté-

rature disponible et en état d'évolution qui nous a obligée à nous positionner au sein d'un domaine qui évolue de façon vertigineuse. Le problème de la régulation de charge a été étudié depuis plus d'une vingtaine d'années dans les domaines des systèmes répartis et parallèles. Profitant de cette expérience nous faisons un petit parcours. Dans leur article [CK88], Casavant et Kuhl présentent le problème général de la répartition de la charge de façon très simple, d'un côté se trouvent les ressources physiques du système (mémoire, périphériques, réseaux, processeurs), d'un autre les besoins de l'application (les consommateurs), et au milieu un dernier composant est la politique d'allocation qui décide où seront allouées les tâches pour accéder aux ressources.

Nous proposons maintenant une abstraction du problème et décrivons-le de la façon suivante : supposons un problème de taille N (N est le nombre de **tâches indépendantes**, deux tâches sont indépendantes s'il n'existe pas de relation de précédence entre elles) et P le nombre de processeurs de la machine, le problème est de répartir les N tâches sur les P processeurs suivant un critère de qualité retenu. On trouve dans la littérature des solutions exactes ou des heuristiques pour réaliser cette répartition. Le problème est connu comme NP-difficile dans le cas général [GJ79].

1.5.1 L'objectif de la stratégie d'allocation

Dans ce premier aperçu les tâches sont indépendantes, et le problème est limité à une étude d'allocation des ressources, par contre quand on inclut de nouvelles contraintes, comme c'est le cas du graphe de précédence (prise en compte des communications ou de la localité des données), il est nécessaire utiliser des techniques plus complexes.

Le but principal de l'allocation des tâches est l'amélioration des **performances** de l'application parallèle, généralement quantifiées par la durée du programme. Deux autres objectifs sont par exemple de respecter les dates limites lors d'une application en temps réel ou de mettre en place tous les mécanismes nécessaires pour supporter la tolérance aux pannes (caractéristique principale dans certains systèmes d'exploitation).

1.5.2 Définitions et taxinomie

Le grand nombre de publications a généré une vaste terminologie utilisée pour décrire le problème de l'allocation des tâches, présentant une difficulté pour établir des relations entre les différentes propositions et incluant des erreurs dans les analyses. Sont présentées ici les définitions de termes utilisés dans ce document.

Partage ou équilibrage de la charge

Pour Eager [ELZ86] l'amélioration des performances du système peut être recherchée par deux moyens, le **partage de charge** (*load sharing*) et l'**équilibrage** (*load balancing*) de charge.

> Dans le **partage de la charge**, on cherche à maintenir la plupart du temps tous les processeurs occupés. Ceci est réalisé en transférant une partie de la charge de travail de la machine surchargée vers d'autres machines. Le placement ou

la migration de processus n'est donc envisagé que lorsque la charge de travail local dépasse un certain seuil.

Par contre dans l'**équilibrage de la charge** l'allocation des processus est envisagée chaque fois que les conditions globales du système changent, c'est-à-dire à chaque création ou terminaison de processus. Un système très fin de mise a jour de l'état du système doit être construit mais il est très coûteux.

Nous pouvons penser que le système est mieux utilisé lorsque chaque ressource a la même charge de travail. Si le coût de mise en œuvre de la répartition du programme est ignorée, l'équilibrage de charge donnera de meilleurs résultats que le seul partage de la charge [KL87]. Malheureusement, le coût de mise en œuvre d'une stratégie d'équilibrage de charge dans les systèmes qui nous intéressent (MIMD à mémoire distribuée), est important, à cause des délais de communication entre les machines nécessaires pour la transmission d'informations de contrôle et du transfert de processus.

Dans ce document nous utiliserons comme synonymes les mots **partage**, **répartition**, ou **distribution** de charge où l'idée de base est de **répartir** la charge entre les processeurs en évitant qu'un nœud ne soit inactif alors que des tâches restent en attente sur d'autres nœuds. **Équilibrer** est la recherche d'une distribution de la charge équitable parmi les processeurs de la machine parallèle à tout moment. La **charge** peut être des données, des processus, des commandes, des tâches.

Krueger [KL87] affirme que les algorithmes de partage de charge conduisent en général à de meilleurs résultats que les algorithmes d'équilibrage de charge, puisque évitant le transfert des tâches qui ne sont pas "rentables". Les transferts "non rentables" sont ceux qui n'apportent pas un gain suffisant sur le temps d'exécution pour justifier leur coût de transfert.

Le terme de **régulation** de la charge (un terme très utilisé en automatique) inclut l'idée d'autorégulation. Cela signifie que lorsque l'on constate d'un déséquilibre, on procède à une nouvelle répartition du travail (on remarque une connotation dynamique intrinsèque) et finalement on utilise le terme d'**ordonnancement** lorsque la répartition de la charge consiste à établir un ordre d'exécution pour les tâches qui forment un algorithme (grâce à la construction du graphe de précédence) en assignant le processeur où la tâche devrait être exécutée mais aussi la date de début d'exécution de cette tâche. Cette date peut-être une date physique ou une date logique.

On nommera **ordonnanceur** l'outil ou mécanisme (programme) qui permet de réaliser la gestion des ressources ou régulation de la charge en mettant en œuvre un ou plusieurs algorithmes d'ordonnancement. Cet ordonnanceur est normalement partie intégrante de l'exécutif. Dans le cas statique il est utilisé au moment de la compilation.

Quelques caractéristiques des ordonnanceurs

Devant la diversité des méthodes possibles d'allocation de la charge, Casavant et Kuhl [CK88] ont proposé une classification très complète des différents modèles d'ordonnanceurs dans le cadre de systèmes distribués, mais elle peut aussi être appliquée pour les systèmes parallèles. Cette classification permet une approche comparative du point de vue de l'implantation mais par contre n'est pas un moyen pour réfléchir à nouvelle proposition

pour la création d'un mécanisme de régulation. Toutefois nous reprenons les critères que nous considérons les plus importants.

Dans chaque processeur existe un ordonnanceur local qui gère l'exécution des processus concurrents. Nous ne ferons pas mention dans ce travail de cet ordonnanceur propre au système d'exploitation, mais de l'ordonnanceur qui est appelé global [KL87] et qui doit gérer les ressources d'un ensemble de processeurs. Cependant il existe des travaux qui essaient de faire coopérer ces deux niveaux d'ordonnanceurs [McC95, Fol92], en remarquant que lors de l'exécution et quand existent dans un même processeur des processus concurrents et prêts à s'exécuter, le résultat peut dépendre de l'ordre dans lequel ils seront exécutés localement. Certains travaux visent à donner des priorités aux tâches, même en présence d'autres tâches concurrentes [Den95].

On distingue les stratégies statiques et les stratégies dynamiques, selon que l'allocation des processus s'effectue avant l'exécution ou si elle est régulée pendant l'exécution. En parallélisme, l'ordonnancement statique a une longue histoire. Il est encore très utilisé et étudié. Des résultats très satisfaisants ont été obtenus du point de vue théorique et pratique [Bou94, Gui95]. Pour notre travail nous nous concentrerons seulement sur les ordonnanceurs dynamiques, puisque nous cherchons répondre aux besoins des applications irrégulières. Pour ce type d'ordonnanceurs nous énumérons les critères qui permettent de décrire les implantations les plus courantes :

1. La distribution : une stratégie est distribuée si les décisions d'allocation sont prises par les processus concernés et elle n'est pas distribuée si c'est un contrôleur(s) qui a la responsabilité de la décision de répartition de la charge.
2. La coopération : il y a des stratégies coopératives, ce qui signifie que deux niveaux du système essayent de trouver la meilleure allocation.
3. L'adaptation : les stratégies adaptatives ont la caractéristique de s'adapter au changement de la charge du système, de mettre en jeu des modifications dans la politique pour répondre au mieux au besoin du système.
4. La redistribution : certaines stratégies sont définitives ce qui signifie qu'après la décision d'allocation, on ne remet pas en cause la décision prise même si la tâche n'a pas encore débuté. On trouve des stratégies révisables, qui mettent en cause les décisions prises si cela est nécessaire. Pour ces dernières des mécanismes pour la redistribution sont nécessaires comme la migration ou la duplication des processus.

En résumé à partir des critères proposés par Cassavant et Kuhl les ordonnanceurs qui nous intéressent sont du type global, dynamique, qui peuvent être distribués ou non, avec la possibilité d'être adaptatifs, avec ou sans redistribution.

1.6 Conclusion

Les ordinateurs parallèles à mémoire distribuée sont actuellement les seules machines capables de simuler des problèmes de grande taille à des coûts raisonnables. Pour obtenir de bonnes performances, il faut que l'exécution du programme parallèle soit efficace.

Il faut souligner que les problèmes d'ordonnancement sont extrêmement divers, et qu'il n'existe pas de théorie générale de ce domaine. Dans le cas général ils sont NP-difficile.

Chapitre 2

Ordonnancement des applications parallèles

Ce chapitre commence par une définition formelle du problème de l'ordonnancement, en considérant des aspects théoriques pour qualifier les performances d'un algorithme parallèle. Ensuite nous présentons un parcours des algorithmes d'ordonnancement statiques et puis on étudie plus à fond les algorithmes dynamiques, terminant avec la description de la structure d'un régulateur dynamique de charge.

2.1 Le problème de l'ordonnancement

Le problème de l'**ordonnancement** consiste à programmer l'exécution d'une application parallèle en attribuant des ressources de calcul aux tâches et fixant leurs dates d'exécution [Car88]. Cette allocation doit respecter les contraintes de précédence du graphe qui représente l'application. Le critère qu'on cherche à optimiser va influer dans le choix du site d'exécution. Un placement peut-être considéré comme la première phase de l'ordonnancement, c'est-à-dire l'affectation des tâches aux ressources. Si la stratégie de placement dynamique ne permet pas la redistribution (mise en cause des choix pris) on est en présence d'une politique définitive. Une remise en cause peut être nécessaire, surtout dans le cas des applications irrégulières, où la création de nouvelles tâches n'est pas prévisible. Pour la redistribution des mécanismes spéciaux peuvent être nécessaires, par exemple la migration d'une tâche déjà débutée. Nous considérons le cas où une tâche qui a été placée mais n'est pas encore débutée peut-être redistribuée de façon dynamique. Dans [CT95] les auteurs définissent le placement d'une application formée par T **tâches indépendantes**[1] de la façon suivante :

Définition 2.1 *Un placement est une application (notée alloc) qui à une tâche t associe un processeur q.*

$$alloc : T \rightarrow P, \forall t \in T, \exists p \in P, alloc(t) = p$$

[1] Deux tâches sont indépendantes s'il n'existe pas une dépendance causale entre elles.

où T est l'ensemble des tâches à placer et P l'ensemble des processeurs.

La recherche d'un placement se fait sur l'ensemble de tous les placements possibles. Si $|P|$ est le nombre de processeurs et $|T|$ le nombre de tâches, alors il existe $|P|^{|T|}$ placements possibles. En général $|T| \geq |P|$.

Dans la théorie de l'ordonnancement, il est connu que trouver la solution optimale pour un problème de placement est NP-difficile en général [Kun91], c'est-à-dire qu'il n'existe pas d'algorithmes exacts pour résoudre les problèmes pour lesquels le temps maximal de résolution est borné par une fonction polynomiale de la taille du problème [2][GJ79, Bok81].

Définition 2.2 *L'ordonnancement d'une application doit décrire une exécution valide, respectant les contraintes de précédence du graphe, sur une machine parallèle donnée, dans le but de minimiser une fonction objectif, par exemple le temps total d'exécution, notée C_{max}. À chaque tâche t_i doit être alloué un processeur p_j et spécifié une date logique de début d'exécution notée $deb(t_i)$.*

Les différentes données d'un problème d'ordonnancement sont : les tâches, les contraintes potentielles (voir graphe des précédences 1.2.2), les ressources (la machine parallèle) et une fonction objectif. Quand on procède à la représentation d'une application comme un graphe G de précédence (voir définition 1.1), on suppose que chaque tâche a un coût correspondant au travail à effectuer. On distingue deux types d'algorithmes d'ordonnancement suivant l'instant où le choix d'allocation est décidé. Alors que l'ordonnancement statique place les processus à la compilation ou au chargement du programme, l'ordonnancement dynamique réalise l'allocation à l'exécution. Les politiques d'ordonnancement statique requièrent une connaissance complète de l'application (du coût des tâches, du coût de communication et de la relation entre les tâches) à priori. Quand ces informations ne sont pas disponibles au début de l'exécution ou quand on ne connaît pas le nombre de processeurs qui forment la machine parallèle, on utilise les politiques d'ordonnancement dynamique. De nombreux algorithmes statiques ont été développés pour le calcul d'un ordonnancement lors de la compilation du programme [Sar89]. Cependant, pour de nombreuses applications les temps de calcul, de communication ou la structure du graphe de précédence G, ne peuvent être déterminés avant l'exécution du programme.

La détermination du type de graphe de tâches permet de mieux cerner les techniques de régulation à utiliser. Les coûts de communication et de calcul peuvent être prévisibles ou imprévisibles. Si le graphe est **prévisible** les techniques statiques sont bien adaptées [YG92, Bou94]. Dans le cas où le graphe est **semi-prévisible** les techniques statiques peuvent aider à faire un premier placement, mais cette solution n'est pas satisfaisante parce qu'on peut perdre toutes les performances par un manque de connaissance parfait du graphe, et les techniques dynamiques sont alors utilisées. Pour un graphe **imprévisible** il est nécessaire d'utiliser des techniques de régulation dynamique de charge.

[2] sauf si $P = NP$.

2.1.1 Évaluation théorique

Différents aspects sont étudiés pour évaluer la qualité d'un algorithme d'ordonnancement. Nous allons ici décrire deux critères qui ont été utilisés pour analyser le problème de l'ordonnancement: l'α-compétitivité et le surcoût dû à l'ordonnancement.

α-compétitivité

J.-L. Roch [Kon97] définit par compétitivité :

Définition 2.3 *Soit h un algorithme d'ordonnancement. La compétitivité de h, noté R(h), est la valeur maximale sur tous les graphes, du rapport entre la durée donnée par l'ordonnancement fourni par h et la durée de l'ordonnancement optimal, c'est-à-dire* $R(h) = Max\frac{C(h)}{C^{opt}}$.

On dit qu'un algorithme h est α*-compétitif si* $R(h) \leq \alpha$*. Plus R(h) est petit, plus l'algorithme est efficace* [3].

Surcoût dû à l'ordonnancement

Remarque 1 *Le principe de Brent [CT95] affirme que tout programme parallèle qui nécessite* x *unités d'opérations séquentielles qui ne peuvent pas être parallélisables et effectuant* N *opérations pour résoudre un problème donné, peut être ordonnancé sur* P *processeurs pour être exécuté en temps correspondant* $[N/P] + x$ *opérations. Néanmoins, le principe de Brent ne prend pas en compte le surcoût du calcul d'un tel ordonnancement.*

Idéalement un algorithme d'ordonnancement devrait satisfaire le fait que le surcoût lié au calcul de l'ordonnancement soit négligeable devant le temps global de l'exécution parallèle. Cependant la gestion des tâches est un problème important, surtout quand il est nécessaire de gérer l'exécution d'un graphe de précédence dynamique et imprévisible.

Définition 2.4 *Soit h un algorithme d'ordonnancement et G un graphe de précédence quelconque. Le surcoût d'ordonnancement de h, noté* $\sigma(h)$*, est le nombre d'opérations effectuées par h pour exécuter G.*

Dans le cas d'une liste de tâches indépendantes et de coût égal, une allocation cyclique (modulo p) permet sans surcoût d'obtenir un ordonnancement. Cependant, dans le cas où le graphe de précédence ne pourrait être déterminé que par l'exécution du programme, l'utilisation d'un ordonnanceur va introduire un surcoût. Ce coût est lié au nombre de tâches et peut être séparé en deux parties : d'une part les opérations requises pour le calcul de l'ordonnancement et d'autre part celles utilisées pour exécuter le graphe de tâches en garantissant le respect de la relation de précédence.

Dans le cadre statique, le graphe étant connu à la compilation, le calcul de l'ordonnancement n'implique pas de surcoût lors de l'exécution, puisque tout a été calculé à l'avance.[4] En revanche, son contrôle nécessite un surcoût lié à la taille du graphe. Dans le

[3] C représente la fonction objectif à minimiser.

[4] Notons que le calcul de l'ordonnancement statique peut-être très cher [YG92, Bou94].

cadre dynamique, le surcoût de contrôle du graphe s'ajoute à celui du calcul de l'ordonnancement.

Le surcoût de l'ordonnancement dynamique comprend non seulement le coût de gestion des tâches (création, affectation des tâches, dépendances, terminaison) mais aussi celui de la gestion de la charge du système (évaluation de la charge, placement). Donc l'exécution d'un programme sera une succession de phases de calcul et d'ordonnancement. Ces phases peuvent être exécutées de manière asynchrone et les schémas qui vont suivre peuvent être réguliers ou irréguliers [Bri96].

2.1.2 Algorithmes de liste

La dénomination **algorithme de liste** est une référence explicite au principe de calcul des ordonnancements utilisant une **liste de tâches**. C'est une des idées les plus simples, elle consiste d'abord à établir une liste des tâches rangées par un critère (priorités décroissantes, coût, localité) puis à ordonnancer les tâches en appliquant la règle suivante : " Dès qu'au moins une tâche est exécutable, exécuter parmi celles-ci la plus prioritaire de la liste " [Car88]. Ces méthodes sont très utilisées en pratique car elles permettent d'obtenir rapidement une (ou même plusieurs) solutions. En considérant le graphe de précédence, dans un algorithme *glouton*, on traite les tâches en suivant l'ordre partiel induit par les contraintes de précédence, depuis les tâches sans prédécesseur vers les tâches sans successeur. Lorsque la date de début d'exécution d'une tâche ($deb(t)$) et le processeur sur lequel elle doit être exécutée ont été déterminés, on dit que la tâche est ordonnancée. L'algorithme est dit *glouton* car il affecte les dates de début d'exécution et les processeurs aux tâches dans l'ordre croissant de leurs dates de disponibilité, sans remettre en cause ses décisions précédentes.

La liste peut être construite explicitement dans une première phase ou bien rester virtuelle et n'être élaborée qu'au fur et mesure du calcul de l'ordonnancement. Le théorème suivant est un classique de l'ordonnancement. Graham a utilisé un algorithme de liste, et dans la modélisation les communications entre les tâches sont ignorées.

Théorème 2.1 *(Graham 69) pour un graphe acyclique quelconque, la durée d'un ordonnancement de liste sur p ($p \geq 3$) processeurs est au pire:*

$$\omega \leq \omega_{opt}(2 - \frac{1}{p})$$

où ω_{opt} est la durée du meilleur ordonnancement possible.

Pour la démonstration de ce théorème on dirige le lecteur vers [CT95].

Un autre résultat fondamental pour les ordonnancements de liste, en utilisant le chemin critique cette fois, se trouve dans une méthode proposée par Coffman et Graham [CG72].

Théorème 2.2 *(Coffman-Graham 72), sous l'hypothèse p=2, l'ordonnancement avec un algorithme de liste basé sur le chemin critique pour un graphe de précédence acyclique quelconque avec des tâches de durées unitaires ou UET (Unit Execution Time) est optimal.*

Par contre si la modélisation prend au compte un délai dû aux communications, le modèle le plus fréquent revient à considérer un graphe de précédence valué, dont le coût des arcs est proportionnel aux tailles des données échangées entre les tâches (que l'on note $C(t,t')$). Ainsi, pour les tâches t_i et t_j, telles que $t_i \prec t_j$, on aura la relation générale suivante :

$$\begin{aligned} deb(t_j) &\geq deb(t_i) + exec(t_i) \qquad si \qquad alloc(t_i) = alloc(t_j) \\ deb(t_j) &\geq deb(t_i) + exec(t_i) + C(t_i,t_j) \qquad sinon \end{aligned}$$

Où $deb(t_i)$ est la date logique de début d'exécution, et $exec(t_i)$ correspond au coût d'exécution. Si les deux tâches sont allouées dans le même processeur il n'aura pas de coût à payer pour les communications entre t_i et t_j. La modélisation considère tous les processeur identiques, de façon que le coût d'exécution $exec(t_i)$ est le même peu importe le processeur où elle sera placée.

Un ordonnancement doit respecter les précédences entre les deux tâches t_i et t_j en assurant que t_j débute son exécution après la terminaison de t_i. De plus si ces deux tâches ne sont pas allouées au même processeur, un délai de communication, modélisant le surcoût temporel des transferts de données inter-processeurs, doit séparer la terminaison de t_i du début d'exécution de t_j, noté $C(t_i,t_j)$.

La modélisation du problème de l'ordonnancement est importante, ici nous avons énoncé trois résultats classiques, deux pour les algorithmes de listes et un pour l'ordonnancement d'un graphe de précédence valué qui prend en compte les communications. Ils ne sont pas les uniques résultats qui existent et ils constituent un domaine de l'ordonnancement indispensable. Cependant plus la modélisation veut prendre en compte les facteurs qui interviennent pour l'ordonnancement d'une application sur une machine parallèle réelle, plus l'analyse devient complexe. Il est fréquent de trouver dans la littérature des algorithmes optimaux mais qu'il est impossible d'implémenter du fait des hypothèses théoriques considérées par les auteurs [Chr94].

2.2 Ordonnancement statique

Hypothèses du problème

- une application est généralement représentée par un graphe prévisible de tâches. Ce graphe est indépendant des données en entrée de l'application ;
- les coûts de calcul et de communication sont connus a priori et obtenus du compilateur ou de l'éditeur de liens.
- la machine parallèle est supposée dédiée à l'exécution de l'application

Les politiques statiques ont été largement étudiées dans la littérature [Bou94, YG92, Chr94, CT95, ST85], et les caractéristiques de ce type de problème sont très spécifiques. La connaissance de l'application inclut normalement la description du graphe de précédence et des informations sur chaque tâche exprimant les différents coûts (calcul et communication). Il faut noter qu'après le début de l'exécution on ne remet pas en cause les décisions prises lors de l'ordonnancement.

La plupart des stratégies d'ordonnancement statique sont classées en deux catégories : les méthodes exactes et les méthodes approchées (heuristiques). On a donc le choix entre deux grandes alternatives, chercher l'optimal en risquant l'explosion combinatoire ou bien se contenter d'un ordonnancement approché, trouvé en un temps raisonnable. Les méthodes exactes, type *Branch&Bound* ou des méthodes de la théorie des graphes comme les couplages, ou la programmation mathématique, sont restreints aux problèmes de petite taille du fait du phénomène d'explosion combinatoire [Bou94]. Elles sont inspirées des domaines de la recherche opérationnelle et de l'intelligence artificielle.

2.2.1 Les algorithmes exacts

Le principe des algorithmes **exacts** [ST85] repose sur une exploration explicite de toutes les solutions possibles. Cette approche doit conduire à la solution optimale, mais est très coûteuse en pratique et inemployable pour des exemples trop grands. Ces algorithmes représentent sous forme d'arbre la construction de toutes les solutions possibles. En partant de la configuration où aucune tâche n'est placée (la racine) vers l'ensemble des tâches placées (les feuilles), différentes explorations de cet arbre sont possibles. Ces algorithmes fournissent une solution optimale, mais ont une complexité exponentielle dans le pire cas. Il est cependant possible d'arrêter la recherche lorsque l'on a trouvé une solution qui paraît satisfaisante. Deux méthodes sont mentionnées couramment : la théorie des graphes et la programmation mathématique.

La théorie des graphes

Ces stratégies modélisent le problème par des graphes (deux graphes, un pour la machine parallèle cible et un second pour l'application) et utilisent des techniques de traitement de graphes pour sa résolution. Pour modéliser l'architecture de la machine cible, on utilise un graphe non orienté $G_m = (V_m, E_m)$ où V_m, l'ensemble des sommets, désigne les processeurs, et E_m, l'ensemble des arêtes, représente les liens physiques. Ce graphe est non orienté car les liens entre processeurs sont supposés être bidirectionnels [5]. Les sommets peuvent être pondérés par les vitesses d'exécution des processeurs et les arêtes seront pondérées par des vitesses de communication. Le programme est représenté lui aussi par un graphe non orienté $G_p = (V_p, E_p)$ dont les sommets désignent les processus, et les arêtes représentent les communications entre les processeurs [6]. Les arcs et les sommets représentent respectivement le volume de communication entre les tâches et le volume d'exécution. Une des techniques généralement utilisée est la recherche d'un homéomorphisme faible entre les deux graphes ou plongement [Bok81].

[5] Ce type de représentation est nécessaire lorsque les noeuds de la machine ne sont pas tous connectés, et la machine est décrite par son graphe : hypercube, tore, étoile, ...

[6] Le modèle utilisé est celui des processus communicants.

L'optimisation combinatoire

La méthode modélise le problème du placement (donc un sous problème de l'ordonnancement) comme un problème d'optimisation combinatoire. Le problème se présente sous forme d'une **fonction de coût** f à minimiser, en tenant compte d'un ensemble des contraintes Q.

$$S = minf(X_1, X_2, \ldots, X_k)$$

$$Q(X_1, X_2, \ldots, X_k) \leq B_i$$

f et Q sont des fonctions polynomiales en X_i, les X_i sont des variables bivalentes; les B_i sont des constantes, et S est la solution du problème.

La recherche d'un coût d'exécution minimal traduit le but à atteindre et permet de spécifier un ensemble de critères Q qui vont diriger l'algorithme d'ordonnancement dans ses choix de distribution de la charge. Ces critères son intégrés dans la fonction de coût f. Les contraintes représentent différents aspects, par exemple la limite en taille de mémoire, la limite de charge de calcul ou les contraintes de redondance ou de terminaison (attente des données, traitement des données, envoi des résultats).

La fonction de coût peut être complexe, permettant d'exprimer les critères que l'on veut satisfaire et permettant de s'adapter à l'architecture de la machine parallèle. Cette modélisation peut avoir pour objectif d'utiliser au mieux les ressources particulières de l'architecture afin d'améliorer le débit du système et cela pour diminuer le temps de réponse moyenne.

La minimisation de la fonction de coût est généralement utilisée pour permettre dans les cas simples d'obtenir un système d'équations linéaires. Cependant dans le cas général, cette modélisation ne permet pas d'obtenir un ordonnancement assurant un temps minimal d'exécution, il faut pour cela introduire une nouvelle donnée, le graphe de dépendances entre les tâches. Ce graphe permet de mettre en évidence les relations entre les tâches. Une tâche ne peut pas s'exécuter avant d'avoir reçu les informations des tâches qui la précèdent. Les critères qui permettent de déterminer le placement ayant un temps d'exécution minimal sont : la durée d'exécution des tâches, les temps de communication et les contraintes de précédence. La recherche de ce placement sera équivalent à la résolution du problème d'ordonnancement des tâches d'exécution globale minimale. Ce qui revient à obtenir un chemin critique dans le graphe et à chercher la valeur minimale de son temps d'exécution [Bra90, Bok81].

2.2.2 Les méthodes approchées - les heuristiques

Ces algorithmes qui permettent de construire une solution approchée, se divisent en deux catégories, les **algorithmes gloutons** et les **algorithmes itératifs**. Parmi ces algorithmes se trouvent les heuristiques obtenues comme une généralisation d'algorithmes exacts qui partent d'une solution donnée et qui en relaxant certaines contraintes fournissent des solutions approchées

Les algorithmes gloutons

Les algorithmes gloutons permettent de construire une solution grossière rapidement. Le principe consiste à construire cette solution de proche en proche, en partant d'une solution initiale partielle que l'on complète; ainsi, l'allocation de la q-ième tâche à un processeur se fait sous un certain critère à partir du placement partiel réalisé sur les (q-1) premières. Il n'y a aucune remise en cause des solutions intermédiaires. Les algorithmes gloutons sont peu coûteux, mais en contrepartie, ils ne sont pas très performants. Le choix arbitraire des premières tâches à placer conditionne l'algorithme complet et vers la fin de l'algorithme, les choix de placement deviennent de plus en plus limités, ce qui peut faire tomber les performances.

Les algorithmes itératifs

Les algorithmes itératifs partent d'une solution complète que l'on améliore par transformations élémentaires. Ces algorithmes sont basés sur une fonction de coût. Dans la plupart des solutions existantes, on procède par permutations de tâches en ne retenant que celles qui améliorent la fonction de coût. Des perturbations aléatoires sont nécessaires dans la plupart des algorithmes afin d'éviter les minima locaux. Parmi ces algorithmes on retrouve les algorithmes génétiques, la recherche tabou et le recuit simulé [Bou94, Tal91].

2.3 Ordonnancement dynamique

Hypothèses du problème

- l'application sera représentée par un graphe orienté sans cycle semi-prévisible ou imprévisible ;
- les informations de coût (communication et calcul) et de localité sont imprécises ou inexistantes ;
- une faible connaissance de l'état du système : nombre de processeurs disponibles, niveau de charge ;

L'algorithme doit prendre des décisions pendant l'exécution du programme. Les contraintes de temps dans l'allocation dynamique des processus font que la plupart des algorithmes proposés ne permettent pas d'obtenir une solution optimale, seules des heuristiques peuvent être utilisées. Dans le cas du placement dynamique, les stratégies doivent disposer d'informations en provenance des différents nœuds - informations dites de **charges locales** - pour ensuite prendre des **décisions** de placement des tâches. Il est nécessaire d'avoir une connaissance dynamique et cohérente de l'état du système. Le choix d'une allocation peut être entrepris à l'occasion du lancement de la tâche ou encore à tout moment en se basant sur des possibilités de migration ou redistribution des tâches.

Lorsque l'ordonnanceur s'exécute, il doit éviter de gêner l'exécution de l'application qu'il est en train d'ordonnancer. Dans cette vision l'ordonnanceur doit être actif seulement à certains moments et avec une interaction très précise avec l'application. Les deux tâches pour l'ordonnanceur dynamique sont de contrôler l'exécution du graphe et de réguler la charge du système. Le contrôle de l'exécution du graphe qui représente l'application

n'est pas simple dans le cas dynamique. Il est impossible de prévoir structure du graphe, à fur et a mesure que le graphe s'exécute, les nouvelles tâches arrivent et doivent être ordonnancée.

Le travail de gérer le graphe est sous la responsabilité de l'exécutif et seulement quand la tâche a la possibilité de s'exécuter (cela signifie que toutes ses contraintes de précédences ont été résolues). Elle est libérée afin que l'ordonnanceur l'alloue à un processeur. En résumé, l'ordonnancement dynamique de l'application sera divisé en deux étapes la première sera sous la responsabilité de l'exécutif et la deuxième fera partie du mécanisme de régulation dynamique. Dans ce cas il est possible de modéliser ce mécanisme de régulation dynamique, comme dans le cas des systèmes d'exploitation.

Cette option de séparer le contrôle du graphe de l'application de sa répartition dynamique est utilisée chez Nexus [FST96], Cilk [BL94a] et ATHAPASCAN[Bri96, GRCD98].

2.3.1 L'utilisation des seuils

L'utilisation des seuils est très répandue dans les systèmes qui proposent la régulation dynamique. Ils indiquent un niveau de charge, établissant si un nœud est **chargé** ou non (voir figure 2.1). Ils sont utilisés au moment de prendre des décisions, comme dans le cas des algorithmes dits **actifs** : un nœud décide de façon autonome, en comparant sa charge par rapport à un seuil global, s'il est chargé ou non. S'il est **chargé**, il peut décider d'exporter des travaux (**actif à l'émission**) et chercher un nœud candidat pour recevoir ses processus. Par contre s'il détecte qu'il est **sous-chargé**, il peut décider de demander du travail (**actif à la réception**).

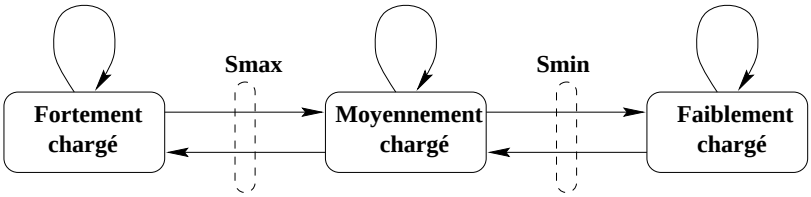


FIG. 2.1 – Transitions d'états de la charge d'un processeur

Les algorithmes actifs à l'émission ou actifs à la réception peuvent être utilisées de façon complémentaire si l'ordonnanceur possède deux seuils. Cela permet aux processeurs adapter leur rôle dépendant des changements de l'état de charge du système. La présence des deux seuils est nécessaire pour éviter les changements brusques de rôle. Chaque nœud peut alors devenir soit demandeur de travail, soit source de travail ou alors ne pas participer à l'étape de régulation (si la charge du nœud se trouve entre les deux seuils). De la figure 2.1, si la **charge** d'un nœud est inférieure au seuil minimal (*SMIN*) le processeur est à considérer faiblement chargé et fera une demande de travail. Si la charge du nœud est supérieure au seuil maximal (*SMAX*) c'est un nœud à considérer fortement chargé et il devra chercher des nœuds candidats pour recevoir son surplus de charge.

Une question à résoudre est de considérer si les seuils sont fixes ou s'ils s'adaptent au fur et à mesure que la charge du système évolue. L'implémentation de seuil variable

n'est pas facile à résoudre car elle pose plusieurs questions : quand démarrer l'activation de la modification, qui devrait faire la modification, comment diffuser la nouvelle valeur ou quelle valeur le nouveau seuil doit-il prendre. Des études ont été menées [JM94, CMKG99], mais il n'a pas été trouvé une méthode simple pour l'implantation de l'adaptation dynamique des seuils.

2.3.2 Structure générale d'un mécanisme de régulation dynamique de charge

Plusieurs travaux ont été effectués afin de proposer un mécanisme général (schéma) pour la régulation dynamique de la charge [7], que l'on appellera aussi ordonnanceur dynamique, qui répertorie les différentes actions qu'un régulateur dynamique de charge doit réaliser. En considérant que les activités sont faites en différents instants, déclenchés par différents événements, et faites par différents acteurs, Casavant et Kuhl [CK88] ont défini un cadre aussi large que possible du contexte. D'autres travaux de classification sont présentés dans les articles [Hém94],[Ber91], [WLR93] et [Jac96a]. Dans un ordonnancement dynamique, l'ensemble des tâches à ordonnancer est découvert au cours de l'exécution. L'ordonnanceur donc doit être capable de contrôler les points critiques suivants : le manque de connaissances sur l'application a priori, l'estimation de l'état de charge du système et la prise des décisions nécessaires lors de l'exécution de l'application. On part du cadre plus général en vue de spécifier les détails de quelques algorithmes d'ordonnancement dynamique classiques [Tal94, WLR93]. L'ordonnanceur sera modélisé par trois éléments :

1. un mécanisme de transfert de la charge (placement, migration, appel de procédure à distance);
2. un gestionnaire de l'état de charge du système (composant d'information);
3. un élément de contrôle et prise de décision (composant de contrôle).

Ces trois composants, mécanisme de transfert, élément d'information, élément de contrôle, ont des fonctionnalités différentes et peuvent être étudiés de façon séparée mais elles ont des interactions importantes.

Mécanisme de transfert de la charge

Nous définissons le mécanisme de transfert comme l'action physique de placer ou déplacer un objet (donnée, tâche, processus) du noeud source vers le processeur récepteur. Il ne s'agit pas de faire un choix mais d'effectuer l'action qui a été décidée par le composant de contrôle. Il peut s'agir de la **migration** d'un processus, ou du **placement** des données ou d'une tâche ou de l'**appel d'une procédure à distance**. Pour notre travail ce mécanisme sera considéré comme un aspect technique supporté par le noyau d'exécution. PM^2 [Den95] par exemple offre la possibilité de migrer une tâche en cours d'exécution, et Athapascan1 [Bri96] permet de redistribuer une tâche que n'est pas encore débutée.

[7] On considère ici le problème de réguler la charge sans considérer le contrôle du graphe des précédences.

La **migration** désigne l'opération qui consiste à interrompre l'exécution d'un processus en cours sur une machine, et à poursuivre l'exécution du processus sur une autre machine, après avoir transféré les informations nécessaires. Tout un support spécifique est nécessaire pour pouvoir arrêter le processus (**préemption**) en cours d'exécution, le transférer (**migrer**) avec tout son contexte d'exécution (mémoire locale du processus) vers un nouveau nœud cible, l'installer, relancer le processus, laisser les informations nécessaires dans le site source pour que les autres processus en relation avec la tâche migrée puissent, si est cela nécessaire, le contacter. Le mécanisme est assez complexe et la migration en elle-même est assez coûteuse en ressources de calcul, mémoire et réseaux des communications. Certains systèmes d'exploitation offrent le support pour réaliser la migration [Fol92]. Des travaux concernant la migration des processus légers existent, et des résultats très intéressant y sont reportés comme c'est le cas de la migration des processus légers dans le projet PM^2 [Den95].

Dans le cas où la migration d'une tâche ne serait pas possible se présente le problème de remettre en cause le placement originel d'une tâche puisqu'on ne peut pas arrêter un processus en route. Si la tâche n'a pas débuté son exécution, on peut toujours la déplacer vers un autre site. Dans ce dernier cas, on ne parle pas précisément de migration mais d'un déplacement de tâche ou redistribution. Comme dans le cas de l'algorithme proposé par Cilk du vol de tâches (*work-stealing*) [BL94a] où les tâches qui ne sont pas commencées se trouvent dans une liste locale à chaque processeur et quand un processeur n'a plus de travail, il vole du travail aux autres processeurs à partir de cette liste des tâches.

Gestionnaire de l'état de charge du système

Le gestionnaire de l'état de charge (composant d'information) a comme objectif de récupérer les informations nécessaires sur l'état de chaque processeur pour établir la charge globale du système. Ces informations nous permettrons de déterminer les meilleurs nœuds récepteurs (les moins chargés) ou de déterminer les nœuds les plus chargés ou potentiels émetteurs de charge. Le gestionnaire peut être absent si l'algorithme d'ordonnancement prend ses décisions de façon aléatoire, sans considérer l'état de charge du système. Deux sous-fonctionnalités forment ce gestionnaire:

- l'estimation de la charge de chaque nœud;
- l'estimation de l'état global du système.

Élément de contrôle de décision

Ce contrôleur (composant de contrôle) aura aussi diverses sous-fonctionnalités. À ce niveau, les aspects auxquels on va s'intéresser sont les principales activités que chaque module doit réaliser: politique d'activation, politique de sélection et politique de transfert [WLR93, Tal94, Fon94, JM94].

- **Stratégie d'activation:** la politique d'activation détermine à quel moment la distribution (ou redistribution) des tâches doit être effectuée, soit à partir des informations fournies par le gestionnaire du système ou activé par un chien de garde ou au moment de la création des nouvelles tâches. De l'activation peut-être responsable

un nœud spécifique(rôle du contrôleur ou des contrôleurs) ou on laisse chaque nœud autonome.

- **Stratégie de sélection:** la politique de sélection détermine les différents nœuds déséquilibrés qui joueront le rôle d'émetteur ou de récepteur. Un nœud est considéré comme déséquilibré s'il se trouve dans l'un des état suivants : il est considéré comme **chargé** ou **futur émetteur** soit comme **faiblement chargé** ou **inactif** et est alors considéré comme **récepteur**. Quand un nœud est **moyennement chargé** il ne participe pas à cette phase d'équilibrage. Différents cas peuvent se présenter, pour déterminer si un nœud est source, ou s'il est récepteur. La décision peut-être prise par le propre nœud concerné, ou par un nœud dont le rôle est d'être contrôleur (comme dans le cas centralisé), ou par un nœud complémentaire. On parle de stratégies actives à l'émission ou actives à la réception, respectivement.
- **Stratégie de détermination locale de charge:** Il faut faire le choix de la tâche (processus, tâche, données) à transférer. Dans le cas où la migration serait possible, on choisit les tâches parmi tâches en cours d'exécution à partir de certains paramètres (âge, priorité). Dans le cas d'un appel de procédure à distance la détermination se fait au moment de la création et si à cet instant on ne donne pas un site d'allocation, la décision peut être différée en conservant ces tâches soit dans une queue locale ou dans une queue globale.

2.4 Bilan

Le problème de l'ordonnancement est complexe, les solutions sont nombreuses et les champs d'utilisation sont divers. Ordonnancer une application parallèle requiert le contrôle du graphe et de la répartition des tâches sur les processeurs. Dans le cas dynamique, il est impossible d'avoir la connaissance complète du graphe. Alors il faut utiliser des techniques différentes d'une modélisation mathématique. La proposition est d'avoir un contrôle souple du graphe de précédences et un mécanisme de régulation dynamique de la charge. Dans le chapitre suivant le régulateur dynamique dont les grandes lignes ont été présentées ici sera étudié en détail.

Chapitre 3

Régulateur dynamique de charge

Dans ce chapitre sont analysées les fonctionnalités d'un mécanisme de régulation dynamique de charge pour les applications parallèles. Les deux composantes décrites sont : la composante de contrôle et la composante d'information.

Ordonnancer une application parallèle, requiert le contrôle du graphe de l'application qu'on veut placer. Un site et une date sont assignés à chaque tâche en respectant les contraintes de précédence entre les tâches. Dans le cas dynamique, où l'application à ordonnancer est inconnue a priori ce contrôle s'exerce de manière spécifique. Dans la maquette ATHAPASCAN-1a ces deux parties de l'ordonnancement sont séparées. Le contrôleur du graphe analyse l'application et libère chaque tâche à fur et à mesure que ses prédécesseurs ont été exécutés. Le régulateur reçoit les tâches prêtes pour être placées, comme dans le cas des algorithmes de liste (voir section 2.1.2).

L'interaction est très subtile entre les deux phases, puisque les deux participent à l'ordonnancement de l'application. Quand le contrôleur du graphe détecte qu'une tâche est finie, il libère les tâches suivantes. Ce régulateur peut être décrit avec la même classification que les mécanismes de répartition dynamique de charge des systèmes d'exploitation [Fol92, Jac96b, Jac96a, Tal95].

Nous appellerons par simplicité " ordonnanceur " ce mécanisme de régulation dynamique de charge qui sera décrit, mais il est clair que l'ordonnancement de l'application se déroule entre ces deux phases, le contrôle du graphe et la régulation dynamique de la charge.

• Composante d'information
• Composante de contrôle

TAB. 3.1 – Régulateur dynamique de charge

Le régulateur dynamique est subdivisé en deux composants : le composant d'information et le composant de contrôle (voir tab. 3.1). Pour chacun de ces deux éléments, différentes stratégies sont présentées. Bien que le deuxième composant semble être a priori lié au premier par une relation de cause à effet - on décide en fonction d'un état de charge estimé -, les deux composants sont étudiés indépendamment. Pour cette étude qualitative, l'approche est faite en partant de questions très simples (*Qui?, Quoi?, Vers Quoi?, Quand?, Combien?, et Comment?*

3.1 Gestionnaire de l'état de charge du système

• Estimation de la charge par nœud • Estimation de l'état de charge du système

TAB. 3.2 – Sous-fonctionnalités de la composante d'information

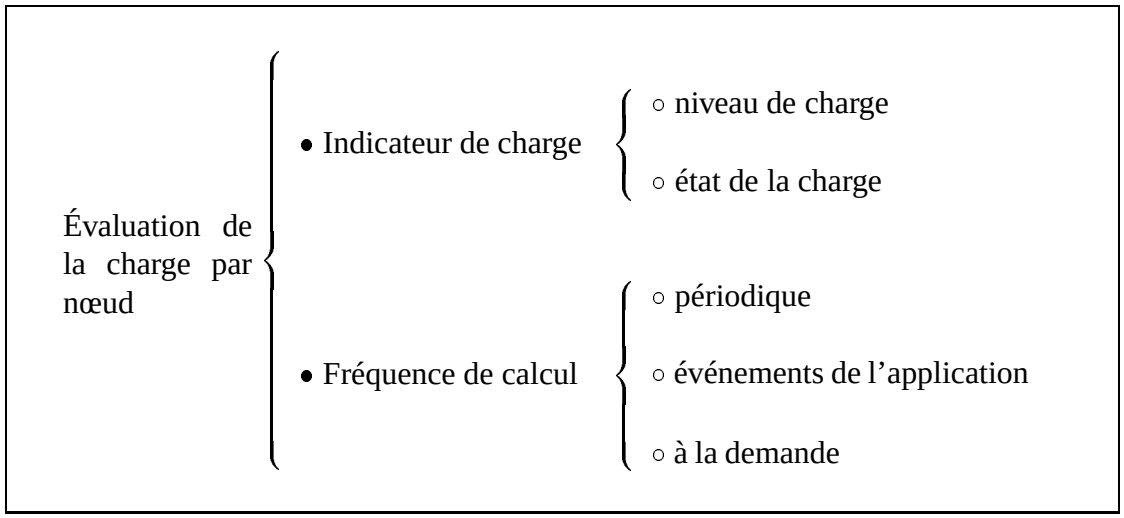
Un résultat qui date des premières études sur la régulation de la charge dans les systèmes distribués est que la probabilité qu'une machine soit inactive alors que des processus [1] sont en attente de traitement sur une autre machine est en général élevé [LM82, ML87]. Cette même observation a été faite par des travaux pour les réseaux des stations de travail [Zho88, Stu88, TL89]. Le problème du maintien de l'état courant du système nécessite d'une part l'évaluation de la charge locale de chaque processeur, et d'autre part le maintien de l'état de charge du système.

Le processus d'estimation de la charge d'un système (parallèle ou distribué), en prévision de décisions de distribution des tâches, doit être capable d'obtenir les informations pertinentes en provenance des différents nœuds de l'architecture. Pour une machine MIMD à mémoire distribuée la stratégie repose sur un exécutif permettant l'échange des messages entre les nœuds. Dans ce processus deux étapes sont différenciées : l'estimation de la charge par nœud et l'estimation de l'état de charge du système (voir tab. 3.2). La deuxième fonctionnalité est subdivisée en deux phases, la collecte des informations et l'évaluation de l'état global. L'étape de collecte est la plus cruciale, l'étape d'évaluation se réduisant le plus souvent, une fois les informations collectées, à effectuer un calcul global.

En résumé, deux coûts sont principalement induits par l'élément d'information (voir tableau 3.2) :

- le coût de l'évaluation de la charge locale de chaque processeur qui dépend de la complexité de la mesure de l'indice de charge ;
- le coût du trafic de communications induit par le protocole d'échange d'information pour le maintien de l'état global ou même partiel du système.

[1] un processus dans le sens unix



TAB. 3.3 – Évaluation de la charge par nœud

3.1.1 Évaluation de la charge d'un processeur

La plupart des algorithmes de régulation dynamique sont basés sur une évaluation de la charge sur les nœuds. L'estimation de la **charge locale** d'un processeur est un problème difficile qui ne connaît pas à l'heure actuelle de solution totalement satisfaisante [FZ87]. Le calcul peut privilégier différents aspects de la charge et avoir une complexité et une fréquence de modification variable [FZ86]. En principe chaque processeur maintient sa charge locale, avec un coût de calcul aussi réduit que possible.

D'abord, il faut établir ce qu'est la **charge** d'un nœud. C'est *Quoi* ? en vue de l'évaluer, et comment on va le mesurer (*Comment ?*). Le choix de l'**indice de charge** est important pour la régulation et ce choix peut dépendre du type d'application à réguler. Étant donné que l'évaluation de la charge d'un processeur se produit souvent (*Quand?*), il est donc nécessaire qu'elle se réalise avec une consommation minimum des ressources du système et quelle soit efficace, afin de perturber au minimum l'activité de calcul du processeur. Cette évaluation ne pourra donc pas prendre en compte tous les paramètres accessibles, même si leur importance est reconnue. Il faudra donc faire un compromis entre une évaluation détaillée qui nécessite une gestion coûteuse et une évaluation plus restreinte mais avec un coût de gestion plus faible.

Propriétés d'un indicateur de charge

Il n'est pas simple de donner une réponse générale à la question : *Quoi?*. Ferrerai [FZ87] a proposé des critères pour l'indicateur de charge que nous reprenons ici:

1. la capacité d'estimer la charge courante ;
2. l'approximation de ce que sera la charge dans un futur proche, car le temps de réponse d'un processus affecté à une machine dépend de la charge future, et non de la charge actuelle ;
3. la stabilité, afin de ne pas prendre en compte les fluctuations à court terme ;

4. la capacité à prendre en compte la spécificité d'un nœud, l'utilisation d'une ressource particulière, comme la mémoire.

L'indicateur de charge choisi doit pouvoir être facilement représentable, normalement par un nombre ou un niveau logique de charge. Un nombre s'il représente un état mesuré, par exemple 5 processus en cours d'exécution. Ou un niveau (**faiblement chargé, fortement chargé**) s'il est comparé avec un seuil. Les valeurs des seuils sont données statiquement ou peuvent changer au cours de l'exécution (voir tableau 3.3). Voici quelque exemples des indices de charge utilisés dans la littérature:

– longueur de la queue CPU [ELZ86, Tho87, EG89]
– longueur de la queue d'entrée-sortie [FZ86];
– moyenne de la longueur de la queue CPU sur un intervalle de temps: [Barak85];
– nombre de messages à traiter ;
– âge des tâches en cours d'exécution ;
– taux d'occupation de la mémoire du nœud ;
– quelques connaissance sur les tâches (coût estimé, complexité, priorité [Den95]);

Cette liste qui n'est pas exhaustive et donne des indices relativement génériques. L'utilisation d'indices complexes ou spécialisés a été aussi étudiée [FZ87]. La combinaison de plusieurs indices, pouvant être possible, les premiers résultats sont intéressants mais la plupart des chercheurs utilisent un indice simple : **la longueur de la queue CPU**. En conclusion, l'idée est d'observer seulement un nombre restreint de paramètres sur l'état du processeur, pour éviter une surcharge, même si la connaissance d'un plus grand nombre serait utile pour l'évaluation.

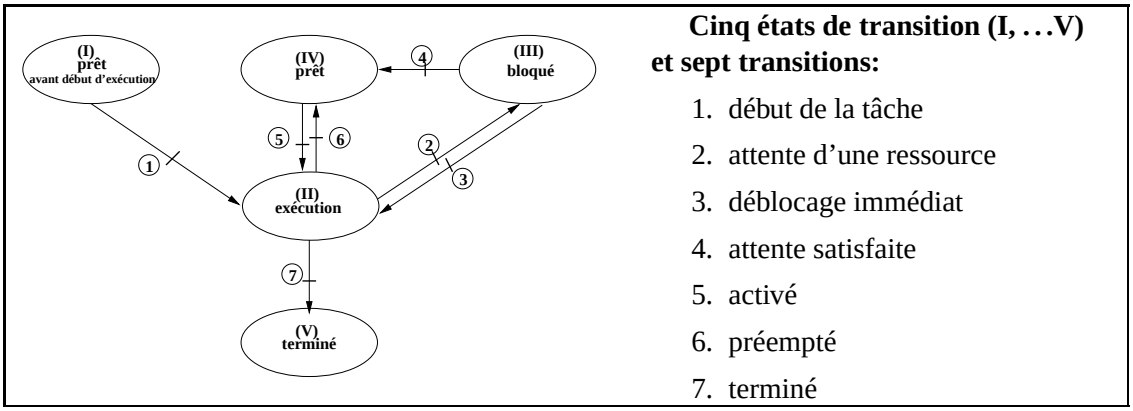


FIG. 3.1 – États et transitions d'une tâche

Dans la section 5.3.2 nous proposons une étude d'un indice complexe un peu particulier, utilisé dans l'ordonnanceur pour le système ATHAPASCAN-1a. Cet indice vise à utiliser les états de transition d'une tâche (voir figure 3.1). Pour cette étude, il y a cinq états de transitions considérés (numérotés de I à V) et sept transitions. Ces états mènent aux cinq compteurs suivants : l'indicateur de charge du nombre de tâches créées mais pas exécutés $load_I$, les tâches en exécution $load_{II}$, les tâches bloquées $load_{III}$, les tâches qui après avoir été bloquées, sont prêts à être exécutées $load_{IV}$ et les tâches qui ont finis. Lors de chaque passage par une transition, l'indice composé va croître ou décroître. Cet indice

est représenté par une combinaison linéaire ($load_I, load_{II}, load_{III}, load_{IV}, load_V$).

Méthode de calcul de la charge d'un nœud

Il y a globalement deux manières de quantifier la charge locale d'un nœud (voir tableaux 3.3). La mesure exacte et la mesure pondérée. La mesure de la charge qui est utilisée le plus souvent est une mesure système qui représente l'ensemble des tâches actives et en attente sur un nœud (c'est à dire la longueur de la queue CPU). Il a été observé que plus cette valeur est importante, plus le temps de réponse d'une tâche sera grand. Ce qui montre qu'il s'agit d'un bon indicateur qui plus est simple.

Comme il s'agit d'estimer la charge dans un futur proche, une mesure ponctuelle peut ne pas être satisfaisante puisque, entre le moment de la mesure et le moment de son utilisation pour prendre une décision, la valeur mesurée peut fortement changer par rapport à la charge réelle, à cause par exemple de nombreux débuts ou terminaisons de tâches. Pour cette raison, une valeur moyenne de la longueur de la queue CPU sur un intervalle de temps donné a été aussi proposée. Cette valeur permet d'éviter une trop grande instabilité des indicateurs de charges et donc des distributions inutiles. Lorsque l'activité d'une tâche peut être mesurée par le nombre de messages qu'elle doit traiter (cas des serveurs des systèmes distribués, ou encore celui des objets), le nombre de messages en attente de traitement sur un nœud peut être utilisé comme un indicateur de la charge dans un futur proche.

Méthode de calcul	• Exacte	◦ instantanée (longueur, ...)	
		◦ moyenne dans Δt (longueur, ...)	
	• Pondérée	◦ instantanée (poids, âge, ...)	
		◦ moyenne dans Δt (poids, âge, ..)	
	• Pondérée avec spécificité du nœud	◦ taille mémoire	
		◦ vitesse de calcul	
		◦ réseaux des communications	

TAB. 3.4 – Méthode de calcul de la charge par nœud

L'âge des tâches en cours d'exécution au moment de la mesure est pris en considération par certains systèmes. Cela se fonde sur l'hypothèse que les tâches actives depuis longtemps le resteront encore longtemps tandis que les tâches jeunes ne sont pas promises à une longue activité. La charge induite par une tâche âgée dans un futur proche sera donc plus importante que pour une tâche jeune.

Le taux d'occupation mémoire d'un nœud est aussi un élément d'information sur sa charge. Il n'est pas intéressant d'utiliser cet indicateur de façon continue, car le temps de réponse d'une tâche n'est pas dépendant du taux d'occupation de la mémoire. Cependant il peut être utilisé bien évidemment pour éviter de distribuer de nouvelles tâches à des nœuds qui n'ont pas de place pour les recevoir.

La connaissance préalable de certaines caractéristiques des tâches peut aussi intervenir dans le calcul de la charge. Il est intéressant de savoir si une tâche entre dans la catégorie de celles qui font beaucoup de calcul - et donc qui chargent beaucoup les nœuds où elles se trouvent, ou plutôt dans celles des tâches qui font beaucoup de communications - et qui donc sont souvent en attente.

3.1.2 Estimation de l'état de charge du système

Le protocole pour l'estimation de l'état de charge du système doit définir différents aspects : le moment où aura lieu la collecte d'informations (*Quand?*), de quelle manière s'opère cette collecte (*Comment?*), établir quels sont les processeurs qui participent à l'échange des informations (structure du collecteur), identifier les nœuds qui doivent donner leur information sur leur charge (*Qui?*) et ceux à qui on destine les informations (*vers Qui ?*).

• Évaluation de la charge par nœud • Protocole pour la collecte des informations • Fonction exploitation de la charge globale

TAB. 3.5 – Mécanismes pour évaluer la charge globale

Lorsqu'il s'agit de prendre une décision concernant le placement d'une nouvelle tâche, il est nécessaire de disposer d'un état de charge des nœuds potentiellement concernés. Comme le type d'architecture des machines parallèles est un système décentralisé, cet état n'est pas disponible directement et doit être estimé. Comme dans le cas de l'évaluation des indices de charge, la réponse n'est pas simple. Il est possible alors qu'une décision doit être prise, que l'information disponible ne représente pas l'état réel du système dû aux changements qui ont été opéré dans les nœuds (des exceptions peuvent se présenter, e.g. un système synchrone où tous les processus sont soit en phase de travail, soit en phase de régulation [Fon94]). Une autre remarque est que si cette information détaille avec précision l'état du système (avec différents indices de charge présentés) ou s'il a une valeur comme la queue du CPU, la recherche d'une valeur de meilleure qualité peut provoquer un surcoût.

Jacqmot [JM94] sépare deux aspects sur cette information de niveau global: d'abord savoir si l'information est prête au moment de son utilisation par le composant de contrôle, ou si elle ne l'est pas et doit être collectée. Dans le second cas il y aura un retard dans la réponse, mais cela peut assurer un meilleur placement pour la tâche. Ensuite il faut faire la différence entre une vision complète comme dans les cas des collecteurs centralisés et une vision partielle comme le cas du jeton ou d'une politique décentralisée.

La collecte de l'information est un processus qu'il ne faut pas négliger aussi bien en coût de communication (combien de messages vont être échangés avant de pouvoir faire l'estimation?) que dans le degré de confiance que l'on pourra avoir dans l'estimation (les informations collectées sont-elles encore pertinentes au moment où une décision doit être prise?). Des collectes trop fréquentes surchargent le système de communication, des collectes trop espacées donnent des estimations peu fiables.

Activation de la collecte de l'information

La question qui se pose ici est de savoir à quel moment (*Quand ?*) les nœuds émettent leurs informations de charges. Des stratégies s'opposent : celles qui privilégient l'autonomie des nœuds ceux-ci vont envoyer leur état sur la base d'une décision locale d'envoi et celles qui privilégient le rôle des collecteurs, les nœuds ne faisant que répondre à des demandes (voir tableau 3.6).

Quand?	• périodique
	• à la demande
	• changement d'état
	  ◦ par événement(s)
	  ◦ passage d'un seuil
	• message en continu
	• jamais

TAB. 3.6 – Activation de la collecte des informations

Si l'on privilégie l'**autonomie des nœuds** le déclenchement peut être provoqué ainsi : périodiquement ou déclenché par changement de la charge locale. Dans une stratégie volontaire, les nœuds envoient leur information de charge en fonction d'une décision locale. Ces envois peuvent se faire, soit à chaque modification de l'état, soit au passage par des transitions particulières - les seuils - (faiblement chargé ou fortement chargé par exemple), soit encore à intervalles de temps réguliers comme dans le cas du calcul local de la charge par déclenchement périodique. La fréquence moyenne d'émission de l'état est ici un paramètre important puisqu'il règle le nombre de messages dans le système de communication. On remarque le cas du message continu où l'on profite des communications naturelles de l'application. Un champ est ajouté à chaque message avec la charge locale du processeur mais un contrôle explicite sur l'envoi et la réception des messages est nécessaire.

Par contre si l'on privilège le **rôle des collecteurs**, les nœuds répondent seulement aux messages des collecteurs, la question est de savoir comment les contrôleurs vont initier leur demande. Ici aussi les demandes peuvent être émises périodiquement par le ou les collecteurs, ou irrégulièrement à l'occurrence d'un événement particulier - par exemple lorsqu'une nouvelle tâche doit être placée et dans ce cas c'est une demande du composant de contrôle.

Représentation de la charge globale

Le plus couramment, la charge globale est représentée par un vecteur dans lequel se trouvent les valeurs estimées de charge de chaque nœud (en fonction de l'indice de charge : valeur mesurée, seuil). Un calcul est nécessaire pour établir la charge globale du système, la moyenne, l'écart à la moyenne, la médiane et donc la charge globale est représentée par cet ensemble de valeurs ou d'état : chargé, normal, faiblement chargé.

Le protocole de collecte des informations

Différents protocoles peuvent être utilisés pour la collecte de l'information. On présente les politiques plus courantes.

Comment?	• protocole explicite	
	• protocole périodique	
	• protocole volontaire	◦ par événement(s) ◦ passage d'un seuil
	• protocole continu	
	• protocole par diffusion	◦ diffusion globale ◦ diffusion partielle

TAB. 3.7 – Protocoles de collecte des informations

• **Protocole par échanges explicites [PEE] :** dans cette politique, la collecte d'information ne se produit que sur demande explicite d'un processeur (le collecteur). Cette méthode a pour avantage de ne faire circuler sur le réseau que les informations utiles. Par contre l'inconvénient majeur est qu'il est nécessaire de faire circuler deux messages pour communiquer un seul état de charge, et que donc, cela incrémente le coût en nombre de messages. Ce protocole du type question-réponse (avec message dédié) entraîne en plus un délai non négligeable qui peut rendre obsolètes les informations obtenues car dans une

stratégie à la demande, les nœuds envoient l'information seulement lorsqu'elle leur est demandée.

• **Protocole par échanges périodiques [PEP] :** chaque nœud de calcul envoie périodiquement sa charge. Dans ce cas là, il est difficile de choisir une bonne période de temps. Une faible période permet une bonne estimation de l'état de charge du système et donc un bon placement, mais elle génère un grand nombre de messages et encombre donc fortement le système de communication. Cette méthode peut aussi générer beaucoup de messages inutiles, par exemple si la charge du processeur n'a pas varié pendant la période.

• **Protocole par échanges volontaires [PEV] :** chaque nœud de calcul envoie son état de charge en fonction d'une décision locale : le changement d'état de charge locale ou le passage d'un seuil. Cette méthode a pour avantage de ne faire transiter sur le système de communication que les messages utiles. En revanche, elle permet seulement une vision partielle du système du fait de ne voir que les franchissements de seuil. De plus, le choix des valeurs de seuil est difficile et peut dépendre de la charge globale du système. Il est important que les nœuds n'envoient pas d'informations inutiles, lorsque par exemple l'état ne change pas ou encore lorsqu'il n'y a pas de décision de placement à prendre.

• **Protocole par échanges continus *(bidding-baking)* [PEC] :** on associe un champ supplémentaire à chaque message émis contenant l'état de charge du processeur émetteur. Cette politique utilise la communication naturelle de l'application en évitant la création des messages supplémentaires. Un problème se présente quand l'application fait peu de communication.

• **Protocole par diffusion [PD] :** Le déclenchement peut être provoqué par une décision locale ou périodique, mais à la différence des autres protocoles, chaque nœuds envoi une série de messages vers plusieurs nœuds. La diffusion peut être globale (de tous vers tous) ou partielle : voisinage, sous-groupe aléatoire, jeton, le message continu, négociation-enchères, gradient.

- *Stratégies avec diffusion globale* : chaque nœud envoie son état à tous les autres nœuds. Cette solution permet à chaque nœud d'avoir une information globale de charge particulièrement à jour. Mais si les diffusions sont fréquentes, le système de communication sera vite surchargé.
- *Stratégies à diffusion restreinte* : les stratégies décentralisées avec une diffusion partielle se basent sur l'idée que chaque nœud ne va envoyer son état qu'à un nombre limité d'autres nœuds, par exemple ses voisins ou un sous-groupe aléatoire et que cela créera des vagues de messages permettant à chaque nœud de mettre à jour son état global au passage de ces vagues. Cette solution peut, comme la diffusion globale, générer un nombre important de messages. Une autre stratégie à diffusion restreinte est l'utilisation d'un jeton circulant. Un jeton contenant l'état global estimé circule entre les nœuds. Lorsqu'un nœud le reçoit, ce nœud met à jour son propre état global, modifie la valeur de son état local dans le jeton, et renvoie le

jeton à son successeur dans la chaîne de circulation du jeton. La vitesse de circulation du message et son itinéraire sont ici des paramètres importants, qui doivent être le plus souvent adaptés dynamiquement en fonction de l'état. On peut utiliser plusieurs messages circulant, par exemple à différents niveaux d'un partitionnement hiérarchique.

Des solutions intermédiaires peuvent être mises en place combinant les stratégies volontaires et à la demande. On les combine souvent de la manière suivante : les nœuds fortement chargés font des envois volontaires indiquant la trop forte charge - ce qui les sort temporairement du processus de distribution, et les nœuds faiblement chargés font des demandes ce qui les favorise dans le processus de distribution. Dans ce cas la technique des seuils est utilisée.

Structure du collecteur

Pour toutes ces stratégies, chaque nœud doit savoir vers qui envoyer son état. Plusieurs cas se présentent pour la structure du collecteur (*de Qui vers Qui?*). La question intéressante au moment de l'évaluation de la charge du système est de savoir si la vision est globale (cas du collecteur centralisé) ou partielle (cas d'un groupe des nœuds), et si le degré de distribution de la responsabilité de la collecte est partagé ou non.

de Qui vers Qui?	• Centralisée • Hiérarchique • Distribuée

TAB. 3.8 – Structure du collecteur

Puisque chaque nœud doit d'une certaine manière fournir son état local de charge au processus d'estimation global, la première question qui se pose est: vers qui chaque nœud envoie-t-il son état?

Collecte centralisée : une stratégie centralisée de collecte des charges utilise une entité unique vers laquelle toutes les informations locales de charge vont être dirigées. Cette solution a l'avantage d'un faible surcoût puisque les nœuds dédiés aux tâches de l'application ne sont pas pénalisés par ce travail. Elle est aussi relativement simple à implanter. Les inconvénients sont par contre ceux de toutes les solutions centralisées. Principalement le goulet d'étranglement sur ce nœud et donc temps de réponse qui risque de devenir important si le nombre de nœuds devient grand.

Collecte décentralisée : Dans une stratégie décentralisée de collecte des charges, tous les nœuds obtiennent toute l'information (ou une vision partielle) sur l'état des autres

nœuds, et donc chaque nœud est capable de faire une estimation de la charge globale du système. Deux écueils doivent être évités : le premier est le cas où le coût de la dissémination de l'information de charge de chaque nœud vers tous les autres nœuds devient trop pénalisant vis à vis de l'amélioration que l'on espère obtenir par ce processus ; le deuxième cas est lorsque le système ne peut plus assurer une diffusion assez rapide de l'information, ce qui induits que les nœuds n'ont plus une estimation assez précise de l'état global.

Collecte hiérarchique : Les stratégies de collecte d'information sont le plus souvent hybrides. Une manière générale, implanter une solution hybride, c'est définir différents niveaux de regroupement des nœuds (méthode hiérarchique) et utiliser des stratégies différentes dans les différents niveaux. Par exemple, les regroupements logiques des nœuds en fonction de considérations de voisinage sont pertinents. Il est envisageable de choisir une stratégie décentralisée entre les regroupements et une stratégie centralisée à l'intérieur des regroupements. Dans chaque regroupement, un nœud est responsable de la collecte des états de charge des nœuds du groupe (centralisation). Tous ces nœuds particuliers participent aussi à une estimation des charges de tous les groupes (décentralisation). Ces stratégies hiérarchiques peuvent être bien adaptées à des architectures particulières - pyramidales par exemple [Jug96]

3.2 Composante de contrôle

Etant donné un système dynamique et le manque de connaissance à priori pour l'exécution et contrôle de l'application, l'élément de contrôle doit décider du placement des tâches sur les processeurs, à partir des informations collectées. Il y a trois décisions à prendre : déterminer le processeur source (*Qui?*), choisir la tâche à transférer/placer (*Quoi?*) et choisir le processeur récepteur (*à Qui?*). Pour répondre a ses questions l'élément de contrôle est normalement séparé en deux composants [Tal94, Zho88, WLR93] : **la politique de détermination locale de charge**, qui décide si un processus (tâche) doit être ordonnancé (transféré, placé), et **la politique de sélection** qui détermine les processeurs qui vont participer à la phase de régulation, ceux qui seront candidats source et ceux qui seront candidats récepteurs. Ajoutons une dernière décision *Combien?* de charge sera transférée, puisque dans certains cas des transferts simultanés peuvent se faire. Il faut inclure comme dans les autres éléments déjà étudiés une **politique d'activation** (*Quand?*). Les décisions de base son :

- La sélection du processeur source, parmi ceux qui sont surchargés et qui devra exporter sa charge : **sélection_source** ;
- La détermination de la charge (choix des tâches) qui sera ordonnancées (placées, migrées) : **détermination_charge** ;
- La sélection du processeur récepteur de la charge (parmi ceux qui la peuvent recevoir : **sélection_récepteur** ;

• Politique d'activation	
• Politique de sélection	◦ sélection du nœud source ◦ sélection du nœud récepteur
• Politique de détermination locale de charge	◦ *Quoi ?* ◦ *Combien ?*

TAB. 3.9 – Politiques du composant de contrôle

3.2.1 Politique d'activation

Comme dans le cas de la mise à jour des informations on a diverses stratégies pour activer la répartition de la charge.

- Une activation périodique. Le problème est de trouver la bonne période.
- Une condition sur l'application en elle-même, par exemple la création des nouvelles tâches ou la terminaison d'une tâche.
- Un changement dans l'état du système. Des processeurs qui deviennent moins chargés ou des processeurs qui deviennent trop chargés. Il faut établir les seuils pour la détection.

3.2.2 Politique de sélection

Le premier but de cette politique est de déterminer les cas de déséquilibre. On pourrait la voir comme un algorithme responsable de former des sous-groupes parmi les processeurs. En fonction de leur état de charge, trois groupes seront présentés, le groupe sous-chargé, le groupe surchargé et le groupe entre les deux considéré comme dans une phase de transition qu'on pourrait nommer de charge équilibrée ou normal. On peut aussi utiliser une valeur quantitative (valeur mesurée, par exemple la longueur de la queue CPU) et on ordonne par rapport à cette valeur tous les processeurs. On pourra voir le système comme une liste ordonnée à partir d'un indice choisi. À partir de cette liste on établit que tous les processeurs au-dessous de la moyenne du système sont des processeurs sous-chargés et les processeurs au-dessus de la moyenne des processeurs surchargés.

En fonction de la politique d'activation, cette politique de sélection doit déterminer le(s) processeur(s) qui sont prêts à recevoir du travail, et les nœuds chargés qui peuvent envoyer du travail aux autres. Nommons ces deux actions, la *sélection_source* et *la sélection_récepteur*. Observons qu'après avoir établi qu'un nœud est une source, il faudra choisir son nœud complémentaire ou nœud récepteur. Par contre si un nœud a été choisi comme récepteur il faudra établir un nœud complémentaire comme nœud source. C. Fonlupt [Fon94] propose une politique d'accouplement. Il travaille avec des machines synchrones, et à chaque pas de synchronisation, des couples de nœuds source-récepteur

se forment pour équilibrer la charge, les deux processeurs ayant un accord pour réaliser l'échange des tâches.

3.2.3 Politique de détermination locale de charge

Ici il faut déterminer d'abord si une tâche est jugée transférable vers un autre nœud. On peut considérer la tâche au moment de sa création. Dans le cas des systèmes d'exploitation, il existe différents mécanismes pour choisir les processus à transférer, comme le filtrage par nom ou le filtrage par âge. Pour les applications parallèles on doit parler d'un type de filtrage plus fin. On peut envisager que le programmateur de l'application donne des caractéristiques sur les tâches (priorité, complexité, localité des données, coût estimé) qui pourront permettre faire des choix intelligents. Pour cette politique l'objectif est de déterminer le ou les tâches à transférer (détermination_processus).

3.2.4 Architecture du composant de contrôle

Lorsqu'on dispose d'une estimation de l'état du système, et que celui-ci laisse apparaître un problème de charge et/ou que de nouvelles tâches doivent être ordonnancées, le processus de décision intervient. La question qui se pose ici est de déterminer qui prend les décisions de détermination. Comment dans le cas de la structure du collecteur de l'information, soit on a besoin de la présence d'un contrôleur (un nœud spécialisé), soit les décisions sont prises par les nœuds concernés de manière décentralisée.

Architecture centralisée

Lorsque l'élément de contrôle est centralisé un seul processeur est seul responsable de l'allocation des tâches aux processeurs du système, on parle de **ferme des processeurs**. Le contrôleur sera donc le seul responsable de la distribution et aura donc un rôle extrêmement sensible. Pour cela, le plus souvent, un nœud particulier est dédié à cette tâche. Cette architecture est souvent mise en relation avec la stratégie centralisée de collecte des informations. Le même processeur estime la charge globale et prend les décisions de sélection. Cette solution ne pénalise pas les processeurs de calcul, mais a les inconvénients déjà cités d'une solution centralisée.

Architecture décentralisée

Dans les stratégies décentralisées, chaque nœud est capable d'estimer une charge globale ou partielle et peut donc prendre des décisions de localisation de tâches. Les stratégies existantes sont pour la plupart des stratégies à seuil. Lorsque la charge locale a évolué, le nœud peut être amené à prendre des décisions. Il s'agit de trouver soit un nœud acceptable, soit le meilleur nœud utilisable pour le transfert d'une ou de plusieurs tâches. Les éléments de décision sont par exemples un choix parmi les nœuds les moins chargés, ou parmi les nœuds qui ont le moins de travail en attente, ou encore parmi les nœuds ayant répondus le plus favorablement à des demandes, etc.. Lorsque plusieurs nœuds sont

acceptables, un choix aléatoire sera utilisé. D'autres stratégies décentralisées sont celles qui donnent l'autonomie de décision aux nœuds, comme le cas des décisions en aveugle et des stratégies coopératives ou de négociation.

Décision à l'initiative de l'envoyeur d'une tâche : Dans les stratégies basées sur l'initiative de l'envoyeur, un nœud qui a des tâches à placer cherche des nœuds moins chargés que lui pour les y placer. Les stratégies à initiative de l'envoyeur sont le plus souvent utilisées car elle ne nécessitent pas obligatoirement un mécanisme de migration de tâches. Si la charge globale est grande, ces stratégies sont peu performantes car elles perdent du temps à chercher sans succès des nœuds faiblement chargés.

Décision à l'initiative du receveur de la tâche : Dans les stratégies basées sur l'initiative du receveur, les nœuds faiblement chargés cherchent les nœuds fortement chargés pour leur demander des tâches. Les stratégies à l'initiative du receveur se basent quant à elles sur la possibilité de pouvoir prendre une tâche à un autre nœud. Ces stratégies sont difficiles à régler car lorsque la charge globale devient petite, les nœuds faiblement chargés deviennent nombreux et inondent le système de demandes de travail.

La présence simultanée des deux stratégies est utilisée. De telles stratégies symétriques permettent de tirer avantage des deux stratégies complémentaires précédentes en s'adaptant à la charge. Elles entraînent globalement une distribution plus rapide et équitable des tâches en toute situation. Elles peuvent néanmoins donner des systèmes instables dans lesquels les tâches font des allers-retours inutiles entre nœuds [LK87]. Une solution sera l'utilisation de seuils qui sont adaptés dynamiquement pour éviter cette instabilité.

Décisions en aveugle : Les stratégies de distribution des tâches les plus faciles à mettre en œuvre sont celles qui n'utilisent pas d'état global du système. Le choix d'un nœud pour une tâche se fait sans collecte d'informations. Dans une version centralisée, un processus unique place les tâches de manière aléatoire ou de manière cyclique sur les différents nœuds de l'architecture. Dans une version décentralisée, chaque nœud, à l'occasion d'une création d'une nouvelle tâche ou lorsqu'il devient trop chargé, place les tâches concernées là aussi de manière aléatoire ou cyclique sur d'autres nœuds. Une variante améliorée permet aux nœuds receveurs de refuser la tâche qui sera alors soit exécutée localement chez l'envoyeur (refus de transfert) soit envoyée sur un autre nœud. Une mémorisation des refus permet aux nœuds de faire une sorte d'apprentissage et donc d'éviter par la suite le transfert vers les nœuds ayant précédemment refusés.

Décisions coopératives : On trouve dans la littérature des stratégies qui mettent en place une coopération particulière entre les nœuds. Les exemples les plus répandus sont:

- La méthode des paires (nommé politique d'appariement). On détermine des couplages, ici par exemple un nœud fortement chargé va chercher un nœud faiblement chargé. Puis les deux vont former une paire à l'intérieur de laquelle les distributions ou redistributions de tâches seront opérées. Et cela jusqu'à l'équilibrage de

la charge pour la paire de nœuds. La méthode cherche donc à créer des canaux privilégiés entre les nœuds pour la distribution des tâches.

- La méthode des enchères. Ici un nœud charge diffuse une demande d'accueil vers d'autres nœuds. Chacun des nœuds touchés répond en indiquant un coût estimé pour la tâche si elle devait être exécutée par lui. Le nœud initial collecte les réponses, et choisit le meilleur nœud destinataire (ou il prend en charge la tâche lui-même). Comme on le voit, les nœuds faiblement chargés surenchérissent en indiquant des coûts bas.

3.3 Combinaisons et relation entre les éléments du mécanisme de régulation

Nous avons décrit un à un, les différentes fonctionnalités du mécanisme général pour la régulation de charge. Maintenant on va étudier les interactions entre les éléments qui le forment. D'abord les interactions entre la composante d'information et la composante de contrôle. Puis les combinaisons des politiques de sélection et de détermination de la charge.

On avait noté qu'il existe une dépendance de cause à effet - on décide en fonction d'un état estimé. Si on oublie cette relation l'espace des stratégies possibles est le produit cartésien de l'ensemble des stratégies d'estimation par l'espace des stratégies de décision, ce qui donne déjà un grand nombre de possibilités. Rappelons qu'il existe des stratégies aveugles qui n'utilisent pas de composantes d'information.

Nous avons trouvé très descriptif les jeux des combinaisons que Jacqmot [JM94] établit à partir des trois décisions à prendre de l'élément du contrôle : la sélection du nœud source (sélection_source), la sélection du nœud récepteur (sélection_récepteur) et la détermination de la tâche qui doit être transférée (détermination_charge). L'ordre dans lequel se prennent les décisions est important et on remarque que de tous les jeux de combinaisons possibles seulement cinq sont réalisables (voir figure 3.2).

Dans les trois premières combinaisons, après avoir choisi un nœud comme source ou comme récepteur, il est nécessaire de trouver un partenaire. Cependant certaines stratégies n'ont pas besoin d'établir l'accord entre les nœuds sources et les nœuds récepteurs.

Analysons les différents cas de la figure 3.2.

1. Si on donne la priorité au choix du processeur source (combinaison 1 et 2), on se pose d'abord la question de savoir si la migration est possible ou si la remise en cause d'un premier placement (les tâches déjà placées mais pas encore débutées) est admise. Deux possibilités sont envisageables :
   - Combinaison 1 : on forme des couples de processeurs qui s'échangeront la charge, en principe on doit chercher un processeur récepteur, et puis si on le trouve, on décide quelle (ou quelles) tâches seront transférées.
   - Combinaison 2 : on choisie les tâches à transférer et à partir des caractéristiques des tâches on cherche le meilleur nœud récepteur.
2. Si on donne la priorité au choix du processeur récepteur. Deux options se présentent

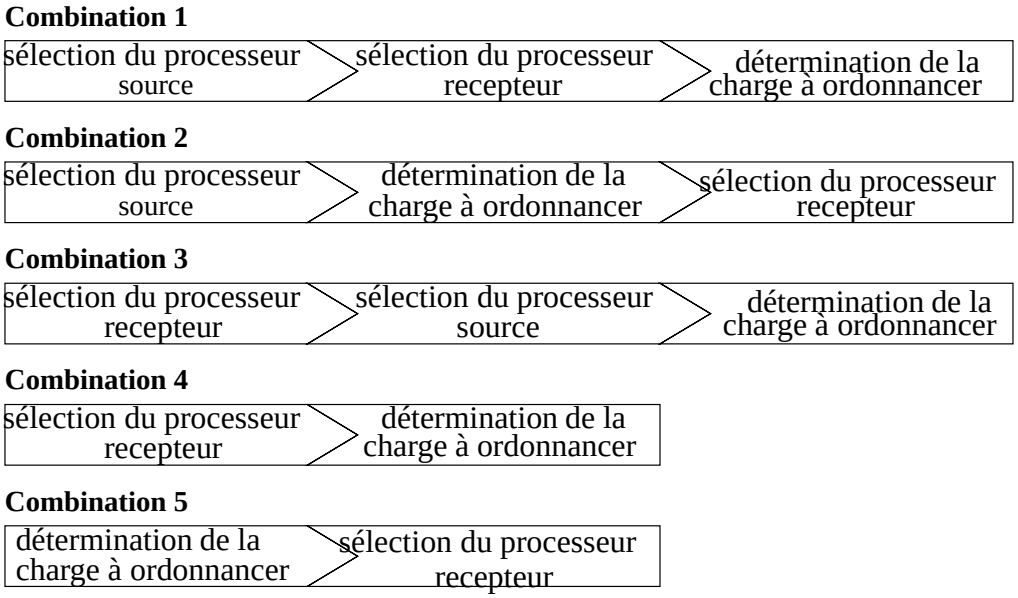


FIG. 3.2 – Combinaisons des décisions à faire par le composant de contrôle

(combinaison 3 et 4) :

- Combinaison 3 : on va chercher un nœud source qui devra faire couple avec le récepteur, et puis on choisira les tâches.
- Combinaison 4 : c'est un algorithme spécial puisque le choix de la source ne se pose pas, soit parce qu'on la connaît déjà, soit parce que les tâches à distribuer se trouvent dans une liste d'attente centralisée par exemple.

3. Si on donne la priorité au choix de la tâche à être ordonnancée (combinaison 5). Dans ce cas le choix de la source ne se pose pas puisqu'on le connaît déjà. Après avoir choisit les tâches à transférer, on doit chercher le meilleur récepteur ou un bon récepteur.

3.4 Conclusion

Lors de la description de la structure du régulateur dynamique de charge, on a constaté que le nombre de variantes est immense et chaque jour de nouvelles propositions apparaissent. Il n'existe pas un " algorithme " capable d'ordonnancer toutes les applications. Pour une application donnée, il peut-être nécessaire de définir un type spécifique de stratégie d'ordonnancement. Un noyau exécutif destiné aux applications parallèles doit proposer un mécanisme de régulation, et ce mécanisme doit être flexible par exemple sous la forme d'une boite d'outils qui permettra l'adaptation du régulateur à une situation donnée.

Deuxième partie

La régulation dynamique de charge au sein d'ATHAPASCAN

Chapitre 4

Environnements de programmation parallèle, le cas ATHAPASCAN

Dans ce chapitre, nous présentons les environnements pour le calcul parallèle qui ont comme base la multiprogrammation légère et les communications. L'environnement ATHAPASCAN[1] *qui est développé au sein du projet APACHE (Algorithmique Parallèle et pArtage de CHargE) [Pa94] sera présenté. Il permet la construction d'applications parallèles irrégulières et gère dynamiquement la régulation de charge.*

La conception de programmes parallèles est délicate et l'écriture d'un programme parallèle est souvent faite à partir d'un algorithme séquentiel. Après la conception de l'algorithme, vient sa mise en oeuvre sur une machine parallèle, et cette phase est souvent longue. Pour aider les utilisateurs, les environnements d'exécution parallèle doivent avoir certaines caractéristiques. Tout d'abord, ils doivent être faciles d'emploi ; l'environnement doit fournir des outils annexes permettant la gestion de trace et de déboggage. Le déverminage est plus difficile que pour les programmes séquentiels à cause de la distribution sur plusieurs processeurs. Quand la mise au point du programme est terminée, il reste à analyser ses performances. Pour cet aspect aussi, l'environnement parallèle doit aider l'utilisateur dans son travail.

L'objectif de la première section est de décrire les environnements pour la programmation parallèle qui permettent un grain très fin de parallélisme et assurent des communications performantes. Ce sont les systèmes qui supportent la multiprogrammation légère, et nous commencerons par la différenciation entre processus lourds et processus légers et par un survol des bibliothèques de communication (avec une description plus détaillée de PVM et MPI). Ensuite nous verrons une description de la multiprogrammation légère et finalement un survol des systèmes qui ont été développés pour offrir la multiprogrammation légère avec des communications performantes.

Dans la deuxième section des environnements qui proposent des solutions (ou implémentations) au problème de l'ordonnancement des applications parallèles sont présentés. Dans la dernière partie l'environnement ATHAPASCAN est présenté. En particulier plus

[1] L'ATHAPASCAN est la langue des Apaches

spécialement nous décrivons son *Interface Applicative* : la couche ATHAPASCAN1.

4.1 Les applications parallèles et leurs supports d'exécution

La recherche de bonnes performances se fait à travers la manière dont l'application parallèle est construite mais aussi par la manière dont elle est exécutée. Le placement des tâches et leur gestion par l'environnement parallèle va influer directement sur les performances du calcul et des communications. Par exemple l'échange de données entre les processus d'un programme parallèle est souvent le point le plus important pour les performances sur un réseau de stations de travail qui est utilisé comme machine parallèle. Le noyau de communication utilisé dans l'environnement doit être particulièrement performant.

4.1.1 Les processus lourds vs. les processus légers

Nous illustrerons la gestion des processus dans les systèmes par le cas du système **UNIX**. Les processus ont toujours tenu une place importante dans les systèmes d'exploitation.

Un **processus lourd** est l'entité d'exécution de base offerte par le système d'exploitation qui garantit une isolation et une protection par rapport aux autres processus lourds du système. L'opération de création de processus UNIX est gourmande en ressources systèmes. Lors de sa création le système alloue des ressources mémoire (segments de code, données, pile) mais également un contexte d'exécution (compteur ordinal, registres). La commutation entre processus est très " lourd " du à toute la mémoire allouée. Le nombre de processus supportés par le système est en général limité. La gestion de ces processus est entièrement réalisée par le système d'exploitation. Son rôle principal est de gérer le partage des ressources. La plupart des systèmes d'exploitation proposent une exécution des processus réalisée par le biais d'un ordonnancement basé sur la préemption, en temps partagé [Bal91].

Un **processus léger**, encore appelé *light weight process* ou *thread* correspond à un fil d'exécution qui s'exécute au sein d'un processus lourd éventuellement en concurrence avec d'autres fils d'exécution. Plusieurs processus légers vont s'exécuter au sein d'un même processus lourd et vont se partager les ressources système du processus donc en particulièr la mémoire par exemple les données ou variables globales, seule la pile est prive. La création d'un processus léger est une opération peu coûteuse et la commutation, entre processus légers est plus rapide (de l'ordre de 100 fois) que la commutation entre processus lourds. La politique d'ordonnancement des processus légers peut être réalisée de façon interne au processus lourd et indépendamment de la gestion de celui-ci par le système d'exploitation [McB94].

4.1.2 Les bibliothèques de communication

Les besoins en communication pour les applications parallèles sont très variés et exigeants. Il existe de nombreuses solutions offertes aux utilisateurs. Il y a principalement deux façons de réaliser la communication : de façon implicite par mémoire partagée virtuelle ou de façon explicite par passage de messages. Des bibliothèques de communication pour le calcul parallèle ont été développées [McB94]. Elles ont d'abord été spécifiques à une architecture donnée, souvent développées par les constructeurs de la machine, puis, avec un plus grand effort de portabilité, elles sont devenue des standards. Des outils comme **Express** [FK94], **NX** [Pie88], **Landa**[Mon96], **MPI** [Mes95], **Parforms**[Mon96], **Parmacs**[CRHW94], **PVM** [Sun90], et **Zipcode** [SSD+94] ont choisi des communications par passage de messages. D'autres ont choisi un modèle de programmation par mémoire partagée : **Linda**[CGMS94], **P4**[BL94b].

Ces bibliothèques, outre des fonctionnalités mieux adaptées au calcul parallèle, présentent pour certaines la possibilité d'être implantées de façon efficace pour exploiter la latence et le débit des réseaux utilisés. Les systèmes offrent principalement les primitives classiques de communications point-à-point et des opérations collectives comme la distribution de données.

La **communication point-à-point** comprend principalement deux primitives :

- envoyer(destination, ..., message)
- recevoir(émetteur, ..., message)

Il existe essentiellement deux protocoles pour transmettre un message de sa source vers sa destination : les communications **synchrones** et les communications **asynchrones**. Les communications sont synchrones si l'émetteur attend que le processus récepteur soit prêt à recevoir les données pour les envoyer, ensuite il attend la fin de la réception des données. Les communications sont asynchrones si l'émetteur envoie directement les données, sans vérifier la réception. C'est claire c'est plus rapide mais la communication n'est pas fiable.

Les primitives de la **communication collective** permettent d'exprimer facilement et directement des communications qui impliquent l'ensemble des processus ou un groupe des processus de l'application. Parmi les communications collectives les plus courantes, on trouve :

- **la diffusion** (*broadcast*) qui est l'envoi d'une donnée d'un processus vers tous les autres. Si la donnée envoyée dépend du processus destinataire, on parle de distribution ;
- **le regroupement** (*gathering*) qui est la réception de données provenant de tous les processus sur un seul processus.
- **l'échange total** (*all to all*) qui est la diffusion simultanée de données à partir de chacun des processus. Chacun des processus communique avec tous les autres.

Dans le cas de problèmes réguliers, la parallélisation conduit souvent à algorithmes globalement synchrones qui sont des enchaînements de phases de calcul équilibrées et de phases de communications pour la redistribution de données [Cal95]. Le modèle d'exécution BSP [McC95] exploite ce type de contrôle d'exécution. Les technologies existantes telles les bibliothèques de communication PVM et MPI, et les directives de partitionne-

ment de données comme HPF [PD96] permettent l'exploitation efficace de ce type de parallélisme. PVM et MPI sont décrits dans les sections suivantes, ces bibliothèques étant parmi les plus utilisées.

PVM

PVM (*Parallel Virtual Machine*) [Sun90], est un environnement de programmation développé à l'*Oak Ridge National Laboratory*. PVM introduit le concept de **machine virtuelle** qui a révolutionné le calcul distribué hétérogène en permettant de relier des machines différentes pour créer une machine intégrée parallèle. Son objectif est de faciliter le développement d'applications parallèles constituées de composants relativement indépendants.

PVM offre les communications point-à-point étiquetées entre tâches et des communications et opérations globales. Les communications se font exclusivement par l'intermédiaire de tampons, dans lesquels sont emballées les données. Pour le support des communications et opérations collectives, PVM introduit la notion de **groupe**. Les groupes sont **dynamiques** et il est possible à une tâche de s'insérer dans un groupe et d'en sortir. Entre membres d'un groupe, il est possible de faire des diffusions, des barrières (une barrière est un mécanisme de synchronisation, un processus qui arrive à la barrière sera bloqué jusqu'à l'arrivé de tous les processus concernés par cette même barrière) ainsi que des opérations collectives de réduction par des fonctions fournies par l'utilisateur.

Avec un environnement comme PVM les mécanismes de régulation reposent généralement sur le placement des tâches à l'initialisation, suivi d'une distribution des données pendant l'exécution. Le mécanisme de placement est assez restrictif avec comme principal inconvénient que le coût de la création est assez important puisque les tâches PVM sont des processus lourds. La répartition des données n'est pas prise en charge par PVM et c'est donc au programmeur de répartir les données convenablement sur ses différentes tâches. On ne peut considérer PVM comme une simple bibliothèque de communication. C'est plutôt un support complet pour un réseau dynamique (groupe dynamique) de processus lourds communicants.

MPI

MPI, (*Message Passing Interface*) [Mes95], est un standard définissant la syntaxe et la sémantique d'un noyau de routines destinées à l'écriture de programmes parallèles portables. Dans MPI ont été intégrées les fonctionnalités les plus intéressantes de plusieurs systèmes existants (dont PVM). Un des objectifs de cette standardisation est de permettre des communications très efficaces sur les principales plates-formes parallèles.

Le standard définit les communications point-à-point, les communications et opérations collectives, les groupes de tâches, les communicateurs, les topologies de processus, la syntaxe des appels en C et en Fortran 77, l'interface de prise de trace et instrumentation et les fonctions relatives à l'environnement d'exécution. Ce standard a été implanté de façon très efficace sur diverses machines dont entre autres l'IBM-SP et le CRAY-T3E.

Par contre, d'importants aspects de la programmation parallèle ne sont pas abordés,

à savoir les opérations relatives à la mémoire partagée, la délivrance de messages par interruptions, l'exécution à distance et les messages actifs, la multiprogrammation légère, la gestion des machines et des processus et les opérations d'entrée-sortie.

4.1.3 La multiprogrammation légère

Comme nous l'avons déjà dit, la multiprogrammation légère (*multithreading*) consiste à exécuter plusieurs processus légers au sein d'un processus lourd. Les processus légers ont un surcoût faible comparé à celui des processus lourds. Aujourd'hui les noyaux de processus légers sont capables de gérer plusieurs centaines de flots d'exécution au sein d'un même processus lourd. Ces noyaux se caractérisent par leurs politiques d'auto-ordonnancement, c'est à dire par la façon dont est gérée l'exécution concurrente des processus légers. Il existe essentiellement deux politiques d'auto-ordonnancement des processus légers : la politique FIFO (*first in, first out*) et la politique de temps partagé. En plus de faciliter l'utilisation entrelacée des ressources, les noyaux de processus légers permettent une programmation efficace grâce à un partage de l'espace d'adressage. Il existe une norme pour l'interface de programmation de processus légers. Il s'agit du standard POSIX [Mue93].

Les difficultés introduites par la multiprogrammation concernent notamment la synchronisation des différents fils d'exécution et la protection des accès aux données partagées. Il faut s'assurer que l'ordre d'exécution des différentes parties du programme est compatible avec l'algorithme et que les données ne vont pas être corrompues par des accès concurrents non contrôlés des différents fils d'exécution.

4.1.4 Multiprogrammation légère et communication

Une bibliothèque de communication peut-être complétée par de la multiprogrammation légère pour proposer un moyen efficace de communication et de création dynamique de tâches. L'intégration des communications et de la multiprogrammation légère présente de nombreux avantages, liés tant à la recherche de l'efficacité dans l'exécution qu'à une plus grande souplesse et facilité de programmation. Ils permettent l'exécution efficace de tâches à grain fin au sein d'une application parallèle ou distribuée. La multiprogrammation légère permet de recouvrir les attentes de communication d'un fil par des calculs d'un autre fil, masquant de ce fait les durées de communication [Gin97, Chr96, Riv97]. En effet, si la découpe du calcul entre fils est " bien faite " dès que l'un entre eux est en attente de donnée, un autre peut prendre le processeur pour effectuer une autre partie du calcul.

Un certain nombre d'environnements de processus légers distribués existent. L'ensemble de ces environnements ont des caractéristiques différentes et des performances qui dépendent du type d'applications. Le tableau 4.1 décrit les caractéristiques des principaux environnements existants. On les résume par rapport à la bibliothèque de communication utilisée (PVM, MPI, ...), les noyaux des processus légers supportés (POSIX, Marcel, REX) et le modèle de programmation proposé [Chr96, Riv97, Gin97].

Environnement	Bib. Com.	Threads	Modèle de programmation
Chant	MPI, P4, NX, Pthreads	POSIX	PLC et RSR
DTMS	PVM	SOLARIS	PLC explicite
NEXUS	type MPI	type POSIX, divers	RSR
MPI-F	MPI	POSIX	PLC et RSR
PM2	PVM	Marcel	LRPC
TPVM	PVM	REX, Solaris, C- Threads	PLC
ATHAPASCAN-0a	PVM	divers	LRPC
ATHAPASCAN-0b	MPI	POSIX, Marcel	PLC

TAB. 4.1 – Environnement basé sur la multiprogrammation légère avec communications

Pour le modèle de programmation, ils présentent soit un modèle de **processus légers communicants (PLC)** soit un modèle d'**appel de service à distance** (**RSR** pour *Remote Service Request*) ou un modèle d'**appel de procédure distante légère** (**LRPC** pour *Lightweight Remote Procedure Call*). Dans un LRPC, l'appelant attend un message de retour en provenance de la procédure appelée. L'appel de service à distance est une opération asynchrone pour que le fil qui l'initie, ne se bloque pas comme dans un appel de procédure distant car il n'y a pas de retour. Nous continuos par un survol de ces environnements.

Chant [HB95], le modèle utilise des identificateurs globaux de processus légers, des échanges de messages, et des appels de services distants. Chant est le support d'exécution du langage OPUS (data parallèle) et du concept des moniteurs distribués (*ShareD Abstractions*).

DTMS [Col95](*Distributed Task Management*), le modèle de programmation utilise des processus légers exécutant des fonctions connues globalement (*fonctions comportamentales*), et l'échange de messages explicite entre processus légers. Le système comprend un mécanisme global d'équilibrage de charge utilisant une distribution statique des fonctions et une distribution dynamique des processus légers. Le système est opérationnel sur SUN/Solaris et DEC/OSF.

Nexus [FST96], le modèle de programmation utilise l'appel de service distant (RSR) en désignant un destinataire par un pointeur global (GP pour *Global Pointer*) et un nom de fonction. La fonction s'exécutera de manière asynchrone dans le contexte pointé par le GP. Il existe deux sortes de RSR : soit un nouveau processus léger est créé pour exécuter la fonction, soit la fonction est exécutée par un processus léger préexistant. Nexus est utilisé comme support pour Composite C++ [Fos95] et Fortran-M [Fos95].

MPI-F [FWR+95], est une version efficace et multi-threadée de MPI pour les machines SPx d'IBM. Sont disponibles l'échange de messages entre processus légers, l'accès à mé-

moire distante et l'invocation de service distant sous trois formes : rapide, " non-threaded " et " threaded ".

PM^2 (*Parallel Multithreaded Machine*) [Den95], le modèle de programmation de PM^2 repose sur l'appel de procédure à distance léger. Un service, préalablement déclaré, est exécuté sur une tâche distante dans un fil d'exécution créé pour l'occasion ou dans un fil système. Trois variantes du RPC existent : l'appel synchrone classique, l'appel asynchrone et l'appel synchrone avec attente différée. Le but de PM^2 est de fournir un support d'exécution avec des fonctionnalités permettant l'écriture de régulateurs de charge utilisant les priorités pour l'ordonnancement des processus légers. Cet environnement supporte la migration des processus légers.

TPVM (Thread-PVM) [FS95], l'interface de programmation est une extension de celle de PVM. Le modèle de programmation est soit celui des processus concurrents (les processus légers sont créés explicitement), soit celui du *dataflow* gros-grain (les processus légers sont créés en fonction d'événements postés). Un nommage symbolique global des processus légers est utilisée, il précise la localisation. La communication entre processus légers utilise l'envoi de message et l'accès à la mémoire distante. Des portages existent sur SunOs, Solaris et Aix.

ATHAPASCAN-**0a** La librairie ATHAPASCAN-0a [Chr96] est la première maquette réalisée par le projet APACHE pour la parallélisation d'applications. Le modèle de programmation est l'appel léger de procédure à distance. La synchronisation entre plusieurs procédures est rendue possible par l'introduction des procédures barrières [Bri95].

ATHAPASCAN-**0b** [Gin97] est le support d'exécution actuel du projet APACHE, dont le but est l'obtention d'une plate-forme portable pour applications irrégulières. Elle utilise la communication par envoi de messages entre processus légers, la communication par accès à la mémoire distante (RMA) ou le partage de mémoire locale entre les processus légers locaux.

4.1.5 Environnements parallèles et ordonnancement

De ce parcours réalisé parmi les environnements qui offrent la multiprogrammation légère on observe de façon générale que :

- Le modèle de programmation proposé est simple et bien connu : processus légers communicants, appel de procédure distant et appel de service distant.
- La tendance générale est d'offrir un système qui puisse être portable sur des supports hétérogènes (pour l'utilisation des réseaux de stations de travail comme machines parallèles principalement).
- L'objectif est de donner des moyens pour mieux exploiter les ressources, avec la possibilité de recouvrir les communications par du calcul entre autres.

Dans la première partie de ce document nous avons constaté qu'il n'existe pas de solution générale pour le problème d'ordonnancement. Il est difficile de trouver dans ces systèmes des mécanismes complets de régulation dynamique de charge. La plupart des systèmes laissent ce travail au programmeur de l'application. Cependant il est vrai qu'ils offrent des fonctions pour réaliser l'implantation des algorithmes de répartition de charge, comme la migration des processus légers ou le contrôle de l'exécution des processus légers par priorités dans PM^2.

L'introduction d'un algorithme de régulation de charge "unique" ne peut pas convenir à tout type d'application, et c'est habituellement le programmeur qui a "une idée" du type de mécanisme qu'il devra utiliser pour améliorer l'utilisation des ressources.

Nous présentons dans la suite un survol sur les environnements de programmation parallèle qui " proposent ", eux, des mécanismes pour l'ordonnancement ou placement de la charge. Leur nature et buts sont en général différents.

Cilk

Cilk [BL94a] est une extension du langage de programmation C pour les applications parallèles avec des mécanismes de contrôle et de synchronisation pour les processus légers. C'est un langage de multiprogrammation légère pour des machines à mémoire partagée. Un programme en *Cilk* est un ensemble de procédures, chacune étant constituée d'une séquence de processus légers. *Cilk* a vécu des évolutions. Dans la première version *Cilk-1*, un *thread-cilk* est une fonction non-bloquante. La communication entre un processus léger et son créateur (par exemple une valeur de retour) nécessite l'expression explicite de la continuation de l'exécution du créateur par le lancement d'un processus léger. Le passage des arguments ou le retour des résultats introduisent des précédences entre les différents processus légers. Le graphe de tâches (un graphe data-flow) est construit et ordonnancé en cours d'exécution par un algorithme de vols de tâches (*work-stealing*).

Compositional C++

Compositional C++ (CC++), offre une série d'extension au langage C++ pour le calcul parallèle. Il utilise comme noyau de multiprogrammation légère *Nexus*. Il est orienté spécifiquement pour l'exécution concurrente, le contrôle de la localité et des communications. Les nouvelles classes proposées sont : *processor object, global pointer, thread, sync variable, atomic function* et *transfer function*. Par défaut un processus léger est exécuté dans le même " processeur objet " où se trouve son père. Si l'opération doit être exécutée dans un autre processeur, cela se fait par un appel léger de procédure à distance. Le placement des processus légers est fait en deux phases, d'abord un processus léger est placé dans un processeur objet et puis le processeur objet sera placé dans un processeur physique. Chaque processeur objet peut avoir un ou plusieurs processus légers et chaque processeur physique peut avoir un ou plusieurs processeurs objets.

HPF

Le langage HPF (High Performance Fortran) [PD96] se base sur le langage Fortran augmenté des instructions pour la description du parallélisme de données (boucle parallèle) et des directives de compilation qui précisent une distribution des données. Le compilateur est alors chargé de générer le code pour l'ensemble des processeurs, y compris la gestion des communications. Les directives de placement pour les données que reçoit le compilateur sont des lignes directrices, ils ne sont pas des instructions. La régulation de charge consiste essentiellement en le choix d'une bonne distribution des données permettant d'équilibrer les calculs sur les différents processeurs. Cette distribution peut être éventuellement remise en question en cours d'exécution. Le langage HPF est essentiellement destiné aux applications qui manipulent des structures de données régulières. Des extensions sont proposées en vue d'intégrer un parallélisme à base de tâches ainsi que des outils pour la manipulation de structures de données irrégulières (par exemple des matrices creuses).

Pyrros

Pyrros[YG92] est un outil d'ordonnancement statique de graphe de tâches orienté et sans cycle. Les graphes sont annotés par leurs coûts de calcul et de communication. Pyrros calcule l'ordonnancement des tâches et génère le code pour des architectures de type MIMD (processeurs identiques) en langage C et Fortran. La prise en compte du recouvrement des communications par des calculs est possible. Pour calculer l'ordonnancement d'un graphe, Pyrros utilise une approche multi-étapes : une première phase de regroupement des tâches (*clustering*) en utilisant un algorithme DSC (*dominant sequence algorithm*). Dans une deuxième phase si le nombre de groupes est plus grand que le nombre de processeur, les groupes de tâches sont placés sur p processeurs virtuels en les regroupant par les contraintes de communication entre les groupes. Finalement les p processeurs virtuels sont placés dans les processeurs physiques. L'expression du parallélisme est considérée comme explicite : l'utilisateur doit fournir le graphe de tâches. La taille du graphe de tâches pouvant être très important, certaines études tentent d'utiliser une paramétrisation comme c'est le cas de PlusPyr [CL95]. Les applications visées par Pyrros sont les applications dont on connaît le graphe de tâches avant exécution soit des graphes prévisibles.

Dynamo

Dynamo [Tar92] plus qu'un outil pour l'ordonnancement, est un environnement qui offre une boite à outil pour la construction des stratégies de régulation dynamique de charge pour les applications parallèles. L'idée de base, est que l'application est séparée de l'opération de régulation. Dynamo est composé de deux parties, l'interface applicative (DAI) et l'interface de régulation de la charge (DBI : *balancer interface*). Dynamo a besoin de la définition d'une *tâche Dynamo*, laquelle est décrite par ses attributs et un pointeur aux données qui lui sont propres. Les attributs qui la décrivent sont utilisés par l'interface de régulation pour ordonnancer la tâche.

ATHAPASCAN**1**

La solution que propose le projet APACHE est l'interface applicative ATHAPASCAN1. L'idée de base est de séparer la programmation de l'application de son ordonnancement. Dans la section suivante nous décrirons la première maquette de l'interface applicative nommée ATHAPASCAN-1a [2] au sein du projet. Cette première interface est basée sur le modèle d'appel de procédure à distance. Une nouvelle interface a été développée et est nommée ATHAPASCAN-1b celle-ci est basée sur un modèle data-flow [GRCD98]. L'application est représentée par un graphe dynamique de précédences des tâches (dans ATHAPASCAN-1a) ou par un graphe *data-flow* (dans ATHAPASCAN-1b) qui sera exécuté à l'aide d'un ordonnanceur. Le support pour la régulation dynamique permet d'adapter ou décrire de nouveaux algorithmes d'équilibrage de la charge. Une bibliothèque d'algorithmes de régulation dynamique est proposée. L'ordonnanceur d'ATHAPASCAN sera décrit dans le chapitre suivant.

4.2 ATHAPASCAN

L'environnement de programmation parallèle ATHAPASCAN est un support d'exécution portable, efficace pour les applications parallèles irrégulières avec ordonnancement dynamique de charge, offrant des outils pour la visualisation et l'évaluation de performances [Ste97]. Il est constitué d'un noyau exécutif parallèle, appelé ATHAPASCAN0 [Chr96, BGP96], et d'une interface de programmation applicative, ATHAPASCAN1 [Bri96]. Comme on l'a vu, le niveau ATHAPASCAN0, un noyau exécutif portable, offre les fonctionnalités de création locale et à distance de processus légers, de communications et d'accès distants à la mémoire. Dans la figure 4.1, les deux couches de l'environnement sont présentées.

ATHAPASCAN1 est une bibliothèque C++ pour la programmation parallèle. Cette bibliothèque permet d'exprimer un parallélisme explicite basé sur la description d'un graphe de précédence sans cycle " DAG ". Ce DAG est soumis pour son exécution à un ordonnanceur dynamique. La construction dynamique du DAG permet de dissocier la gestion de l'ordonnancement de l'écriture d'un programme, simplifiant ainsi le travail du programmeur. Plusieurs ordonnanceurs dynamiques sont proposés par ATHAPASCAN1. Dans ce document nous utiliserons ATHAPASCAN1 pour nous référer à ATHAPASCAN-1a .

4.3 ATHAPASCAN**1 : Interface applicative**

ATHAPASCAN1 s'utilise à travers une interface applicative écrite en C++. Un ensemble d'algorithmes d'ordonnancement est offert (work-stealing, cyclique, glouton centralisé, aléatoire) et des outils de développement permettent à l'utilisateur d'en écrire de nouveaux.

[2] Cette première interface a été conçue par J.L. Roch, M. Doreille, G. Cavalheiro, 1996

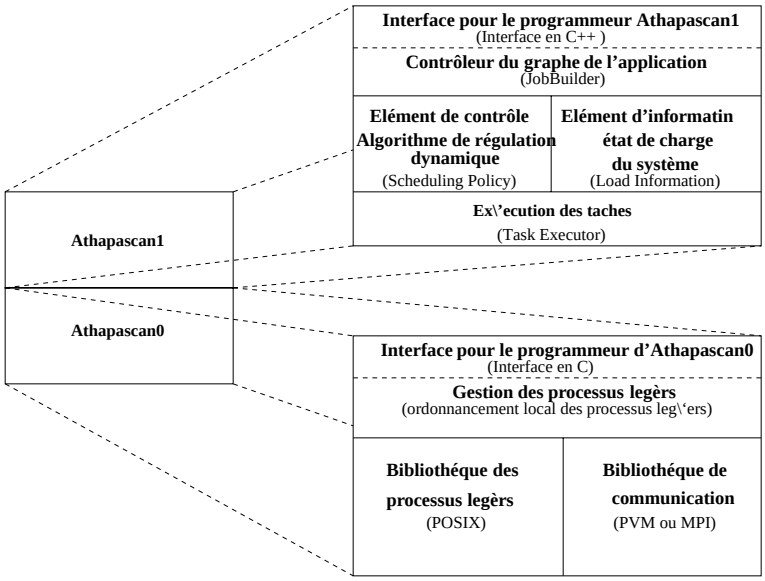


FIG. 4.1 – Les deux couches du système ATHAPASCAN

Comme ATHAPASCAN1 est écrit en C++, des références au mécanisme de constructions des applications orientées objets seront utilisés (notions de classe, d'héritage, d'objet, de classe virtuelle, de familles de classes)[Cal94].

4.3.1 ATHAPASCAN**1, un modèle de programmation**

ATHAPASCAN1 [BGR96] a été conçu pour séparer l'écriture d'un algorithme parallèle de la manière dont il va être ordonnancé. La gestion des communications et la distribution des tâches et des données est prise en charge par un noyau d'ordonnancement qui gère également la granularité des tâches. ATHAPASCAN1 propose un modèle de programmation basé sur l'appel de procédure à distance. Le graphe de précédence DAG créé dynamiquement, est construit par dérivation de classes C++ pré-définies par ATHAPASCAN1 et qui correspondent :

- soit à des requêtes élémentaires (appel léger asynchrone de procédure à distance),
- soit à des squelettes des graphes génériques du type **découpe-calcul-fusion** (*Split-Compute-Merge*), qui est un appel parallèle d'un ensemble de requêtes indépendantes nommées multi-procédures ou («RP^{2}C» pour *Remote Parallel Procedure Call*)(voir figure 4.2)

Une tâche est représentée par un objet C++. La classe de base en ATHAPASCAN1 est nommée *a1cExecutionGraph*, cette classe offre deux types d'opérateurs : des méthodes d'accès et des méthodes de construction [Bri96]. Le programmeur d'une application ATHAPASCAN1 voit seulement les méthodes d'accès. Pour construire une application, l'utilisateur doit définir une classe concrète qui dérive de la classe abstraite *a1cExecutionGraph*.

Nous allons maintenant décrire quelques méthodes parmi les plus importantes qui

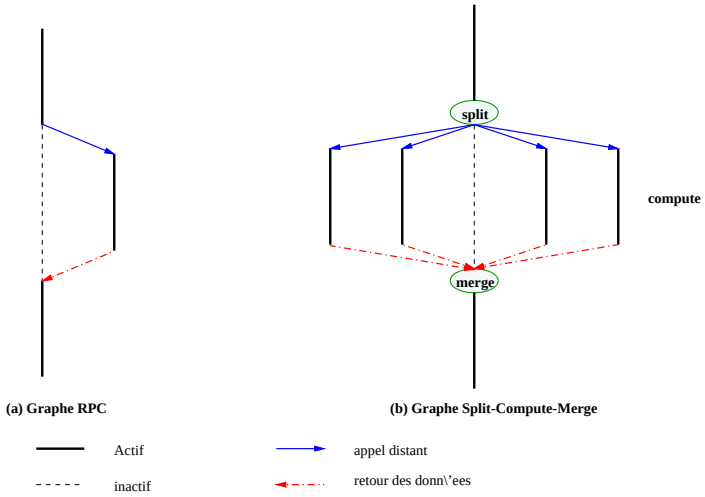


FIG. 4.2 – Appel de procédure à distance et appel de multi-procédures.

caractériseront une tâche ATHAPASCAN1 et qui sont utilisées par le régulateur. Afin de permettre un placement initial plus rationnel, des informations de coût, de découpe et de localité peuvent être associées aux tâches.

Informations de coût

Ces attributs d'une tâche de type *a1cExecutionGraph* peuvent être statiques ou dynamiques (la durée de calcul de la tâche est une estimation). Les informations de coût d'une fonction ne sont calculables qu'avec la connaissance de ses arguments effectifs et en fonction de quelques paramètres (comme la taille de ces paramètres). Ces informations sont le coût estimé en calcul et/ou le coût estimé en communication. Elles sont exprimées dans un modèle arithmétique. Le programmeur inclut cette information en spécifiant la méthode *a1CostInfo()*.

Informations sur la découpe

Les informations de découpe pour l'ordonnancement sont uniquement des informations de type "directives" sur les possibilités de découpe d'un problème en k sous-problèmes indépendants : l'intervalle possible pour le paramètre k est précisé et la valeur est choisie à l'exécution par le séquenceur ou régulateur. La définition de cet intervalle est à la charge du programmeur et dépend d'une instance particulière du problème. Elle ne se base que sur une étude *a priori* de la complexité de l'algorithme programmé. Ces informations sont représentées par la méthode de type *a1GetSplitFacteur()*.

Informations sur la localité

Comme dans les deux autres cas, il est possible de donner des directives de localité pour l'exécution d'une tâche. L'attribut de localité pour une tâche peut prendre l'une des

valeurs suivantes :

- ANY : le placement de la tâche est déterminé par l'ordonnanceur.
- LOCAL : la tâche doit être exécutée sur le même processeur physique que celui où elle a été créée.
- FIXED(p) : l'utilisateur impose le placement sur le processeur p. Cette fonctionnalité permet d'exécuter une application avec un ordonnancement statique, par exemple
- SEQ : une autre relation est la séquentialité, deux tâches seront exécutées en séquence sur le même processeur virtuel.

4.4 Le DAG ATHAPASCAN1

Nous allons maintenant décrire brièvement les sous-classes les plus importantes de la classe abstraite *a1cExecutionGraph*. L'écriture d'un nouveau graphe de la classe *a1cExecutionGraph* consiste en la définition d'une classe qui hérite des méthodes de la classe base. Cet héritage nécessite la spécialisation de certaines méthodes qui permettent de spécialiser un type particulier de graphe.

4.4.1 Classes élémentaires

Un appel de procédure à distance a deux éléments, un appelant (le client) et un appelé (le serveur). Toute requête élémentaire est implementée par un couple d'objets *Client-Serveur* :

- Le *Client* créé dans l'appelant, définit la fonction à exécuter ainsi que ses entrées : c'est un objet de la classe *a1cExecGraphDescRC*, sous-classe de *a1cExecutionGraph*.
- Le *Serveur* est l'implémentation de la fonction proprement dite, c'est le code de la fonction. C'est l'objet à placer pour l'ordonnanceur et il appartient à la classe *a1cRequestServer*, sous-classe de *a1cExecutionGraph*.

Classe Client : *a1cExecGraphDescRC*

C'est la classe de base d'un algorithme appelant une fonction sur un autre site. La dérivation d'une classe pour une application parallèle nécessite la redéfinition des méthodes permettant de transmettre les paramètres, des données, ainsi que d'identifier la tâche qui sera exécutée sur le site distant, puis à la fin de récupérer les résultats de cette procédure distante. Il s'agit de :

1. *a1CostInfo()* : est une estimation du coût des calculs qui seront exécutés ou celui des communications nécessaires (hérité de la classe *a1cExecutionGraph*).
2. *a1PutArgs(a1cOStream& args)*: cette fonction est appelée pour " emballer " les arguments, le type *a1cStream* permettant de transmettre des objets de n'importe quel type ; c'est un médium de communication d'objets non-typés. Le type *a1cOstream* permet d'écrire dans le médium (sortie, output).

3. *a1GetRes(a1cIStream& res)* : cette fonction est appelée pour " déballer " le résultat de l'appel à distance. De la même manière, le type *Istream* permet de lire des objets dans le médium (entrée, input).

Classe Serveur : *a1cRequestServer*

L'exécution à distance d'un calcul (décrit par les classes *a1cExecGraphDescRC* et *a1cExecGraphDescSCM*) se fait par la création d'un objet du type *a1cRequestServer* à travers un créateur de tâche. Une tâche possède plusieurs attributs :

- des arguments d'une fonction ;
- des informations sur le coût estimé ;
- des informations sur la localité d'exécution ;
- des informations sur sa priorité d'exécution ;
- le propre code de la fonction.

Une tâche associée à un algorithme parallèle de type *a1cRequestServer* nécessite de redéfinir les méthodes suivantes qui seront appelées après sa création :

1. *a1Init(a1cIStream& args)*: fonction chargée d'initialiser les entrées de la tâche accessibles à travers le médium de communication de type *a1cIStream* (accessible en lecture seulement).
2. *a1Main()* : fonction appelée pour l'exécution du code de la tâche.
3. *a1End(a1cOStream& res)* : fonction chargée de transmettre, à travers un médium de communication de type *a1cOStream* (accessible en écriture seulement), le résultat de la tâche.

4.4.2 Classes *Split-Compute-Merge*

Dans le cas d'un appel du type multi-procédures (*Split-Compute-Merge* ou *SCM*) plusieurs appels de procédure de découpe du calcul à distance se font avec un seul appelant.

Classe *Client* : *a1cExecGraphDescSCM*

C'est la classe de base des algorithmes du type "découpe des instances, calcul de sous-instances et fusion de résultat". Le choix de la découpe est pris par l'ordonnanceur grâce au facteur de découpe que le programmeur donne dans la méthode *a1GetSplitFactor()*. Les méthodes à redéfinir sont :

1. *a1GetConstInfo()* : retourne une estimation du coût des calculs qui seront exécutés et celui des communications.
2. *a1GetSplitFactor()* : retourne l'intervalle de découpe possible.
3. *a1SplitClient (i, K, a1cOStream& args)* : cette fonction est appelée afin de transmettre à travers le médium de communication de type OStream la i-ème sous-partie (parmi le K de la découpe) des arguments qui serviront en entrée de la i-ème tâche de calcul.

4. *a1MergeClient (i, K, a1cIStream& res)* : cette fonction est appelée pour fusionner le i-éme sous-résultat des K sous-résultats de la découpe. Le résultat est accessible à travers le flot de communication de type IStream.

Classe *Serveur* : *a1cRequestServerSCM*

Dans le cas d'une tâche du type *a1cRequestServerSCM* (dérivant de la classe *a1cRequestServer*), l'interface de la méthode *a1Init()* change afin de prendre en argument son numéro *i* parmi les K découpes effectuées :

1. *a1Init (i, K, a1cIStream et args)* : les arguments i et K correspondent aux arguments passés lors de l'appel à la fonction *a1SplitClient()*. La valeur de i correspond au numéro de la tâche parmi les tâches K crées lors de la découpe des données de l'instance.

4.5 Interaction entre l'application ATHAPASCAN1 et l'ordonnanceur

La programmation en ATHAPASCAN1 consiste à spécifier un graphe G où les tâches communiquent selon les relations de précédence. Une fois ce graphe de précédence décrit, il doit être soumis à l'ordonnanceur du système (classe *a1sSchedule*) pour son exécution par un appel à la primitive *a1Execute(G)* (voir code suivant).

```
main(...){

int res;
// déclaration d'un graphe A1 du type ClientArbre
ClientArbre QuickSort()
// déclaration de l'ordonnanceur
a1sSchedule<a1sDecision, a1sJobManager> MySche;
// Soumission du graphe
MySche.a1Execute(QuickSort())
// attent de la fin de l'execution de l'application
res = QuickSort.Result();

}
```

La classe *a1sSchedule* et ses sous-classes *a1sDecision*, *a1sJobManager*, *a1sLoadCollector*, *a1sJobBuilder* seront décrites dans le chapitre suivant.

L'ordonnanceur calcule en développant le graphe, les nœuds de G en choisissant le degré de granularité pour les tâches et sélectionne celles sans prédécesseur pour les allouer

à des processeurs. Au fur et à mesure que les tâches sont exécutées, leurs successeurs sont recherchés dans une nouvelle expansion du graphe.

Le graphe d'ATHAPASCAN1 est développé dynamiquement et toutes les tâches ne sont pas créées dès sa soumission. À l'instant t, Q' est la liste de l'ensemble des tâches prêtes $Q'(t) = \{T_1, \ldots, T_k\}$ (toutes les tâches sont deux à deux indépendantes), cette liste peut-être centralisée ou distribuée. L'interface entre l'application et l'ordonnanceur peut être vue comme une liste dynamique des tâches prêtes. Lorsqu'une tâche doit être allouée à un processeur, l'ordonnanceur doit choisir une nouvelle tâche de la liste suivant un critère spécifique.

Des informations pour l'ordonnancement peuvent être associées aux tâches : le programmeur peut établir des priorités pour la localité *a1Locality()* ou donner des bornes de découpe pour l'adaptation de la granularité *a1SplitFacteur()* ou même des informations auxiliaires sur l'estimation du coût de la tâche *a1ConstInfo()*.

L'interaction entre les applications ATHAPASCAN1 et le régulateur dynamique de la charge est donc la suivante :

- Le choix de découpe est fait par ordonnanceur sur des informations précisées dans le programme afin de limiter son degré de parallélisme en interdisant la génération de tâches trop petites.
- Le choix des processeurs et le séquencement des tâches dépendent à la fois de la charge de la machine et de la technique d'ordonnancement utilisée. Les informations de coût (temps et calcul) peuvent être utilisés par le séquenceur pour le choix des processeurs.
- Le choix des processeurs peut dépendre de la priorité donnée à la localité des données.

4.6 Conclusion

Ce chapitre a introduit d'une part les noyaux exécutifs à base de " processus légers ", dont celui d'ATHAPASCAN0. Il a aussi décrit l'interface applicative ATHAPASCAN1 et la façon dont les tâches sont générées. Le chapitre suivant fait le lien entre les deux couches et définit le rôle de l'ordonnanceur (ou régulateur de charge).

4.6.1 Utilisation d'ATHAPASCAN1

Nous avons une vision comme utilisateur de la bibliothèque ATHAPASCAN1, c'est sur cette interface que nous avons écrit nos applications parallèles irrégulières. Nos applications sont **synthétiques**, dans les sens qu'elles consument les ressources du système mais sans faire aucun calcul réel. Elles vont simuler le comportement des applications parallèles irrégulières, dans le cas présent le modèle proposé par ATHAPASCAN1. Deux classes de bases ont été définis (voir section 6.3).

L'exemple suivant illustre la déclaration d'une nouvelle classe dérivée de *a1cRequestServer*. Nous illustrons ici une partie du code des programmes synthétiques que nous

avons implémentés. Ce modèle est du type diviser-pour-paralléliser. D'abord la déclaration d'un *Serveur* nomme *NoeudArbre*.

La classe *Client* qui est le complement de ce *Serveur*sera décrit dans la section suivante 4.6.1.

Classe *NoeudArbre*

La nouvelle classe *NoeudArbre* dérive du type classe *a1cRequestServer*

```
class NoeudArbre : public a1cRequestServer {
public:
 // constructeur de base d'une instance
  NoeudArbre(){};
 // destructeur
  ~NoeudArbre(){}
 // réception des arguments
  void a1Init( a1cIStream& b );
// corps principaux de la procédure
  void a1Main( void );
 // emballage des résultats
  void a1End( a1cOStream& b );

  friend a1cIStream& operator>>( a1cIStream & in, _NoeudArbreParam& ndprm ) ;
  friend a1cOStream& operator<<( a1cOStream & out, _NoeudArbreParam& ndprm );

private:
  // structure complexe pour les paramètres des graphes synthétiques
  _NoeudArbreParam* NdPrm;
  int  cumm;     // pour les accumulations des communications
  int  dept ;    // information sur la profondeur du graphe
} ;
```

Maintenant nous montrons le code pour les méthodes obligatoires pour le serveur : le *a1Init(), a1Main()* et *a1End()*.

class NoeudArbre::a1Init

```
void NoeudArbre::a1Init( a1cIStream& b ) {
    _NoeudArbreParam* nodetmp = new _NoeudArbreParam ;
     NdPrm = new _NoeudArbreParam ;
     // reception des paramètres du graphe synthétique ;
     b >> *nodetmp ;
     NdPrm = nodetmp;
     cumm = 0; // cumm local au noeud
     // réception de données
     vector<int> VectDonnees;
     VectDonnees.insert( VectDonnees.begin(), NdPrm->DimData, int()) ;
     a1Unpack(b, VectDonnees.begin(), NdPrm->DimData) ;
}
```

class NoeudArbre::a1End

```
void NoeudArbre::a1End( a1cOStream& b ) {
  vector<int> vect;
  int taille;
     // envoi des resultats
     taille = NdPrm->dim_result(cumm);
     b << taille;
     vect.insert(vect.begin(),taille,int(0));
     a1Pack( b, vect.begin(), taille ) ;
}
```

class NoeudArbre::a1Main

```
void NoeudArbre::a1Main( void ){
  int lim;
  double t1;
     if (NdPrm->condition_arret()) {
        // je suis une tache de calcul
        lim = NdPrm->cost_task();
             time_working(lim);
        cumm = 1;
        if ( NdPrm->BackResult == 0)
             // accumulation
             cumm = NdPrm->ParamResult;
        dept = NdPrm->depth ; // profondeur du graphe
     } else {
// je suis une tache génératrice de parallélisme
      // appel à l'ordonnanceur
         MySched sched;
         // Création d'un nouvel graph
         ClientArbre arbre( *NdPrm );
         // Exécute le graphe ...
         sched.a1Execute( arbre );
         lim = NdPrm->cost_fusion();
              time_working(lim);
          cumm = arbre.Accum();
      }
}
```

Classe *ClientArbre*

Nous illustrons l'utilisation de cette classe *a1cExecGraphDescSCM*, en présentant le code de la classe utilisée pour les programmes synthétiques. D'abord la déclaration de l'objet.

La nouvelle classe ClientArbre dérive du type a1cExecGraphDescSCM

```
class ClientArbre : public a1cExecGraphDescSCM<NoeudArbre> {
public:
 // constructeur de base
  ClientArbre(){};
 // destructeur
  ~ClientArbre(){}
 / Constructeur d'une instance : initialisation des données
  ClientArbre(_NoeudArbreParam& nt);
 // méthode pour le retour des résultats
  int Accum() { return Cumm; };
 // coût de la fonction
   a1cCostInfo a1GetCostInfoSCM( int ith, int K );
 // nombre de fils pour la découpe
   a1cSplitFactor a1GetSplitFactor () { return a1cSplitFactor( NdPrmCl->NbSon ) ;}
 / découpe, avec l'emballage des arguments pour la i-ème tâche
   void a1SplitClient( int ith, int K, a1cOStream& b );
 / déballage du résultat de la i-ème tâche
   void a1MergeClient( int ith, int K, a1cIStream& b );

  friend a1cIStream& operator>>( a1cIStream & in, _NoeudArbreParam& ndprm ) ;
  friend a1cOStream& operator<<( a1cOStream & out, _NoeudArbreParam& ndprm ) ;

private:
  _NoeudArbreParam* NdPrmCl;
  int   Cumm;
  // pour les valeurs aléatoires
  vector<double>   VectParamAleatoire;
```

```
} ;
```

ClientArbre : constructeur

```
ClientArbre::ClientArbre(_NoeudArbreParam& nt):res(0), Cumm(0){
  NdPrmCl =  &nt;
  int lim= NdPrmCl->cost_division();
         time_working(lim);

  // Méthode envoi répartition aléatoire (uniforme et gauss)
    if ((NdPrmCl->MetSendParam) == 2 ||(NdPrmCl->MetSendParam) == 3 ){
        VectParam.insert(VectParam.begin(), (NdPrmCl->NbSon)+1,double(0) );
        for (int i=0; i<= (NdPrmCl->NbSon); i++)  VectParam[i] =  drand48();
    }
}
```

ClientArbre::a1SplitClient

```
void ClientArbre::a1SplitClient( int ith, int K, a1cOStream& b ) {
     // nécessaire le nodetmp pour réduire la profondeur
     _NoeudArbreParam* nodetmp = new _NoeudArbreParam ;
     *nodetmp = *NdPrmCl;
     (nodetmp->Depth)--;

     nodetmp->DimData = nodetmp->dim_data(VectParam[ith+1],VectParam[ith]);

     b << *nodetmp;
   // envoi des données
     vector<int> vect;
     vect.insert(vect.begin(),nodetmp->DimData,int(0));
     a1Pack( b, vect.begin(), nodetmp->DimData) ;
}
```

ClientArbre::a1MergeClient

```
void ClientArbre::a1MergeClient( int ith, int K, a1cIStream& b ){
  int taille_result;
  vector<int> vect;

      b >> taille_result;
      Cumm += taille_result;
      vect.insert(vect.begin(),taille_result,int(0));
      a1Unpack(b, vect.begin(),taille_result);

}
```

Chapitre 5

L'implémentation du régulateur dynamique de charge du système ATHAPASCAN

Dans ce chapitre l'ordonnanceur dynamique [1] *du système* ATHAPASCAN *est décrit. Le système proposé a été structuré de façon modulaire. Chaque module (est une classe C++) peut être adapté aux besoins de l'application en créant des nouveaux algorithmes d'équilibrage de charge.*

Comme nous l'avons présenté dans la première partie de ce document, les algorithmes de répartition dynamique sont très nombreux, et en proposer un seul dans un système destiné à l'écriture des programmes parallèles comme ATHAPASCAN nous a semblé peu convainquant, surtout parce qu'il n'est pas possible de connaître le type d'application (ATHAPASCAN vise surtout les applications irrégulières) qui seront programmées. Nous avons pour but de proposer un algorithme général pour la répartition dynamique que soit efficace pour l'ensemble des applications.

Nous avons vu aussi qu'il existe des systèmes qui offrent des mécanismes pour la régulation spécifique de certains types d'application (BSP par exemple) ou des systèmes qui offrent des mécanismes pour la répartition des données comme HPF ou d'autres encore qui laissent la responsabilité de la régulation au programmeur.

Au début de ce document, le problème de la répartition dynamique de charge a été décrit comme le problème de consommateurs et de ressources [CK88] où l'ordonnanceur gère l'utilisation des ressources. Pour implémenter ce modèle, sans être obligé de construire un ordonnanceur spécifique à chaque application, l'idée est de créer une interface entre l'application et l'ordonnanceur et une interface entre l'ordonnanceur et la machine parallèle : c'est l'originalité de la proposition de ce régulateur (voir figure 5.1).

Rappelons que l'environnement de programmation parallèle ATHAPASCAN est constitué d'un noyau exécutif parallèle, appelé Athapascan0 [Chr96, Gin97], et d'une interface de programmation applicative, Athapascan1 [Bri96, GRCD98]. Le niveau Athapascan0

[1] Ce travail a été développé en collaboration avec Olivier Schiavo lors de son DEA et avec Gerson Cavalheiro sous la direction de J.L. Roch.

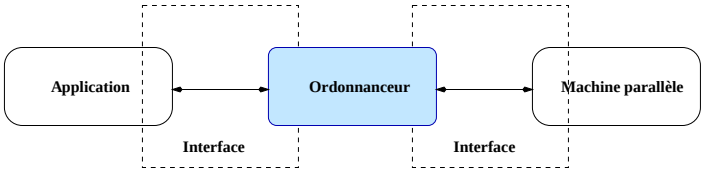


FIG. 5.1 – Modèle consommateur-ressources avec des interfaces.

offre les fonctionnalités de création locale et à distance de processus légers (*threads)*, de communication et d'accès distants à la mémoire. Le niveau Athapascan1, permet la programmation d'applications parallèles par la description d'un graphe de précédence sans cycle qui est soumis à un ordonnanceur dynamique. Cette construction permet de dissocier la gestion de l'ordonnancement de l'écriture du programme, simplifiant ainsi le travail du programmeur.

Dans le chapitre trois, nous avons décrit les fonctionnalités d'un mécanisme de régulation dynamique. Le mécanisme de base est composé de deux éléments, le composant de contrôle et le composant dédié à la collecte de l'information. Etant données les couches de multiprogrammation légère (ATHAPASCAN0) d'un part et d'autre part l'interface applicative (ATHAPASCAN1), l'ordonnanceur est placé entre ces deux couches du système comme le montre la figure 5.2. L'ordonnanceur est modélisé avec ses deux composants de contrôle et d'information et en plus les deux interfaces une pour la machine parallèle cible (ATHAPASCAN0 sera vu comme une machine parallèle virtuelle) et l'autre en direction de l'application parallèle, soit vers ATHAPASCAN1. Les interfaces sont construites en s'adaptant aux couches du système ATHAPASCAN. Ces interfaces seront décrites dans ce document, mais pour plus de détails nous vous dirigeons vers les travaux de G. Cavalheiro [CDR98].

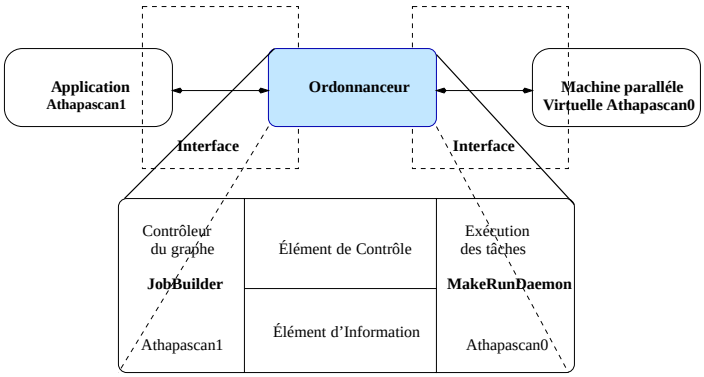


FIG. 5.2 – L'ordonnanceur interfacé entre les couches ATHAPASCAN1 et ATHAPASCAN0.

La souplesse de ce modèle est que l'ordonnanceur n'est pas lié à un modèle d'application ou a une machine donnée, il suffit de modifier les interfaces pour faire l'adaptation [2].

[2]Et c'est précisément ce qui s'est passé quand la nouvelle Interface Applicative Athapascan1b a été développée, l'ordonnanceur a conservé ses fonctionnalités [CDR98].

```
Algorithme A (entre : travail(), sortie : résultat)
{
// où la règle prendre en entrée
// les paramètres de la fonction travail()
algo = règle (type, mode, taille) ;
choix (algo)
cas a : algorithme a(travail(), résultat) ;
cas b : algorithme b(travail(), résultat) ;
cas c : algorithme c(travail(), résultat) ;
}
```

La modularité de l'ordonnanceur permet de structurer un ensemble d'algorithmes d'équilibrage dynamique de charge. Il est seulement nécessaire de dériver la classe à laquelle nous intéressons, et de réécrire les méthodes concernées. Du fait du grain de parallélisme, généralement fin, des applications, on n'a pas considéré la migration de tâches en cours d'exécution. Comme on le verra, une version souple de la migration a été proposée en considérant la possibilité de migrer les tâches avant leur exécution (replacement des tâches).

Différentes possibilités ont été envisagées pour l'implémentation de l'ordonnanceur. On a cherché une structure fonctionnelle, en offrant des briques de base pour l'écriture des algorithmes de régulation de charge. Une possibilité envisagée a été l'utilisation des poly-algorithmes [Sni92] c'est-à-dire qu'un algorithme de régulation est décrit non pas par un seul algorithme, mais par un ensemble d'algorithmes possibles et une règle de choix qui doit déterminer quel algorithme utiliser dans un cas particulier, en fonction des paramètres d'entrée de l'application (type de fonction, taille des données à traiter, granularité, découpe, etc.). Dans le code suivant un exemple de tel type de poly-algorithme est présenté :

Le problème principal est d'établir la règle de choix, qui à partir des informations sur l'application doit prendre des décisions "intelligentes". Cette règle de choix reste en principe à la charge du programmeur de l'application. Il faudra donner à l'ordonnanceur des éléments pour choisir l'algorithme d'ordonnancement qui s'adapte le mieux au problème posé. La question que nous nous posons est de quelle manière un algorithme d'équilibrage peut-être préféré à un autre. Il est clair qu'ici on donne cette possibilité en fonction de l'application donnée et de la connaissance du programmeur du comportement de son application. Cependant c'est un procédé empirique. Dans la dernière partie de ce document nous proposons une méthodologie pour l'évaluation des algorithmes d'équilibrage de charge et nous essayons de donner une réponse à cette question.

Le choix s'est dirigé vers la construction d'un ensemble de classes C++ de base qui forment la classe complexe *Scheduler*, munies de fonctionnalités primitives et donnant la possibilité de créer de nouveaux algorithmes en dérivants ces classes de base. Il s'agit donc d'une boıîte à outils.

5.1 L'implémentation de l'ordonnanceur du système ATHAPASCAN1

La particularité principale du modèle de programmation d'ATHAPASCAN1 (voir chapitre 4) est de séparer l'expression d'un algorithme donné de la manière dont il sera ordonnancé. La bibliothèque ATHAPASCAN1 fournit donc :

• **Un graphe dynamique de précédences des tâches :** Comme on l'a déjà vu dans le chapitre précédent, le parallélisme est exprimé explicitement par la description d'un graphe de précédences de tâches construit dynamiquement. Les requêtes élémentaires sont du type appel de procédure à distance et les requêtes peuvent être groupées dans les multi-procédures du type *Split-Compute-Merge*.

Toute requête élémentaire est implémentée par un couple d'objets *Client-Serveur* :
- Le *Client* crée dans l'appelant, définit la fonction à exécuter ainsi que ses entrées. Il attend le résultat de la fonction appelée.
- Le *Serveur* est l'implémentation de la fonction proprement dite. C'est le code qui peut être exécuté sur un processeur quelconque.

Pour les multi-procédures un seul *Client* est crée chez l'appelant et il est chargé d'attendre les résultats de toutes les procédures appelées.

• **Un ordonnanceur dynamique :** La philosophie d'ATHAPASCAN1 [BGR96] est de mettre à la disposition du programmeur plusieurs ordonnanceurs. Une fois le graphe décrit, il peut être exécuté grâce à un objet complexe du type *Scheduler* (qui est décrit dans la section suivante). L'ordonnanceur est formé par des modules qui peuvent être adaptés aux besoins de l'application. Comme il est écrit en C++, on dérive des nouveaux ordonnanceurs à partir des classes de base. Les méthodes peuvent être réécrites si nécessaire. Plusieurs protocoles pour la collecte des informations sont proposés et peuvent avoir des interactions avec les différentes stratégies de placement dynamique expérimentées. Parmi celles-ci se trouvent deux stratégies réparties et sans élément d'information (un aléatoire et un cyclique), un ordonnanceur du type ferme de processeurs [Sch96], un glouton centralisé, et des algorithmes décentralisés basés sur une politique de seuils. La figure 5.3 montre l'implémentation modulaire du régulateur de charge réalisé dans ATHAPASCAN1.

Des fonctionnalités d'ATHAPASCAN1 sont supportées par l'ordonnanceur, le programmeur d'une application ATHAPASCAN1 peut établir des priorités pour la localité des données, donner des bornes (degré de découpe) pour l'adaptation de la granularité et des informations auxiliaires tel que l'estimation du coût de la procédure, la taille des données.

5.2 L'architecture du régulateur

Au vu des considérations précédentes, et de la multiplicité des stratégies d'implémentation possibles, le but de la spécification est de fournir des briques de base permettant un cadre le plus général possible. La programmation en C++, avec l'utilisation de l'héritage,

de la structuration et de la généricité a permit de créer un environnement de programmation modulaire et aisément extensible.

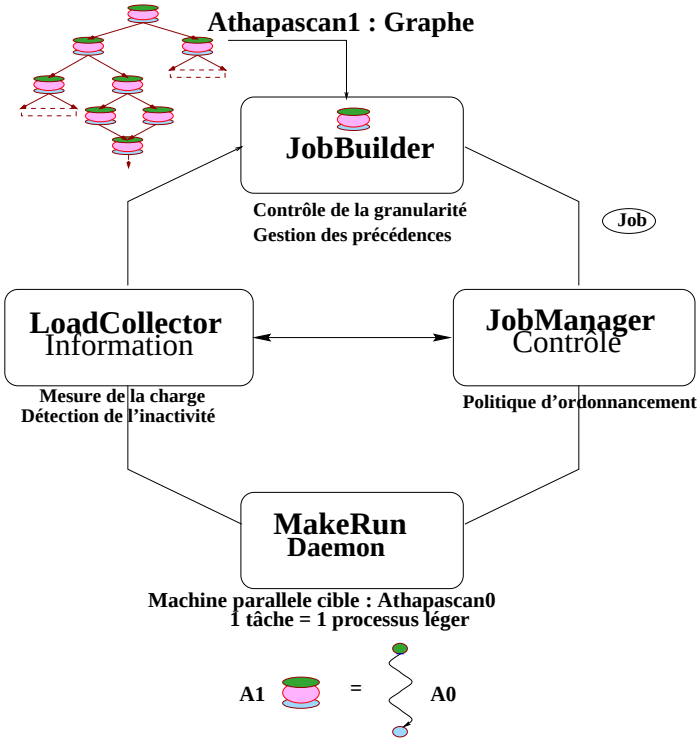


FIG. 5.3 – Architecture du régulateur de charge dans ATHAPASCAN1

Les classes de base proposent un cadre ouvert pour l'écriture des divers types d'algorithmes d'équilibrage dynamique de charge, mais suffisamment précis pour donner une cohérence à la bibliothèque d'ordonnanceurs [Sch96]. L'héritage nous autorise à définir le comportement général d'un objet *Scheduler*, et à spécialiser ensuite ce comportement pour chaque classe implémentant un module particulier. Conçu de façon très modulaire, le *Scheduler* est formé des sous-objets de base qui permettent de mettre en œuvre les fonctionnalités longuement décrites au sein du chapitre trois.

La soumission d'un graphe au *Scheduler* dans une application ATHAPASCAN1 est un appel bloquant du flot d'exécution, il doit être vu comme l'implémentation de la méthode *Execute* de l'ordonnanceur. Le flot d'exécution de l'application ATHAPASCAN1 sera repris quand le graphe aura été entièrement exécuté.

Quatre modules forment cet objet complexe (*Scheduler*) [Sch96] : le *JobManager* correspond à l'élément de contrôle, le *LoadCollector* à l'élément d'information, le *JobBuilder* est l'interface avec l'application ATHAPASCAN1, et le *MakeRunDaemon* est l'interface vers ATHAPASCAN0 (voir fig 5.3).

Les quatre modules se trouvent dans chaque processeur et la communication entre eux est explicite et sans coût (communication locale). Par contre chaque module peut communiquer avec un autre du même type chez un autre processeur (voir figure 5.4) par des messages explicites.

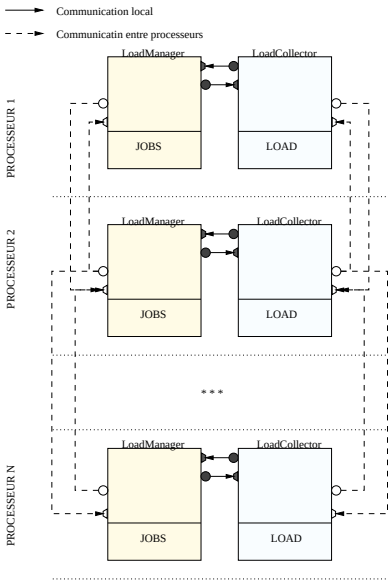


FIG. 5.4 – Les communications entre les objets *JobManager* et les objets *LoadCollector*

5.2.1 Une entité fonctionnelle : le *Job* ATHAPASCAN**1**

L'ordonnancement d'un graphe de tâches devient vite très compliqué même dans des cas «simples». Dans Pyrros [YG92] par exemple, la construction du graphe statique va occuper beaucoup d'espace mémoire. Contrôler toute cette information devient vite très lourd compte tenu du fait de sa complexité - tant de point de vue de la taille mémoire que de sa structure -. Dans notre cas, le graphe est crée dynamiquement et il n'est pas aisé de traiter directement le graphe pour créer un ordonnancement. Donc, l'idée est de travailler avec un nombre réduit d'informations décrivant les tâches du graphe, comme le fait Dynamo [Tar92]. Une tâche Dynamo est décrite à partir de certains attributs et un pointeur vers les données (non pas le code de la tâche) mais un ensemble d'informations qui permettent au régulateur de trouver un site d'accueil pour la tâche.

Dans la présente implémentation, l'appel de procédure à distance est représenté par un couple *Client-Serveur*. Alors nous nommerons *Job* une référence au couple *Client-Serveur*. Le *Job* est un objet mobile, avec un ensemble d'attributs qui décrivent la fonction qui sera exécutée et qui devront aider l'ordonnanceur à mieux placer la tâche : le *Serveur* qui permet de retrouver la source ou tâche mère : le *Client*. Cette référence est la plus légère possible pour pouvoir être communiquée à moindre coût. Dans le tableau 5.1, les attributs du *Job* sont décrit. Les *Jobs* sont les objets qui vont être manipulés par l'ordonnanceur pour gérer l'exécution du graphe et réguler la charge dans le système.

Si un *Job* n'a pas encore débuté son exécution, il peut "migrer" et passer d'un processeur à un autre. Pour son exécution, un processus léger lui sera dédié. Les informations d'une tâche ATHAPASCAN1 (comme on l'avait vu dans le chapitre précédant) : coût, localité, priorité, données par le programmeur sont utilisées par l'ordonnanceur comme guides pour mieux placer cette tâche, ces informations étaient copiées dans l'objet *Job* correspondant.

La méthode *MakeRun* est l'interface pour exécuter le Job (*Job.MakeRun()*), et c'est le *MakeRunDaemon* qui est responsable de cette action. Le *MakeRunDaemon* est l'interface avec le noyau exécutif ATHAPASCAN0, c'est lui qui crée les processus légers responsables de l'exécution de la fonction. Après avoir débuté le *Job*, son avancement est contrôlé par le noyau ATHAPASCAN0.

Méthode	Description
MakeRun()	Fonction qui permet l'exécution " réelle " de la tâche décrite par le *Job*. Cette méthode est exécutée par le *MakeRunDaemon*, la tâche *Serveur* (qui est la procédure appelée) est débutée et un processus léger lui est dédié.
Attributs	Description
Client	Référence globale à la tâche *Client* (source du *Job*).
sch	Référence globale à l'ordonnanceur qui exécute le graphe.
coût	Coût estimé de la tâche serveur donné par le programmeur.
localité	Des directives pour la localité des données.
priorité	Priorité d'exécution pour la tâche *Serveur*.
taille	Taille des données.

TAB. 5.1 – La classe *a1Job*

Les objets actifs/réactifs

Les *Daemons* sont des objets statiques présents dans le système pendant toute l'exécution, ils ont la fonction de s'activer à la détection d'un événement (création d'une tâche ou d'un *Job*) ou à la réception d'un message (demande d'information sur la charge, demande de travail). Le *Daemon* activé effectue son travail et puis s'endort en attendant un nouvel événement d'activation. Le *JobManager* et le *LoadCollector* sont les classes qui ont des objets *Daemons*. Au début de l'exécution d'une application, un processus léger est assigné à chaque objet, ces objets sont détruit à la fin de l'application.

Le *MakeRunDaemon* comme son nom l'indique est un objet actif. Il est chargé d'assigner un processus léger à toute tâche *Serveur* quand il débute son exécution. Cela signifie que pour chaque tâche *Serveur* ATHAPASCAN1 un processus léger ATHAPASCAN0 lui est dédié. Le *MakeRunDaemon* est chargé de l'interface avec la machine parallèle ATHAPASCAN0.

5.2.2 Le *JobBuilder* : contrôle de la granularité et expansion du graphe

Pour l'ordonnanceur, le *JobBuilder* est le producteur des *Jobs*. Il pourra être modélisé comme une boîte noire qui génère des tâches. A partir d'un graphe de précédence du type ATHAPASCAN1, il génère les *Jobs* pour le *JobManager*. Son travail se décompose en deux phases principales : l'expansion du graphe (contrôle de la granularité) et le contrôle des

dépendances entre les tâches au fur et à mesure que le graphe est exécuté. Rappelons que c'est un graphe dynamique.

L'appel de la méthode *Execute* chez le *Scheduler* provoque la soumission du graphe au *JobBuilder*. Comme le graphe est un graphe de précédence dynamique, toutes les tâches ne peuvent pas être générées dès la soumission du graphe : c'est au cours de l'exécution qu'elles le seront. Le tableau 5.2 montre les principales méthodes de cette classe. Le *JobBuilder* génère le graphe et crée des tâches (*Client-Serveur*) exécutables tout de suite, c'est-à-dire des tâches dont les prédécesseurs ont été exécutés. Il crée le *Client* dans le processeur appelant et le *Job* avec ses attributs de base (adresse locale de la tâche *Client*, référence à l'ordonnanceur, estimation de coût de la tâche *Serveur*, priorité et information de localité).

C'est lors de l'expansion d'un graphe que la granularité est choisie. Le cas typique est celui du graphe série-parallèle (*Split-Compute-Merge*) : le programmeur donne des bornes pour le facteur de découpe, le *JobBuilder* va choisir dans cet intervalle le nombre de tâches à produire au moment du *Split*.

Une fois qu'un *Job* (et donc la tâche qu'il représente) a terminé ses calculs, le *JobBuilder* qui avait créé ce *Job* est informé pour qu'il puisse faire une mise à jour des précédences du graphe.

5.2.3 Le *JobManager* : élément de décision

Le *JobManager* va ordonnancer les *Jobs* sur les processeurs disponibles. C'est le cœur de la régulation et c'est à ce niveau que les décisions sont prises. Il représente la fonctionnalité de contrôle que nous avons décrit dans le chapitre trois. Cette classe doit obligatoirement fournir la fonction membre *AddJob* (*AddJob* et des autres méthodes sont décrites dans le tableau 5.3). Cette méthode est appelée par le *JobBuilder* local qui donne les nouveaux *Jobs* pour leur placement. Pour la prise de décision le *JobManager* doit avoir accès aux informations sur la charge, cela signifie un protocole avec le *LoadCollector* local. Les classes de base du *JobManager* conforment les différents protocoles avec les autres modules du *Scheduler*. Pour les décisions proprement dites, une classe abstraite a été créé : la classe *Decision*. Nous décrivons maintenant cette classe associée au *JobManager*.

Les politiques de décision

Pour pouvoir dériver aisément et implanter différents algorithmes d'équilibrage de charge une classe virtuelle, la classe *Decision* a été créée. Les principales méthodes de cette classe, associée au *JobManager*, sont présentées dans le tableau 5.4.

Méthode	Description
Submit(graph)	Soumission du graphe, fonction appelé par le *Scheduler*.
InsertList()	Lors de l'expansion du graphe, les tâches prêtes à être exécutée forment une liste qui sera envoyée au *JobManager*.

TAB. 5.2 – La classe *JobBuilder*

Méthode	Description
AddJob()	Le *JobManager* reçoit la liste de *Jobs* prêts du *JobBuilder* local. Et il prend la décision de garder les *Jobs* ou de les envoyer : fait appel à *LocalDecision()*.
SendJobVector()	Méthode qui envoie les jobs vers un autre processeur.
MakeRunLocalJobs()	Méthode qui exécute les jobs dans la liste locale (*Qlocal*). C'est un *Daemon* activé par un *sémaphore*.
MakeRunExternJobs()	Méthode qui s'active à la réception des *Jobs* d'un autre processeur. Pour la prise de décision elle fait appel à *GlobalDecision()*.
ActiveControlReserve()	Contrôleur de la liste de réserve globale des *Jobs* sans placement, activé par des demandes de travail, par la réception des nouveaux *Jobs* ou par la réception des messages sur l'état de charge dans les processeurs.
SomeOneLookForWork()	Méthode qui s'active à la réception d'une demande de travail de la part d'un autre processeur.
LookForWork()	Méthode qui, lorsque le processeur n'a plus de travail, va chercher ou demander de travail à un autre processeur.
Attributs	Description
Qlocal	La liste des jobs placés dans le processeur pour leur exécution.
Qreserve	La liste globale de réserve.

TAB. 5.3 – La classe *JobManager*

Deux décisions principales sur le placement d'un *Job* sont prises :

1. Au moment de la création du *Job*, par le processeur source. Il fait appel à la méthode *LocalDecision*.
2. A tout autre moment, si la tâche n'a pas été exécutée tout de suite, elle est placée dans une liste (globale ou locale). La décision de placement doit alors être activée par d'autres événements. La méthode appelée est *GlobalDecision*.

Lors de la création d'un nouveau *Job*, le processeur source ou propriétaire de la tâche doit prendre une décision : *LocalDecision* qui concerne principalement le maintient de ce *Job* localement pour son exécution ou son exportation vers un autre processeur. Dans une implémentation centralisée, il l'envoie vers le processeur contrôleur. Un seuil est utilisé *MAXT*. Si *MAXT* a une valeur nulle, le processeur envoie toujours les *Job* vers le contrôleur. Par contre, le processeur va décider de garder la tâche si la longueur de sa liste locale est plus petite que la valeur du seuil ($taille(Qlocal) < MAXT$).

La *GlobalDecision* sont les décisions que va prendre le *JobManager* à tout autre moment sur les *Jobs* qui sont prêts et qui se trouvent dans les listes de chaque processeur (ou dans le cas centralisé dans la liste centralisée). Le *JobManager* doit gérer au moins une réserve de *Jobs*. Cette gestion peut être entièrement centralisée - c'est le cas le plus

simple à implémenter - partiellement - voir complètement de façon distribuée. L'idée est de répartir sur un ou plusieurs processeurs les listes de *Jobs* à ordonnancer.

Considérons le cas centralisé, une liste avec les *Jobs* sans placement est présente dans le contrôleur, selon la politique de décision (à l'aveugle, ou en prenant en compte les charges des processeurs), le contrôleur décide le placement de chaque *Job*, et puis envoie chaque *Job* vers son site d'accueil. Le contrôleur peut envoyer plus d'un *Job* à la fois, ce nombre est contrôlé par un seuil *JOBT*.

Méthode	Description
LocalDecision()	Décision du site auquel le *Job* de création récente doit être placé. Il est appelé par le *JobManager* quand il reçoit du *JobBuilder* la liste des nouveaux *Jobs*.
GlobalDecision()	Méthode qui décide le site pour les *Jobs* à tout autre moment ou qui se trouvent dans la liste globale.
Attributs	Description
MINT	Seuil minimum, au dessous de ce seuil le processeur cherche de travail.
MAXT	Seuil maximum, au dessus de ce seuil le processeur exporte le travail, au-dessous il le garde dans sa *Qlocal*.
JOBT	Nombre des *Jobs* à envoyer vers un autre processeur.
MRUN	Nombre des *Jobs* qui peuvent débuter leur exécution concurremment.
IMPJ	Nombre des *Jobs* à importer ou à voler des autres processeurs.

TAB. 5.4 – La classe *Decision*

Dans le cas décentralisé la présence de deux seuils est le cas plus courant. Dans la *GlobalDecision*, il y aura deux seuils, *MAXT* pour l'exportation des jobs vers un autre processeur et *MINT* pour l'importation de travail ou une demande de travail.

Pour pouvoir choisir à quel moment allouer un *Job* à un processeur le *JobManager* utilise :

- Des informations du Job.
- Des informations de charge sur l'ensemble des processeurs donnés par le *LoadCollector*.

5.2.4 Le *LoadCollector* : élément d'information et la structure *LocalLoad*

L'évaluation de la charge locale et la collecte de cette information présente à elle seule un problème complexe. Comme nous l'avons vu précédemment, de multiples stratégies peuvent être utilisées. Il faut garder une structure modulaire pour pouvoir appliquer différentes stratégies d'évaluation de charge tout en gardant la même stratégie d'ordonnancement (par exemple : le même JobManager).

Le couple *LoadCollector-LocalLoad* implémente la collecte d'information de charge du système et l'évaluation locale de la charge, la classe *LocalLoad* permet d'utiliser différents indices de charge, cet objet conserve l'information locale à chaque processeur. L'appel aux méthodes qui modifient ces indices doit être explicite. Tous ces indices ne sont pas toujours nécessaires et surtout leur importance varie selon le type d'ordonnancement désiré. Une discussion sera présentée dans la section suivante.

Le *LocalLoad* est un objet statique dans chaque processeur qui garde des informations sur les différents indices de charge (voir tableau 5.5 : nombre de tâches actives, nombre de tâches exécutées, nombre de tâches bloquées ...). Ces indices sont mis à jour à chaque événement dans le processeur.

Indice	Description
load_new	Nombre de tâches créées dans le processeur.
load_init	Nombre de tâches débutées.
load_end	Nombre de tâches terminées.
load_stop	Nombre de tâches bloquées, qui ont initié leur exécution mais qui n'ont pas encore fini. Ces tâches ne peuvent pas être migrées.
load_run	Nombre de tâches en cours d'exécution.
load_map	Nombre de tâches placées sur le processeur. Ces tâches peuvent être migrées.
load_exp	Nombre de tâches exportées (migrées).
load_imp	Nombre de tâches que le processeur a récupérées, dont il n'est pas la source.
load_local	Tâches qui ont été créées et placées localement dont le processeur est la source, le placeur et l'exécuteur.

TAB. 5.5 – indices de charge simples

Le *LoadCollector* (voir tableau 5.6) fait une partie du travail localement. Il calcule à partir des informations du *LocalLoad* (indices de charge) une estimation de charge du processeur. L'ensemble de ces indices donne un indice global de charge de tout le système. Dans une vision centralisée, il maintient un tableau (global) avec les valeurs envoyées par tous les processeurs.

La politique de mise à jour de ce tableau peut prendre des formes multiples. La version implémentée fonctionne par comptage des événements locaux et cette information est envoyée au processeur collecteur (propriétaire du tableau) par échanges volontaires. La fréquence de ces échanges est contrôlée par un seuil *EVENT*.

5.2.5 L'objet régulateur : *Scheduler*

L'ordonnanceur est donc constitué par l'ensemble des objets décrits ci dessus. Ils sont entièrement masqués au programmeur grâce à l'encapsulation dans la classe. Déclarer un objet de type *Scheduler* installe implicitement tous ces sous-objets. L'utilisateur

Méthode	Description
ReqCollectorLoad()	Demande explicite de l'information.
SendLoad(load, proc)	Envoi de l'information sur la charge à un nœud spécifique.
DiffusionLoad(load, teams)	Diffusion de la valeur de la charge vers un groupe de nœuds (*teams*). Les groupes sont soit tous les processeurs présents, soit un groupe partiel : les voisins, un groupe aléatoire.
UpDateInfos()	Actualise l'information.
RecvLoad()	Réception de l'information de la charge.
Evaluation()	Evaluation de l'état de charge du système à partir des informations collectées. La méthode *Evaluation* estime la charge des nœuds et ordonne les processeurs par leurs niveaux de charge dans une liste *ListeProc*.
GetIndice(type)	Récupère la valeur de l'indicateur de charge spécifique.
GetLoad(proc)	Retourne la charge du processeur.
GetProc(rank)	A partir de la liste *ListeProc*, retourne celui qui est dans la position *rank*.
GetRank(proc)	Donne la position du processeur *proc* dans la liste, où les processeurs sont ordonnancés par leurs niveaux de charge.
Attributs	Description
EVEN	Seuil pour contrôler les fréquences des messages pour actualiser l'information
LocalLoad	Valeur de la charge locale.
GlobalLoad	Tableau avec l'information sur la charge de chaque processeur.
ListeProc	Liste ordonnancée, contient les processeurs ordonnancés par leur charge informée dans le tableau *GlobalLoad*.

TAB. 5.6 – La classe *LoadCollector*

ne les manipule jamais directement, il ne voit que les fonctions membres publiques de l'objet[Sch96]. Pour programmer un ordonnanceur, il n'est pas obligatoire de réécrire toutes les classes décrites : par dérivation des classes de base et utilisation de la généricité C++, l'écriture de nouveaux algorithmes se limite aux parties nécessitant une nouvelle implémentation.

5.3 Discussion sur l'implémentation

Rappelons que lorsqu'une procédure est appelée, l'appelant doit attendre les résultats de la procédure appelée. Deux chemins sont suivis par la procédure : soit, c'est une tâche de calcul et elle sera exécutée jusqu'à sa fin, soit, elle est génératrice de parallélisme. Cela signifie qu'elle appelle de nouvelles procédures.

Ces scénarios d'exécution pour les procédures génèrent un ensemble d'événements.

Pour une tâche de calcul, ses événements sont : création, début d'exécution, calcul et fin d'exécution (on n'a pas encore considéré les événements pour leur ordonnancement). Si toutes les tâches en exécution sont seulement des tâches de calcul, indépendantes entre elles, l'exécution concurrente est possible [3]. La charge en cours d'exécution du processeur pourra être mesurée par l'équation :

$$load\_exe = load\_init - load\_end \tag{5.1}$$

où *load_init* et *load_end* sont des indices de charge ou compteurs des événements.

Nous proposons une étude d'un indice plus complexe pour le modèle d'appel de procédure à distance du système ATHAPASCAN1 (voir section 5.3.2). Cet indice vise à utiliser les états de transition d'une tâche (voir figure 3.1). Pour cette étude, il y a cinq états de transitions considérés (numérotés de I à V) et sept transitions. En considérant comme indicateur de charge le nombre de processus créés mais pas exécutés *load_new* [4], les processus en exécution *load_run*, les processus bloqués *load_stop* et les processus qui après avoir été bloqués, sont prêts à être exécutés *load_att* et finalement les tâches terminées *load_end*. Lors de chaque passage par une transition, l'indice complexe doit être modifié. Ici les événements observés ne prennent toujours pas en compte les événements nécessaires à l'ordonnancement de la tâche.

Considérons maintenant les événements de l'ordonnancement, nous allons décrire l'état de chaque tâche par rapport à l'événement observé. Pour chaque événement, une liste avec l'ensemble de tâches dans cet état de transition sera considérée.

5.3.1 Les listes de tâches par événement d'ordonnancement

.

Les listes constituent un ensemble d'outils largement utilisé dans le cadre de l'ordonnancement. Dans l'implémentation de l'ordonnanceur ce sont des classes d'objets C++ qui font partie intégrante des entités de base. Ce sont des archétypes *liste <X>* et on peut faire correspondre un véritable type au type *X* pour obtenir une *liste<entier>* ou une *liste<jobs>* ou une *liste <tâches>*.

Entre le moment où la tâche [5] est crée et le moment où elle débute son exécution, l'ordonnanceur intervient pour lui assigner un processeur. La tâche peut suivre différents chemins en fonction du mécanisme utilisé et de l'implémentation de l'ordonnanceur. Les tâches à ordonnancer seront placées dans une liste *Q* en attendant l'affectation d'un site d'exécution. En réalité, les diverses listes présentées sont quelques-unes virtuelles, et quelques-unes réelles. A un instant donné de l'exécution nous pourrons avoir les listes suivantes :

[3] Quand nous parlons d'exécution concurrente, c'est dans le sens de la présence simultanée des processus légers qui concourrent pour les ressources de calcul, ils sont deux à deux indépendants.

[4] *load_new* est un compteur des tâches créées dans le nœud, elles ne seront pas exécutées toutes localement, cet indice est indicatif de la quantité de charge générée par le nœud mais pas du travail que ce nœud a réalisé.

[5] Dans cette section nous utilisons l'abstraction générale d'une tâche sans la différencier de sa description *Job* utilisée dans l'implémentation

Q(new) : liste avec l'ensemble des tâches de création récente. Le site d'exécution n'est pas encore choisi.

Q(ord) : liste avec l'ensemble de tâches à ordonnancer. Le site d'exécution n'est pas encore choisi.

Q(map) : liste avec l'ensemble des tâches placées pour leur exécution sur le processeur propriétaire de la liste. Elles peuvent être réordonnancées.

Q(run) : liste avec l'ensemble des tâches en exécution. Ces tâches ne peuvent pas migrer.

Q(stop): liste avec l'ensemble des tâches bloquées en attente de communication ou de synchronisation. Ces tâches ne peuvent pas migrer.

Q(att) : liste avec l'ensemble des tâches débloquées, et en attente du *CPU*. Ces tâches ne peuvent pas être migrées.

Le parcours des tâches à travers les diverses listes est dessiné sur la figure 5.5. Les tâches qui se trouvent dans la *Q(ord)*, n'ont pas encore un site d'exécution désigné, dans la *Q(map)* sont déjà arrivées au processeur récepteur mais le placement peut être remis en cause (indiqué dans la figure par la transition 3). La transition 4 est le début d'exécution des tâches et donc les tâches dans les *Q(run)*, *Q(stop)*, *Q(att)* ne peuvent pas être migrées.

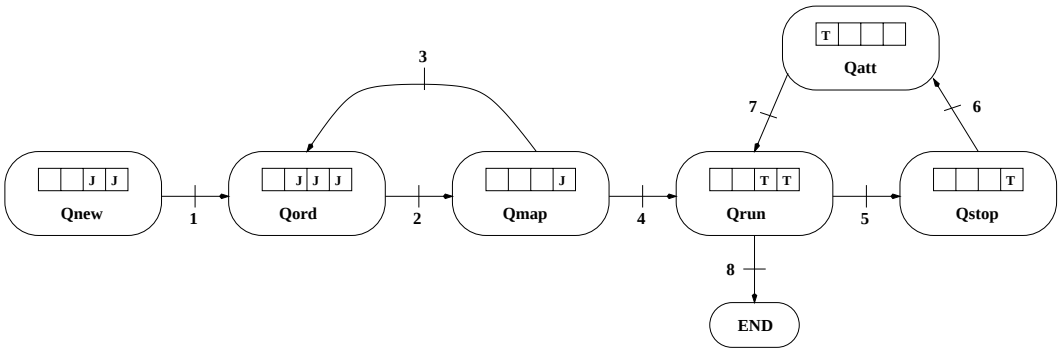


FIG. 5.5 – Parcours d'une tâche à travers les listes lors de son exécution.

Supposons qu'il soit possible d'observer l'ensemble des événements et qu'un ensemble de compteurs (voir tableau 5.5) soit établi. Un indice sera modifié à l'arrivée de l'événement. A un instant donné de l'exécution nous pourrons avoir différents indicateurs locaux. L'étude suivante est faite pour déterminer quelles observations sont pertinentes en vue d'établir un état de charge de chaque processeur et une vision globale de l'état du système.

5.3.2 Etude sur les indicateurs de charge

Nous considérons maintenant l'évaluation de la charge du système à partir des indicateurs spécifiques pour le modèle d'appel de procédure à distance. Nous allons présenter comment un collecteur central estime la charge de chaque processeur. Différents cas sont étudiés. Dans une implémentation centralisée, une seule liste *Q(ord)* existe et nous la

nommerons *Q(global)*. Elle contient toutes les tâches qui n'ont pas encore été ordonnancées.

Pour la charge locale un vecteur est utilisé pour maintenir les valeurs des indices de charge *loadlocal[événement]*, ces indices sont des compteurs d'événements (voir table 5.5). Le collecteur global a un ensemble de structures pour estimer la charge du système. $load\_map_{global}[proc]$ est un vecteur avec les décisions de placement faites par le contrôleur central (c'est un compteur du nombre des tâches placées dans chaque processeur par le contrôleur) et $load_{global}[proc]$ est un vecteur avec l'estimation de la charge de chaque processeur à partir des informations collectées des processeurs et l'information du vecteur $load\_map_{global}$.

Cas d'étude 1 : un indice simple

Le placement est contrôlé par un processeur central. À la création, toutes les *Jobs* sont envoyés au contrôleur, qui les garde dans la liste *Q(global)*. Le contrôleur va décider en fonction de sa " vision " de la charge des processeurs le site de placement pour les *Jobs*. Notons que cette vision n'est qu'une estimation, à cause des délais de transmission des messages qui actualisent les informations et l'avancement de l'exécution sur chaque processeur. En effet sur chaque processeur, des événements peuvent avoir modifié de façon importante la charge.

La vision du collecteur global sur la charge d'un processeur p est calculée par la formule :

$$load_{global}[p] = load\_map_{global}[p] - load\_end_p \tag{5.2}$$

Donc, l'information que chaque processeur p doit envoyer au contrôleur est l'indicateur du nombre de tâches finies $load\_end$ C'est un bon indicateur, simple, de la charge pour des applications où les tâches sont indépendantes et en grand nombre. Un exemple d'utilisation serait une élimination de Gauss ou plus généralement une application série-parallèle où le nombre de tâches est beaucoup plus grand que le nombre de processeurs.

Cependant dans le cas d'appel de procédure à distance, les tâches génératrices de parallélisme ou qui font appel à des nouvelles procédures seraient bloquées en attendant la fin des procédures appelées. Il peut se passer qu'à un instant donné, la machine semble chargée mais qu'en réalité toutes les tâches soient bloquées, et comme elles n'ont pas encore fini, la vision du collecteur reste que le processeur est chargé. On propose donc de mesurer plus d'événements, mais quels événements mesurer ? On peut l'analyser dans le cas suivant.

Cas d'étude 2 : un indice détaillé

Une tâche lorsqu'elle a débutée, peut être exécutée jusqu'à sa fin (sans être interrompue) si elle est une tâche de calcul, et peut être bloquée si elle est génératrice de parallélisme et doit attendre des résultats. Il faut noter que la récupération des résultats peut générer encore du calcul (comme dans le cas de la fusion de tableaux) et que ce coût

peut ne pas être négligeable. Dans l'implémentation actuelle ces tâches ne peuvent pas être migrées. Ces tâches qui restent bloquées, dans les circonstances décrites ci-dessous, ne sont pas une charge active à cet instant, mais représentent une charge future pour le processeur. Nous proposons de différencier ces deux charges du processeur : les tâches placées et pas encore débutées et les tâches qui ont déjà débuté mais qui ne sont pas encore finies. Les premières se trouvent dans la liste *Q(map)* locale au processeur. On peut calculer le nombre de tâches qui sont en exécution concurrente dans le processeur p (tâches en exécution, bloquées ou en attente de *CPU* après avoir été débloquées) :

$$load\_exe_p = load\_init_p - load\_end_p \tag{5.3}$$

L'indice $load\_exe$ ne différencie pas les tâches qui sont bloquées en communication ou synchronisation, des tâches en cours d'exécution ni de celles qui sont en attente de *CPU*. Le nombre de tâches qui ont été placées sur le processeur mais pas encore débutées est :

$$load\_map_p = load\_map_{global}[p] - load\_init_p \tag{5.4}$$

Alors l'estimation de la charge du processeur p est calculée par la formule :

$$load_{global}[p] = load\_exe_p + load\_map_p \tag{5.5}$$

Maintenant donnons des poids différents pour chaque type de charge α pour *load_exe* et β pour la charge dans la liste locale *load_map*. En considérant qu'une tâche qui vient d'être placée est effectivement un travail pour le processeur à cet instant on lui donne un poids différent de celui d'une tâche qui a été débutée mais qui est bloquée pour des besoins de communication ou de synchronisation. Pour cette tâche on ne sait pas à quel moment elle sera libérée. Elle est une charge future mais "moins" importante, puisqu'une partie de son travail a déjà été réalisée. On procède à la pondération de l'équation 5.5.

$$\begin{aligned} load_{global}[p] &= \alpha * load\_conc_p + \beta * load\_map_p \\ &= \alpha * (load\_init_p - load\_end_p) \\ &\quad + \beta * (load\_map_{global}[p] - load\_init_{p_i}) \end{aligned}$$

et on simplifie,

$$\begin{aligned} &= (\alpha - \beta) * load\_init_p - \alpha * load\_end_p \\ &\quad + \beta * load\_map_{global}[p] \end{aligned} \tag{5.6}$$

Donc chaque processeur devra envoyer au contrôleur le résultat de l'équation suivante :

$$(\alpha - \beta) * load\_init_p - \alpha * load\_end_p \tag{5.7}$$

Cette équation est l'indice de base complexe proposé par l'ordonnanceur. Si le programmeur veut utiliser l'indice du cas 1, il suffit de donner $\beta = \alpha = 1$ et on obtient l'équation 5.2.

Cas d'étude 3 : placement local au moment de la création

Considérons le cas où on donne un peu d'autonomie au processeur source de la tâche, qui peut décider de l'exécuter chez lui. Cette décision est prise au moment de la création : soit il la garde soit il l'envoie au contrôleur global. La politique accepte que le processeur source décide de garder un certain nombre de tâches qui sera inférieur à un seuil *MAXT*. L'exactitude de l'information peut s'avérer un peu plus compliquée à établir pour le collecteur central. Un autre indice sera ajouté : $load\_local$, qui représente le nombre de tâches créées et qui ont été gardées par la source pour leur exécution. Dans ce cas, le nombre de tâches qui ont été placées sur le processeur mais pas encore débutées est :

$$load\_map_p = load\_map_{global}[p] + load\_local_p - load\_init_p \tag{5.8}$$

Cette valeur pour $load\_map_p$ est substituée dans l'équation 5.5, on pondère comme pour le cas 2 et on simplifie, l'estimation de la charge par le collecteur central est calculée par :

$$\begin{aligned} load_{global}[p] = {} & (\alpha - \beta) * load\_init_p - \alpha * load\_end_p \\ & + \beta * load\_local_p \\ & + \beta * load\_map_{global}[p] \end{aligned} \tag{5.9}$$

Donc chaque processeur devra envoyer au contrôleur le résultat de l'équation suivante :

$$(\alpha - \beta) * load\_init_p - \alpha * load\_end_p + \beta * load\_local_p \tag{5.10}$$

Cas d'étude 4 : autonomie des processeurs pour exporter du travail

Dans ce cas nous allons donner plus d'autonomie aux processeurs. Les processeurs sont libres de décider d'exporter de travail. Les tâches qui sont placées et qui ne sont pas débutées tout de suite, restent dans la liste *Q(map)* et c'est de cette liste qu'elles seront prises pour être exécutées ou redistribuées. Pour simplifier, considérons que la décision d'exportation se fait seulement à partir des tâches dans la liste locale *Q(map)*. La nouvelle décision signifie qu'on peut remettre en cause le placement décidé et cette décision se fait de façon décentralisée, alors pour le collecteur, l'estimation de la charge devient plus complexe. Deux nouveaux indices sont définis : $load\_exp$ pour les tâches exportées et $load\_imp$ pour les tâches récupérées par ce mécanisme de redistribution.

Le nombre de tâches qui ont été placées dans le processeur, mais pas encore débutées est calculé par :

$$load\_map_p = load\_map_{global}[p] + load\_imp_p - load\_exp_p - load\_init_p \tag{5.11}$$

Cette valeur pour $load\_map_p$ est substituée dans l'équation 5.5, on pondère comme pour les autres cas les termes de l'équation et on simplifie, l'estimation de la charge par le collecteur central sera établie par l'équation :

$$\begin{aligned} load_{global}[p] = {} & (\alpha - \beta) * load\_init_p - \alpha * load\_end_p \\ & + \beta * (load\_local_p - load\_exp_p \\ & + load\_imp_p + load\_map_{global}[p]) \end{aligned} \tag{5.12}$$

Dans ce cas le processeur évalue cette équation et envoie cette valeur au collecteur central.

$$(\alpha - \beta) * load\_init_p - \alpha * load\_end_p + \beta * (load\_local_p - load\_exp_p + load\_imp_p) \tag{5.13}$$

Bilan

Au fur et à mesure qu'un nouvel événement implique un changement de la charge, il faut étudier si cet événement est pertinent pour le maintien de l'état de la charge et s'il doit être pris en compte - incrément ou décrément d'un compteur quant l'événement se produit -. Plus le nombre d'événements observés sera grand, plus l'estimation de la charge par le collecteur sera exacte . Plus on donne d'autonomie aux processeurs et plus l'estimation devient complexe. L'incrément des événements observés va générer un incrément dans les messages échangés dans un protocole basé sur l'envoi volontaire de la charge. Pour contrôler sa fréquence, un seuil est utilisé (*EVEN*).

5.4 Construction de la bibliothèque des algorithmes de régulation dynamique de charge

Nous présentons les conventions d'écriture des algorithmes de régulation dynamique, à travers un exemple d'implémentation centralisée. Pour le régulateur centralisé nous avons spécialisé les classes. Pour les décisions, c'est la classe auxiliaire au *JobManager*, la classe *Decision* qui est spécialisée. Il existe des méthodes qui correspondent à une décision locale et d'autres qui ont une influence globale. Dans le cas où nous allons considérer une liste centrale avec les *Jobs*, tout processeur envoie sa charge au contrôleur central qui a aussi la responsabilité d'être *LocalCollector*.

Tous les modules de l'ordonnanceur se trouvent présents dans chaque processeur. Les communications entre ces différents modules localisés dans le même processeur se font par appel des interfaces clairement définies. Un objet ne peut communiquer qu'avec un autre objet du même type qui se trouve dans un autre processeur si le protocole a été prévu. Seuls les *JobManager* et les *LoadCollector* ont des protocoles préétablis dans cette implémentation.

Après cette première phase, on procède à l'étude plus détaillée de chaque stratégie (sous-stratégie). On ne pourra pas faire les analyses de tous les détails, le nombre de combinaisons est combinatoire, mais on devrait pouvoir remarquer les paramètres qui sont les plus utilisés.

5.4.1 Élément de contrôle : la spécialisation du *JobManager*

Nous avons vu dans la section 3.2 que l'élément de contrôle du point du vue fonctionnel est composé de la politique de sélection et de la politique de détermination locale de la charge. Ces deux composantes sont bien distinctes dans l'implémentation proposée.

La politique de détermination locale de la charge

La politique de détermination locale de la charge est répartie. Sur chaque processeur, un élément de décision local décide si une tâche créée doit être exécutée localement ou si elle doit être exportée sur un autre processeur (dans notre cas vers le processeur centralisé). Dans l'implémentation, la décision est prise en fonction de la charge locale du processeur. Les éléments locaux de décision utilisent les informations du *LocalLoad*. Si la charge locale dépasse un certain seuil *MAXT* alors l'élément de contrôle décide d'exporter les tâches créées. Le seuil local est fixé au début du programme et ne change pas durant toute l'exécution.

La politique de sélection

La politique de sélection peut-être centralisée. Dans ce cas, seule la sélection du processeur récepteur doit être réalisée. A partir des informations collectées, le *JobManager* décide du processeur vers lequel la tache sera envoyée.

Dans un cas décentralisé, après que la charge locale à exporter ait été déterminée, c'est à la méthode de sélection locale du processeur source de choisir le processeur vers lequel la tâche doit être exportée. L'élément de contrôle choisit le processeur le moins chargé. Celui-ci étant le processeur dont l'indice global de charge est le plus faible. Un problème se pose lorsque plusieurs processeurs sont de charge égale. Si nous choisissons un processeur et que nous lui affectons une tâche, lors de la demande suivante de choix de processeur, l'état de charge du système a de très fortes chances d'être le même, car l'écart de temps qui sépare deux demandes d'exécution est en général plus court que le temps de transfert d'une tâche d'un processeur vers un autre, plus le temps de communication de la nouvelle charge du processeur vers l'élément d'information. Donc nous risquons de choisir le même processeur. Pour palier ce problème nous avons décidé dans un premier temps de choisir aléatoirement un processeur parmi les processeurs dont le niveau de charge est le plus faible. La seconde solution consiste en la création d'un historique des décisions de placement déjà réalisées afin de corriger la vision locale de la charge du système en fonction de ces décisions.

5.4.2 L'élément d'information : la spécialisation du *LoadCollector*

L'élément d'information est composé d'un élément d'information central qui maintient une vision globale du système grâce à un tableau et d'éléments d'information locaux qui sont chargés d'évaluer la charge locale du processeur sur lequel ils se trouvent et de la transmettre à l'élément de charge globale (voir section 5.3.2).

LocalLoad

A chaque changement de charge locale (placement d'une tâche, terminaison d'une tâche, mise en attente d'entrée sortie d'une tâche, réactivation d'une tâche bloquée), un objet de description de charge locale est mis à jour. Ensuite l'élément local d'information décide, selon la condition choisie si l'élément central d'information doit être informé de ce changement local de charge. La politique que nous avons retenue pour l'instant est une politique volontaire. Si la charge a varié d'un certain écart (*EVEN*), alors le nouvel état de charge est communiqué.

L'élément d'information global

L'objet en charge de gérer l'information globale est centralisé. Il centralise les messages d'information venant des différents processeurs. Il maintient à jour l'état du système basé sur l'état de charge des processeurs. Cet objet permet de faire des opérations de tri des processeurs en fonction de la charge et de faire une estimation de la charge future du système par rapport aux choix de placement de nouvelles taches.

5.5 Conclusion

L'ordonnanceur décrit dans ce chapitre peut être vu comme une interface virtuelle pour l'implémentation de nouveaux algorithmes d'équilibrage de charge. Il propose un support pour les fonctionnalités de contrôle et d'estimation de la charge des heuristiques dynamiques : le *JobManager* et le *LoadCollecteur* - *LocalLoad*. Il a été interfacé aux deux couches du système ATHAPASCAN, grâce à la présence de deux modules d'interface : le *JobBuilder* et le *MakeRunDaemon*. Cette implémentation permet de séparer le modèle de programmation et la machine parallèle cible du mécanisme de régulation.

Dans cette structure, des paramètres utilisés pour adapter ou contrôler la stratégie de régulation, ont été spécifiés. Ces paramètres sont : *MAXT, MINT, JOBT, EXPJ, IMPJ, MRUN* et *EVEN*. Dans la partie suivante du document, on étudiera l'influence de ces paramètres dans les différentes stratégies d'équilibrage de charge.

Troisième partie

Évaluation des performances

Chapitre 6

Planification expérimentale

Dans ce chapitre nous décrivons la planification de nos expériences. La planification est primordiale dans notre méthodologie pour l'évaluation des stratégies de régulation dynamique de charge. Notre approche est l'évaluation des performances par des mesures effectuées sur une machine parallèle réelle. On verra en détail les programmes parallèles synthétiques utilisés comme charge applicative. Nous décrivons la plate-forme expérimentale et les différents algorithmes qui sont évalués.

Dans l'informatique comme dans un grand nombre de domaines, la complexité du comportement réel du système à étudier et à utiliser est telle qu'il n'est pas possible de prendre en considération l'ensemble des paramètres qui le caractérisent. L'évaluation de performances, qui selon Jain [Jai91] est un art, recoupe une grande diversité de sujets d'études et de méthodes. En effet le but de toute évaluation est de déterminer les qualités et les défauts d'un système, afin de dégager la solution la plus performante et la moins coûteuse.

Une fois les stratégies d'ordonnancement choisies et implantées, il convient maintenant de réaliser l'évaluation de leurs performances.

À partir de l'étude qualitative des algorithmes de régulation de charge et de l'implémentation dans un système comme ATHAPASCAN1, on a déterminé un groupe important de paramètres qui " semblent " avoir une influence sur l'exécution d'une application parallèle. Ces paramètres (que nous nommerons variables ou facteurs [1]), qui sont déterminés avant l'exécution d'une application ATHAPASCAN1 et au moment de la compilation (pour le choix de l'ordonnanceur) dépendent de l'application parallèle, de l'algorithme de répartition dynamique choisi et de la machine parallèle pour le nombre de processeurs. Le nombre de combinaisons de paramètres est très grand. L'évaluation de performance par prise de mesures est un problème délicat et demande la mise au point de plans d'expériences rigoureux. Sans méthodologie expérimentale, la validité des résultats est sujette à caution [Jac96b].

Nous présentons une étude systématique de performances où nous faisons varier différents paramètres en même temps. Le peu de publications dans le domaine prouve le

[1] En statistique, on appelle facteur un paramètre qui peut prendre différentes valeurs.

caractère singulier et délicat d'un tel travail expérimental.

La première partie de ce chapitre présente les méthodes classiques d'évaluation de performances et justifie le choix de baser l'étude sur des mesures d'exécutions réelles. Une telle étude nécessite la constitution d'un jeu d'essais. Nous utilisons une charge synthétique dynamique [KP94], où les applications vont simuler le comportement d'une application parallèle réelle, e.g. vont consommer des ressources (calcul, communication) mais ne font aucun calcul réel. Notre modèle est base sur le modèle ANDES [Kit94]. Nous avons évalué différentes stratégies d'équilibrage de charge, lesquelles restent dans le cadre d'un ordonnanceur centralisé. Pour notre planification expérimentale nous nous sommes basés sur les plans factoriels complets.

6.1 Méthodologie pour l'évaluation des stratégies de régulation dynamique

6.1.1 Les techniques d'évaluation de performances

Il existe différentes techniques d'évaluation de performances. Parmi elles, on peut citer la modélisation analytique, la simulation, et les mesures obtenues à partir d'un système réel [Jai91]. Le but de toute évaluation est de déterminer les qualités et les défauts du système. Cela concerne donc les domaines où on recherche la meilleure performance pour un coût minimum, en particulier les domaines de développement d'algorithmes et de systèmes informatiques. Par contre chaque problème d'évaluation est unique et nécessite l'élaboration d'une métrique, d'une modélisation du système et de techniques d'évaluation propres à ce système donné.

La modélisation analytique

La modélisation analytique consiste à représenter les conditions réelles de façon formelle à l'aide d'outils mathématiques. Différentes théories mathématiques sont employées : la théorie des réseaux de Petri [Fri89], les réseaux d'automates, la théorie des files d'attente [All80], la théorie des probabilités [Jug96]. On travaille sur un système d'équations définissant le système. Ces équations analytiques contiennent le critère de performance exprimé en fonction des paramètres du système.

A partir de ce modèle du système, la modélisation tente de trouver la valeur du critère d'évaluation choisi. Par exemple l'utilisation de chaînes de Markov résout les systèmes de files d'attente. Le résultat est généralement trouvé dans un temps raisonnable. La modélisation a une description peu coûteuse, mais le modèle peut quant à lui être très coûteux en temps de mise au point. Une critique de ce type d'analyse est la précision sur les résultats. En effet la résolution se fait à un niveau d'abstraction assez élevé ne permettent pas toujours de conclure sans précaution. Plus le modèle est proche de la réalité, plus complexe est la modélisation, et plus précis soient les indices obtenus.

La simulation

La simulation consiste à modéliser la globalité du système étudié et à le simuler numériquement à l'aide d'événements provenant de mesures sur un système réel ou de modèles probabilistes. L'intérêt de la simulation est de pouvoir travailler sur des systèmes non disponibles. Par exemple lors d'étapes de conception, il est beaucoup moins coûteux de réaliser une simulation préalable des alternatives envisagées. De plus, la simulation est un moyen très souple d'étudier un problème. Cette technique permet des réexécutions de programme avec changement de paramètres et une prise de trace d'exécution sans perturbation. Enfin elle permet, puisque le temps est modélisé, d'arrêter la simulation pour revenir à un état antérieur.

Il existe des simulations dans de nombreux domaines. La simulation est couramment utilisée pour la validation de résultats issus de modèles analytiques. Malheureusement, la simulation ne réussit pas toujours à tout modéliser à cause de la complexité du système. Elle prend de plus beaucoup de temps. Ceci peut venir du niveau de détail demandé et de bien d'autres paramètres. La qualité des simulations est en rapport direct avec la qualité de la modélisation du système.

Ces deux premières méthodes ne permettent pas de prendre en compte finement les conditions d'exécution comme la surcharge due au système d'exploitation, ainsi qu'aux utilisateurs (si le système est multi-utilisateurs). En effet il est très difficile de créer un modèle mathématique précis ou de recréer une simulation prenant en compte le comportement d'utilisateurs ou du système d'exploitation.

Les mesures

On peut enfin évaluer le système à partir de mesures réelles. Cette approche nécessite d'une machine parallèle réelle. A partir de l'exécution de programmes (réels ou synthétiques), on fait des mesures sur le système. Il n'est donc plus question du modèle. Cette technique permet de prendre en compte la surcharge («overhead») imposé par le système d'exploitation, ainsi qu'aux utilisateurs (si le système est multi-utilisateurs) et d'étudier le comportement dynamique des applications irrégulières. Si on gagne en réalisme par rapport au système, on est confronté à l'incertitude sur les mesures. Il convient alors de faire une série d'expériences et une analyse statistique des résultats pour déterminer une mesure et son intervalle de confiance. L'évaluation de performance par prise de mesures est un problème délicat qui demande la mise au point de plans d'expériences rigoureux.

L'instabilité et la difficulté de prédiction des performances applicatives résultent des interactions complexes et généralement non-déterministes entre le logiciel applicatif, le système d'exploitation (incluant la gestion des ressources de communication) et le matériel [Ree93]. Nous considérons que l'interprétation d'un système dans son contexte réel, bien que l'analyse soit difficile, est nécessaire et importante pour le développement des applications parallèles.

6.1.2 Problématique des mesures dans un système parallèle dynamique

La motivation pour le développement d'une application scientifique parallèle ou distribuée est la vitesse d'exécution, mais les performances applicatives, comme la vitesse d'exécution ou le taux d'accélération sont en général instables sur une architecture parallèle [Mai96]. On est en général incapable de prédire les performances d'une application donnée sur une architecture parallèle nouvelle [Ree93]. Dans le domaine des machines parallèles, le nombre de facteurs qui jouent sur la performance est encore bien plus élevé que sur les machines séquentielles.

Nous avons choisi d'utiliser la technique des mesures sur un système réel avec des bancs d'essais orientés application (**applications synthétiques**). La première raison qui motive ce choix est que nous disposons d'une machine parallèle pour exécuter nos programmes (IBM SP2 à 32 processeurs). La seconde est que nous travaillons au développement d'un système pour les applications parallèles [2] et c'est sur cette plate-forme que nous avons fait nos mesures.

Nous avons évalué différentes stratégies d'équilibrage dynamique de charge pour les applications parallèles. Les surcharges induites par le système et les utilisateurs sont donc importantes. En effet leur existence permet à un régulateur dynamique de se démarquer fortement d'un régulateur statique lorsque cette surcharge devient importante. Pour réaliser ce travail nous avons suivi une méthode de planification expérimentale décrite dans les sections suivantes. L'analyses des performances est présenté dans le chapitre suivant.

6.1.3 Méthodologie suivie

Les éléments les plus importants de notre planification sont [Dag96]:

but (objectif) Le ou les buts poursuivis peuvent être extrêmement variés, et doivent toujours être définis de façon très précise. Les autres éléments du protocole expérimental découlent en effet très largement des objectifs à atteindre.

conditions de réalisation description de la plate-forme expérimentale, la machine parallèle, les outils pour la prise des mesures, la description de la charge synthétique induite.

facteurs (qualitatif et quantitatif), un facteur est **qualitatif** quand les différents éléments qui le compose ne peuvent pas être mesurés par une échelle numérique, nous les appellerons facteurs qualitatifs (ou paramètres structuraux [Kit94]). Par exemple pour décrire les applications nous avons deux types de structure, arborescente ou série-parallèle. Un facteur est au contraire **quantitatif** quand ses éléments se classent tout naturellement selon une métrique, par exemple nombre de processeurs ou le nombre de tâches d'une application.

modalités les différentes valeurs qu'un facteur peut prendre : valeur pour un facteur quantitatif et niveau pour un facteur qualitatif. Par exemple le seuil local (pour

[2] ATHAPASCANvoir chapitre 4.

décider si une tâche créée localement est exécutée localement ou exportée), peut prendre les valeurs 0, 1, 2.

objets combinaison des modalités des différents facteurs pour une expérience donnée. Il peut s'agir de fixer un nombre de facteurs ou le fait que certains facteurs ne vont pas participer.

observations quelles sont les observations qui sont considérées ? Dans notre cas, principalement le temps d'exécution parallèle (makespan est pris en compte). Un problème important est la forme de stockage des mesures.

planification des expériences combien de mesures seront effectue en considérant les limites en temps pratiques.

analyse des résultats que faire avec les mesures récupérées, ce point sera développé dans le chapitre suivant.

6.1.4 Outils pour la planification expérimentale : les plans

Quel est l'objectif de l'expérience ? La réponse va guider la construction d'un plan expérimental. Concevoir un plan d'expérience revient à sélectionner les essais (expériences) que l'on va faire au moyen du système ainsi que l'ordre dans lequel ces expériences seront réalisées.

On s'intéresse aux performances des différentes stratégies de répartition dynamique de charge sur un jeu d'application. Dans notre problème nous avons différents " facteurs globaux " qui sont les différents types d'algorithmes de régulation dynamique de charge, les différents paramètres de ces algorithmes et les différents programmes applicatifs. Nous avons un nombre très élevé de combinaisons et nous nous demandons comment on peut obtenir des conclusions scientifiquement correctes ?

Pour éviter de faire une démarche par tâtonnement qui prendra trop de temps pour arriver à des conclusions valides, ou de faire varier un seul paramètre de notre système complexe ce qui nous donnera une vision partiale du phénomène, nous faisons appel aux plans factoriels.

Les plans factoriels

Nous présenterons ici les plans factoriels complets et fractionnaires. D'autres plans expérimentaux existent plus économiques [Dag96, Mon91].

- **le plan factoriel complet** relatif à un ensemble de deux ou plusieurs facteurs est tel que chacune des modalités de chacun des facteurs est associée à chacune des modalités de l'autre ou de chacun des autres facteurs. Ce type de plans permet de mettre en évidence tous les effets des facteurs et leurs interactions [Mon91]. Mais prendre toutes les combinaisons a un coût trop élevé, et pour diminuer ce coût on utilise un plan factoriel fractionnaire.
- **les plans factoriels fractionnaires** sont constitués de sous-ensembles, judicieusement choisis, d'objets de plans factoriels complets [Dag96]. Pour les plans 2^h, il s'agit en général de la moitié ou du quart des objets du plan complet, tandis que pour les plans 3^h, il s'agit le plus souvent du tiers ou du neuvième du plan complet.

Pour avoir une réponse concernant les interactions entre facteurs, un plan factoriel fractionnaire à deux niveaux n'est pas suffisant et il est nécessaire de recourir aux plans à trois niveaux, qui normalement incluent toutes les combinaisons des valeurs maximales, minimales et moyennes des facteurs. Ces plans permettent de mettre en évidence tous les effets principaux des facteurs et leurs interactions. Ils sont de toute façon assez coûteux.

Exemple : On a deux facteurs A et B (voir tableau 6.1), chacun avec différents niveaux ou modalités. On dénote a le nombre de niveaux de A et b pour les différents niveaux de B. Y_{ijk} représente la réponse quand le facteur A a le niveau i (pour $i = 1, 2$, ..., a) et quand le facteur B a le niveau j (pour $j = 1, 2, \ldots,$ b) dans la k ème répétition (pour $k = 1, 2, \ldots,$ n). Dans le cas d'un plan factoriel complet, le nombre des observations est $a * b * n$. Dans le cas d'un plan fractionnaire du type 2^h on aura seulement deux modalités par facteur (h est le nombre de facteurs, dans l'exemple 2). Ce nombre sera $2 * 2 * n$ et dans le cas du type 3^h on aura $3 * 3 * n$ observations.

		fac. B			
		1	2	…	b
fac. A	1	$Y_{111}, Y_{112}, \ldots, Y_{11n}$	$Y_{121}, Y_{122}, \ldots, Y_{12n}$		$Y_{1b1}, Y_{1b1}, \ldots, Y_{1bn}$
	2	$Y_{211}, Y_{212}, \ldots, Y_{21n}$	$Y_{221}, Y_{221}, \ldots, Y_{22n}$		$Y_{2b1}, Y_{2b1}, \ldots, Y_{2b1}$
	⋮				
	a	$Y_{a11}, Y_{a12}, \ldots, Y_{a1n}$	$Y_{a21}, Y_{a21}, \ldots, Y_{a21}$		$Y_{ab1}, Y_{ab1}, \ldots, Y_{abn}$

TAB. 6.1 – Exemple pour une expérience à deux facteurs

Quand les observations ont été réalisées la forme de leur interprétation est importante. Il est possible d'utiliser différents techniques dépendant de l'objectif cherché ou du type de données utilisées. Parmi ces techniques, on trouve l'analyse de variance (ANOVA [Jai91]), l'analyse de covariance, la régression multiple (voir section 7.1.2) et la technique d'analyse en composantes principales (voir section 7.1.1).

6.2 Nos objectifs

Il est difficile d'évaluer et comparer des algorithmes d'équilibrage dynamique de charge [Jac96b],[Kit94],[FZ87]. Comme objectif général nous voulons, étant donné un algorithme d'équilibrage classique : aléatoire, cyclique, module, glouton, par seuils, …ou un nouvel algorithme implementé dans une application spécifique, avoir la possibilité d'étudier son comportement dans différentes situations afin de le caractériser et de le comparer à d'autres stratégies. Cet objectif général inclut différents objectifs aussi importants que nécessaires que nous décrivons ci-dessus :

Obj. 1 Nous voulons comprendre le comportement des algorithmes dynamiques de régulation de charge et déterminer les éléments (paramètres) qui ont une influence sur les performances obtenues ainsi que les éventuelles interactions (nommée synergies) entre ces éléments.

Obj. 2 Nous voulons déterminer quelles valeurs donner aux paramètres d'une stratégie d'équilibrage de charge pour lui assurer un " bon rendement ". Ces valeurs doivent elle rester fixes, être adaptatives ?

Obj. 3 Et finalement nous aimerions évaluer si deux stratégies dynamiques d'équilibrage de charge sont significativement différentes.

Par exemple, on peut se demander si pour l'ordonnancement le rapport entre les calculs et les communications (nommé granularité) de l'application est un paramètre influent sur leur comportement.

6.3 Les programmes parallèles synthétiques

Les techniques d'évaluation pour la mesure de performances ont besoin de jeux d'essais (*benchmarks*). Deux approches se dégagent aujourd'hui : les **bancs d'essais fins** et les **bancs d'essais orientés application**. Le but des jeux d'essais est de fournir une évaluation quantitative de la puissance d'une machine ou du système logiciel étudié. Ils évoluent vers de véritables collections d'applications d'un grand nombre de domaines du calcul scientifique et de l'ingénierie [BBBa94].

Les **bancs d'essais fins** sont destinés aux opérations de bas niveau : commutation des tâches, gestion des communications, temps de préemption de tâches. L'objectif est ici d'isoler une caractéristique du système et de la mesurer. Par exemple, on les utilise pour connaître la bande passante effective de l'ordinateur parallèle en utilisant différentes bibliothèques de communications. Ce type de banc d'essais a aussi été utilisé pour comparer les performances d'ATHAPASCAN0 par rapport à la bibliothèque de base (PVM ou MPI) [Gin97, CRCG95]. Dans la section 6.3.1, un programme *ping-pong* a été utilisé pour mesurer le temps d'envoi et retour d'un message entre deux processeurs, en faisant varier la taille des messages. Les principaux inconvénients de ces mesures résident dans la difficulté d'isoler la caractéristique à mesurer sachant que l'ordre de grandeur de celle-ci est souvent de la micro-se conde. Les horloges logicielles ne possédant pas la précision requise, l'opération doit être répétée un grand nombre de fois de façon à obtenir un agrégat de mesures suffisant [Ste97].

Dans les **bancs d'essais orientés application**, on trouve des programmes réels dérivés principalement du calcul scientifique et aussi des bancs d'essais synthétiques. Le but des ces bancs d'essais est de mettre en oeuvre un ensemble d'activités concurrentes. Pour les applications synthétiques les résultats produits se situent dans le contexte du système réel en jouant auprès de lui le rôle d'une vraie application.

Le modèle de programme pour la charge synthétique que nous avons implementé se base sur le modèle *ANDES* (Algorithms aNd DEScription) décrit dans [Kit94]. Dans ce modèle, les applications parallèles sont représentées par un graphe de précédence orienté sans cycle (DAG) où les sommets représentent les tâches de calcul et sont pondérés par un coût de calcul. Les arcs représentent les précédences entre ces noeuds de calculs et sont pondérés par un coût de communication. En résumé, ce modèle de programme est une description quantitative d'applications parallèles et décrit principalement le découpage de l'application en tâches, définit le nombre d'instructions (opérations de calcul) et

les volumes de communication. Dans ce cadre une charge synthétique est un modèle de programme qui imite le comportement d'un programme parallèle réel du point de vue de la consommation des ressources de la machine (utilisation des processeurs, de mémoire et des liens de communication), mais qui ne calcule aucun résultat.

Dans ATHAPASCAN1, la programmation d'une application parallèle consiste en une description de son DAG et des calculs réalisés par les tâches, basée sur l'appel de procédure à distance. Notons que pour le type de phénomène que nous voulons étudier (l'équilibrage de charge dynamique), il faut que les applications synthétiques possèdent un caractère d'irrégularité et d'aléatoire. Nous avons créé deux modèles (classe base $C++$) de programmes parallèles synthétiques dynamiques et paramètrables (voir fig. 6.1) dont la structure a été inspirée des programmes parallèles réels et de modèles ANDES. Ce sont les graphes de type diviser-pour-paralléliser et les graphes série-parallèle.

La régulation dynamique de charge est un problème très complexe. L'utilisation de programmes synthétiques nous permet d'élaborer un jeu de tests rapidement et efficacement. Maintenant on va à décrire en détail les deux modèles de graphes synthétiques utilisés dans notre expérience.

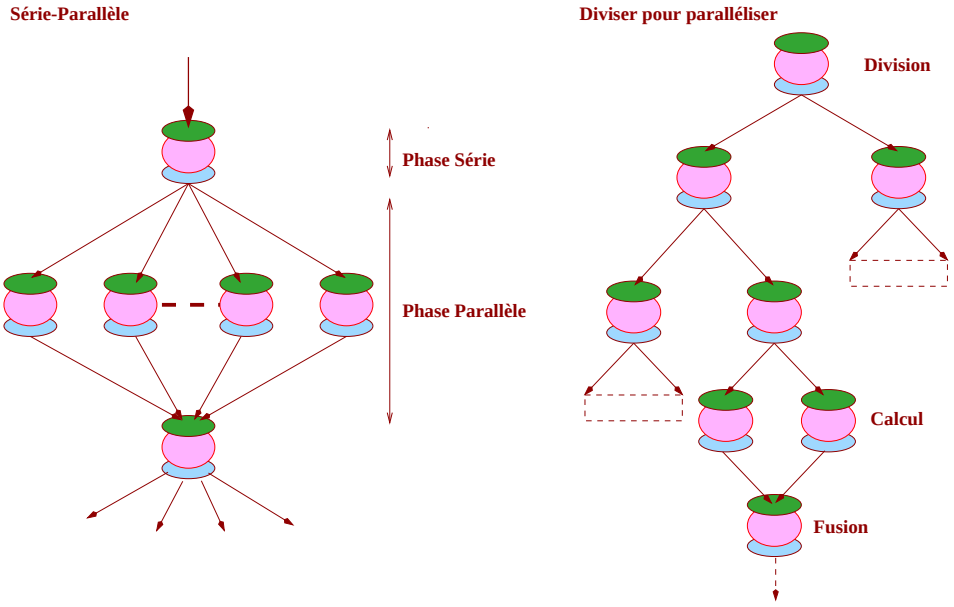


FIG. 6.1 – Les modèles

6.3.1 Modèle Série-Parallèle

Les algorithmes de type série-parallèle sont composé d'une série de phases parallèles commencée par une tâche de division et terminée par une tâche de synchronisation. La figure 6.1 montre une portion d'un tel algorithme. On compte parmi ces algorithmes les algorithmes de calcul de point fixe du type Jacobi ou l'algorithme de Gauss avec pivot total et bien d'autre encore. Dans l'algorithme de recherche de point fixe, une première

tâche lance en parallèle l'exécution d'un calcul sur une matrice (une multiplication matrice vecteur). Puis à la fin du calcul parallèle, les tâches se synchronisent. Si le point fixe est atteint, l'algorithme s'arrête sinon une nouvelle phase parallèle commence.

Dans notre modèle un graphe série parallèle se définit par:

NbIter	Le nombre d'itérations ou de phases parallèles que contient le graphe. Par exemple pour modéliser un algorithme de Gauss avec pivot total, le nombre d'itérations est égal au nombre de lignes contenues dans la matrice.
NbSon	Le nombre de tâches créées dans la partie parallèle du graphe.
VarIter	L'évolution du nombre de tâches dans la partie parallèle. En effet, ce nombre peut varier au fur et à mesure du déroulement de l'algorithme. Dans l'algorithme de Gauss, le nombre de tâches créées dans la partie parallèle décroît de 1 à chaque itération. Si VarIter = 0, le nombre reste constant entre chacune des itérations, si VarIter = 1 alors il décroît de 1.
DATA	Taille des données d'entrée.
CCD	Les coûts de calcul dus à la séparation des données dans la phase série.
CCT	Les coûts de calcul des opérations parallèles.
CCF	Les coûts de calcul de la fusion des données dans la partie de synchronisation.
SDAT	Les coûts de communication de données.
BACK	Les coûts de communication des résultats.

6.3.2 Modèle Diviser-pour-Paralléliser

Les programmes parallèles à structure d'arbre sont nombreux [MS91, Kit94]. Parmi eux, on trouve les algorithmes de somme d'éléments d'un vecteur, de produit scalaire, de tri d'un vecteur, de multiplication de matrices, etc. Ces algorithmes sont basés sur l'utilisation récursive d'une 'stratégie diviser pour paralléliser '. Ces algorithmes sont typiquement composés de trois parties, voir figure 6.1. Une phase récursive de division ou partitionnement, durant laquelle les données sont divisées, une phase de calcul réalisée sur les données, et une phase de fusion des résultats calculés dans les étapes précédentes. L'algorithme du tri rapide est un bon exemple d'un tel algorithme : il consiste à diviser en forme récursive un tableau tant que le nombre d'élément est supérieur à un seuil, puis la phase de calcul qui consiste en un tri séquentiel du tableau et une phase de fusion triviale.

Dans notre modèle, un arbre est défini par:

NbSons	Le nombre de fils créés à chaque étape de la division.
Depth	La profondeur maximum du graphe.
DATA	Taille des données en entrée.
CCD	Le coût de calcul dû à la séparation des données.
CCT	Le coût de calcul dû au calcul proprement dit sur les feuilles (le tri séquentiel).
CCF	Le coût de calcul dû à la fusion des données (la fusion de listes triées).
SDAT	Le coût de communication dû à l'envoi des paramètres.
BACK	Le coût de communication dû aux retours des résultats.
STOP	Méthode d'arrêt.

Les conditions d'arrêt des divisions sont deux, la division ne s'arrête que lorsque la profondeur maximale de l'arbre est atteinte ou lorsque la taille des données devient plus petite qu'un seuil arrêt de la découpe parallèle due à la taille du grain).

6.3.3 Les coûts de calcul et de communication

Les méthodes qui vont décrire les coûts sont basés sur les poly-algorithmes [3]. Décrivent deux paramètres, le premier pour identifier la méthode pour calculer le coût et un deuxième comme paramètre auxiliaire. Exemple *CCTask* est le paramètre pour choisir la méthode pour calculer le coût de la tache et *CCTaskAux* est le paramètre auxiliaire. Le nombre d'opérations de calcul à faire pour chaque tâche et le nombre de données communiquées par la tâche peuvent être aléatoire ou dépendantes. Les valeurs aléatoires suivent une loi uniforme. Les valeurs dépendantes seront une fonction des paramètres que reçoit en entrée la tâche, normalement de la valeur de $DATA$ et du paramètre auxiliaire. Notons que $DATA$ est un paramètre qui évolue avec l'exécution de l'application. La tâche racine de l'application reçoit en entrée une valeur $DATA$ donné par l'utilisateur, puis en fonction de la répartition des données, chacune de ses tâches filles peut avoir une valeur différente comme entrée pour ce paramètre (si la méthode utilisée est une fonction aléatoire).

Les coûts de calcul

Les coûts de calcul modélisent un calcul en fonction de la taille des entrées. Ce coût peut être constant, linéaire, polynomial ou aléatoire en fonction de cette taille. Si n est la taille des données en entrée de la tâche et si x est le paramètre du coût de calcul, alors le coût de calcul de la tâche peut être de O(x), O($x * n$), O(n^x), ou O(normal()) opérations entières.

[3] voir chapitre 5

Exemple pour le coût de calcul d'une tâche du modèle Serie-Parallèle :

```
int GraphSP::cost_task(){
int lim ;
switch(CCTask){
        // constant
        case 0 : lim = CCTaskAux ; break ;
        // linaire
        case 1 : lim = DimData * CCTaskAux ; break ;
        // polinomial
        case 2 : lim = pow(DimData, CCTaskAux) ; break ;
        // aleatoire, aprox aux valeur de DATA
        case 3 : lim = DimData(ale() - ale())
                       * DimData*CCTaskAux/100 ; break; }
return lim ; }
```

Communications : envoi des paramètres

Les coûts de communication pour l'envoi des paramètres sont fonction de la taille des entrées d'une tâche. Dans le cas d'un tri rapide le choix du pivot fait que le tableau est scindé en deux parties inégales en général. C'est pour représenter de telles classes de programmes que nous avons introduit la notion de partage aléatoire des données. Les données sont partagées selon une loi uniforme. Celle-ci peut partager ses données entre ses fils selon les méthodes suivantes:

1. Communication de l'ensemble des données.
2. Partage équitable entre les fils.
3. Partage aléatoire entre les fils.
4. Taille constante pendant tout le programme donnée par $SendParamAux$.

Exemple pour l'envoi des données dans un modèle SP :

```
int GraphSP ::dim_data(){
int taille ;
switch(SendParam){
        // l'envoi de l'ensemble des données, constant
        case 0 : taille = DimData ;
                 break ;
        // équitablement répartie
        case 1 : taille = DimData / NbSon ;
                 break ;
        // aleatoire, aprox aux valeur de DimData
        case 2 : taille = DimData/NbSon
```

```
                + (ale()-ale())* DimData*SendParamAux/100 ;
                  break ;
         // constant
         case 3 : taille = SendParamAux ;
                  break ;
}
return taille ;
}
```

Communications : retour des résultats

Les coûts de communication pour le retour des résultats sont aussi de divers types:

1. Accumulatif, ou fur et à mesure qui l'on parcourt le graphe des tâches terminales vers les tâches initiales, on récupère les résultats.
2. Constant. Le résultat du calcul de la tâche est de longueur constante et ne dépend pas de la taille des données. La constante est donnée par $BackResultAux$.
3. De même longueur que les données reçues.
4. La tâche effectue une transformation des données et en renvoie la nouvelle valeur du coût.

Exemple pour le retour des résultats :

```
int GraphSP ::dim_result(){
int taille ;
switch(BackResult){
         // accumulatif
         case 0 : taille = cumm + ParamResult ;
                  break ;
   // constante
         case 1 : taille = ParamResult ;
                  break ;
         // constante
         case 2 : taille = DimData ;
                  break
         case 3 : taille = DimData/ParamResult ;
                  break ;
         case 4 : taille = DimData*ParamResult ;
                  break ;
         }
return taille ;
}
```

Il est très facile de modifier ou d'incrémenter une nouvelle méthode pour le comportement du graphe autant pour le coût de calcul que pour le coût de communication.

6.4 Conditions de réalisation : la plate-forme expérimentale

Nous allons dans cette section décrire l'environnement dans lequel s'est déroulée l'expérimentation. Notamment nous allons modéliser la machine parallèle sur laquelle nous avons travaillé, en donnant un modèle pour l'architecture de la machine, pour ses communications et ses calculs. Ceci nous permettra d'établir pour chaque programme synthétique les temps de calcul et de communication dans une situation réaliste.

6.4.1 La machine parallèle

Le modèle de machine que nous avons utilisé est le plus proche possible de la machine parallèle physique sur laquelle se sont déroulées nos expérimentations. Cette machine est un IBM SP composée de 32 noeuds de calcul RS6000 avec 1Go de disque et 64Mo de RAM par noeud. Elle a réseau ethernet à 10Mbits/s et un reseau *Hight Performance Switch*(HPS). La topologie de cette machine est telle que chaque noeud est à égale distance de tous les autres noeuds. C'est pourquoi le modèle de machine que nous utilisons est une machine parallèle à mémoire distribuée composée de 32 noeuds de calcul interconnectés par deux réseaux de communication. Cependant comme c'est une machine d'expérimentation pour tout le laboratoire il n'était pas possible d'utiliser tous les nœuds et seulement une partie à été dédiée, soit 8 processeurs de calcul et un routeur.

Le modèle de communication

Pour définir le modèle quantitatif de communication nous avons réalisé une étude statistique de mesures. Nous avons réalisé un programme d'échange de messages (ping-pong) entre deux noeuds de la machine pour différentes tailles de messages. Nous avons réalisé cette étude sur ATHAPASCAN0b pour prendre le surcoût du au système [4]et cela sur le réseau HPS. La figure 6.2 représente la courbe de communication. Nous avons modélise les communications entre deux noeuds de la machine par une droite, qui a été calculé avec un programme de régression linéaire [Jai91].

Entre deux noeuds de calculs le temps de communication de n octets est de (en microsecondes):

– $t_{comm}(\mu) = 1391.197 + (0.03582428 * n_{octets})$

Dans le modèle, le coût de communication est une droite de la forme $ax + b = 0$. Dans cette équation b est le coût initial de la communication. Ce coût est dû à l'établissement de la communication. On remarque que ce coût de mise en oeuvre de la communication est bien supérieur au coût de transfert d'un octet. Ainsi le coût de transfert de 38833 octets est égal au coût de mise en oeuvre de la communication. C'est pourquoi par la suite nous ne prendrons en compte que le coût de démarrage d'une communication pour choisir la granularité minimum des programmes.

[4]L'interface applicative ATHAPASCAN1 ne permet pas de mesurer les communications, c'est pour cela que nous avons utilisé Athapascan0b

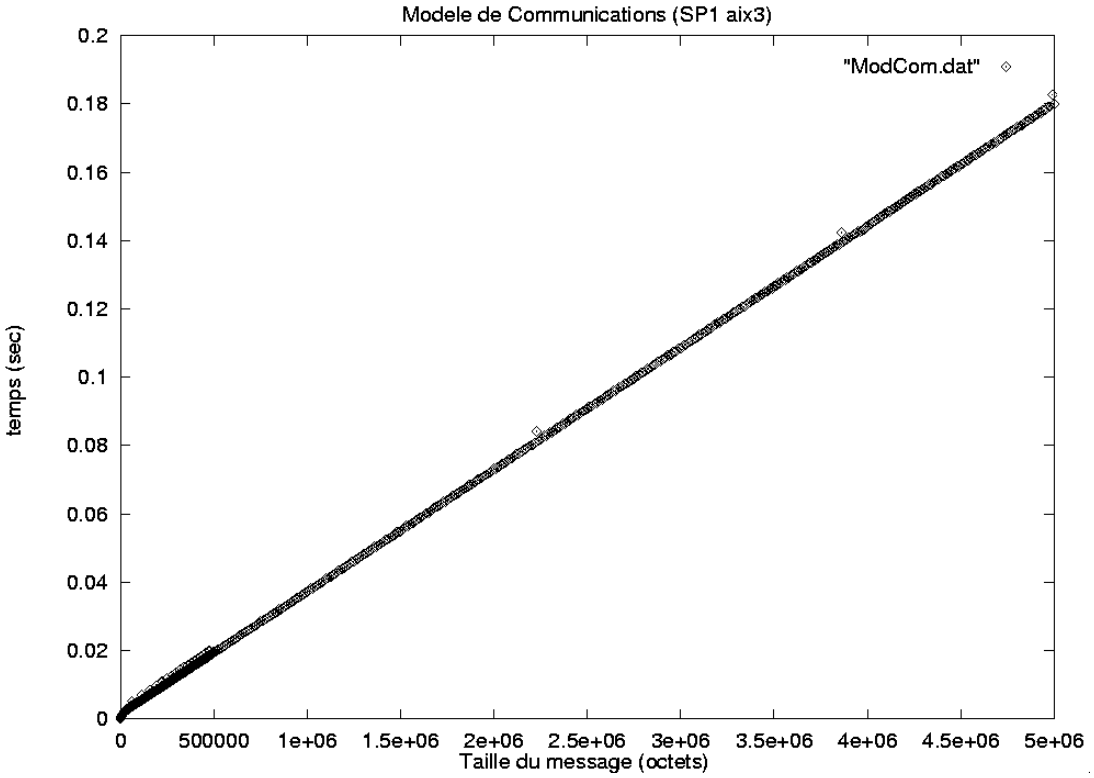


FIG. 6.2 – ATHAPASCAN0 : modèle de communication

Le modèle de calcul

Nous avons réalisé une seconde étude sur la machine SP, pour connaître les temps de calcul sur les noeuds de la machine. Nous avons réalisé la prise des mesures des temps de calcul sur une application ATHAPASCAN1 afin de donner un modèle de calcul qui prend en compte le surcoût de l'environnement ATHAPASCAN. On a fait 100 mesures pour chaque point. Chaque point est le temps de calcul de n additions d'entiers, ces valeurs sont rapportées dans la figure 6.3. Le temps de calcul de n additions entières est modélisé par une régression linaire :

– $t_{calc}(\mu) = 0.3408761 * n_{additions} + 63.37159$

Les modèles que nous avons mesurés décrivent la machine parallèle utilisée comme cible pour notre expérimentation (IBM SP1). Ces modèles sont primordiaux parce que vont nous permettre d'estimer le comportement de nos applications irrégulières (voir section 7.2.3).

6.5 Facteurs et modalités

Dans le système complexe qu'est l'exécution d'une application, nous considérons en général pour notre étude trois éléments : l'application parallèle, l'ordonnanceur et la ma-

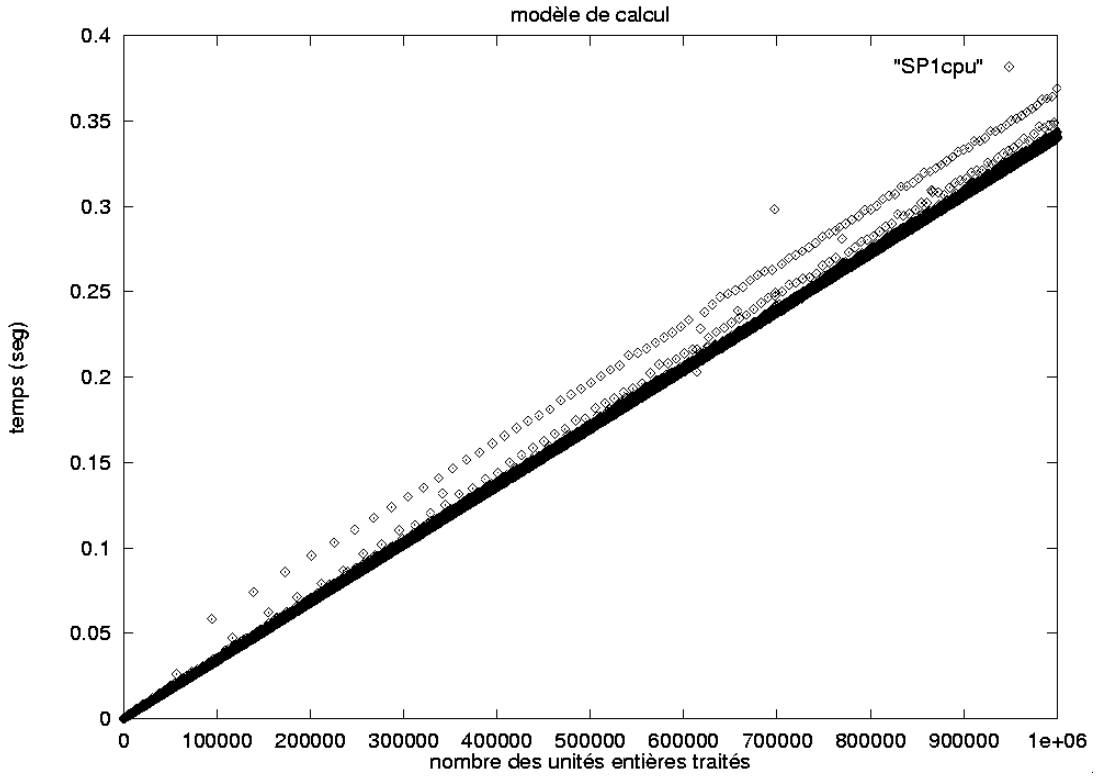


FIG. 6.3 – ATHAPASCAN1 : modèle de calcul

chine parallèle. Nous allons décrire chaque élément par un ensemble de paramètres (facteurs). Cette description doit nous permettre de nous abstraire du problème original. Pour la machine parallèle ce sera le nombre de processeurs, pour l'application synthétique le type de modèle, en faisant varier le nombre total des tâches et la granularité par exemple.

Décrivons notre système par rapport à ces éléments (voir figure 6.4. Les paramètres pour l'application sont notés $a_0, a_1, \ldots a_s$, les paramètres pour la méthode d'ordonnancement en considérant séparément les paramètres pour la politique d'information $i_0, i_1 \ldots i_t$, de la politique d'équilibrage $b_0, b_1 \ldots b_u$ et finalement les variables qui nous permettent la description de la machine parallèle $m_0, m_1, \ldots m_p$. La combinaison de ces paramètres représente le domaine expérimental.

Nous pouvons considérer qu'une combinaison de ces paramètres explique les performances du système. Nous avons choisi l'efficacité (*EFFI*) comme variable qui devrait être décrite par les paramètres :

$$f(EFFI) = (a_0, a_1, \ldots a_s, i_0, i_1 \ldots i_t, b_0, b_1 \ldots b_u, m_0, m_1, \ldots m_p).$$

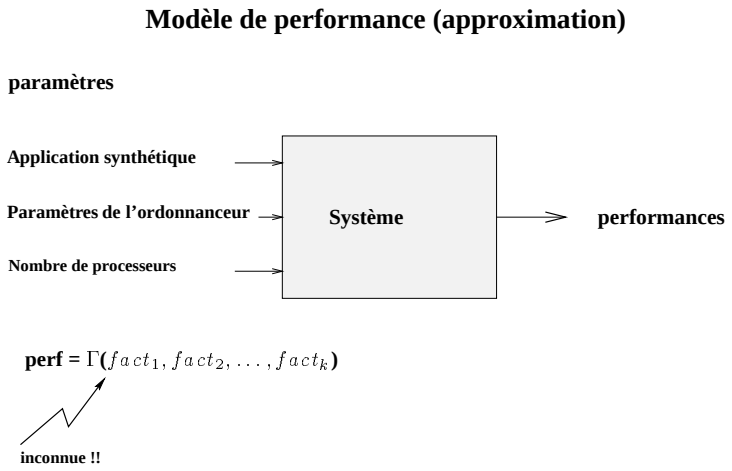


FIG. 6.4 – Le système complexe

6.5.1 Le jeu d'essai de charge synthétique

Il n'existe pas à l'heure actuelle de collection représentative et standard d'algorithmes parallèles en vue du test d'un régulateur dynamique de charge. La représentativité est importante pour la généralisation des conclusions tirées de nos expérimentations. Aussi le jeu de test que nous avons mis au point est basé sur des programmes réels : *Fusion*, *Quicksort*, *Jacobi* et *JacobiCreux*.

Fusion Le premier programme modélisé est un programme de tri par fusion d'un vecteur. C'est un algorithme à structure d'arbre équilibré car la découpe est équitable entre les tâches filles. Le coût de calcul d'une feuille est de l'ordre de n^2, le coût de division est nul, et le coût de fusion est de l'ordre de n.

Quicksort Le second programme modélise un tri rapide d'un vecteur. Cet algorithme a une structure d'arbre déséquilibré dans le cas général. En effet la découpe du vecteur lors de la division se fait par rapport à un pivot, ainsi les deux sous vecteurs sont en général de tailles différentes. Le coût de la division est de l'ordre de n, le coût de calcul d'une feuille est de l'ordre de n^2, et le coût de fusion est nul. La profondeur du graphe est variable car la découpe récursive doit être arrêtée si la taille des données passe en dessous d'un certain seuil.

Jacobi Le troisième programme est de type série-parallèle. Il est une modélisation de l'algorithme de calcul de point fixe selon la méthode de Jacobi. On considère que le programme que nous modélisons continue à garder la même structure (le même nombre de tâches) tout au long du déroulement du programme. Les coûts de calcul sont dans un premier temps fixés et communs à toutes les tâches.

JacobiCreu Dans un second temps, nous modélisons le même programme mais qui travaille sur des matrices creuses, ce qui implique un coûts de calcul et communication aléatoire. Nous considérons que le programme que nous modélisons continue à garder la même structure (le même nombre de tâches) tout au long du déroulement du programme.

Paramétrage d'entrée des programmes synthétiques

Nous désirons apprécier la qualité des régulateurs par leur capacité à réguler des programmes ayant des granularités de calcul fortement inégalé et une variation dans le nombre de tâches. La granularité des calculs est définie par le rapport entre le coût de calcul et le coût de communication. Nous avons défini trois granularités de calcul pour les tâches. Pour calculer ce rapport nous ne prenons en compte que le coût de démarrage de la communication ou «start-up» = 1391 μs:

Granularité

Faible granularité de calcul : dans ce cas le coût de calcul des tâches composant les programmes est de l'ordre de 10 fois une mise en route de communication.

Granularité moyenne : les programmes à granularité moyenne ont un coût de calcul de l'ordre de 300 fois une mise en route de communication.

Forte granularité : dans ces programmes le calcul est fortement prépondérant par rapport aux communications de l'ordre de 5000 fois le «start-up».

Nombre de tâches Il est important de faire varier le nombre de tâches qui composent les applications. En effet pour des programmes dont les tâches sont nombreuses, et à coût de calcul équivalent, les régulateurs sans élément d'information ont la possibilité de réaliser un bon placement. On considère trois cas différents pour le nombre de tâches (faible < ou = 100 tâches, moyenne=300 tâches et forte = 1000 tâches).

Pour chaque exécution, comme le phénomène étudié est dynamique et irrégulier, il n'a pas le même comportement pour chaque exécution et les observations. Par exemple le nombre de tâches exécutées dans chaque noeud où le travail effectué varie, et nous nous ne pouvons pas faire de pré-analyses et devons attendre la fin de l'exécution pour récupérer les informations de l'application (nombre de tâches, total de travail, nombre de communications) en plus, des observations du type le temps total écoulé et les temps que les processeurs ont passé dans les phases de travail et communication. Cela a été la difficulté expérimentale principale. Les valeurs des paramètres qui représentent chaque expérience ainsi que les observations qui ont été obtenues après chaque exécution forment une matrice qui sera analysée.

Les valeurs des paramètres pour les graphes synthétiques sont indicatives et de référence pour l'exécutif de l'application synthétique. Par exemple dans la figure 6.5 nous montrons les observations (moyennes) pour l'application QuickSort. Dans chaque graphique il y a 3 courbes que correspondent à 2,5 et 8 processeurs. Chaque courbe reporte 9 valeurs correspondantes aux 9 expériences pour l'application synthétique qui corresponde à 3 granularités et à 3 différents nombres de tâches. On observe que les granularités de

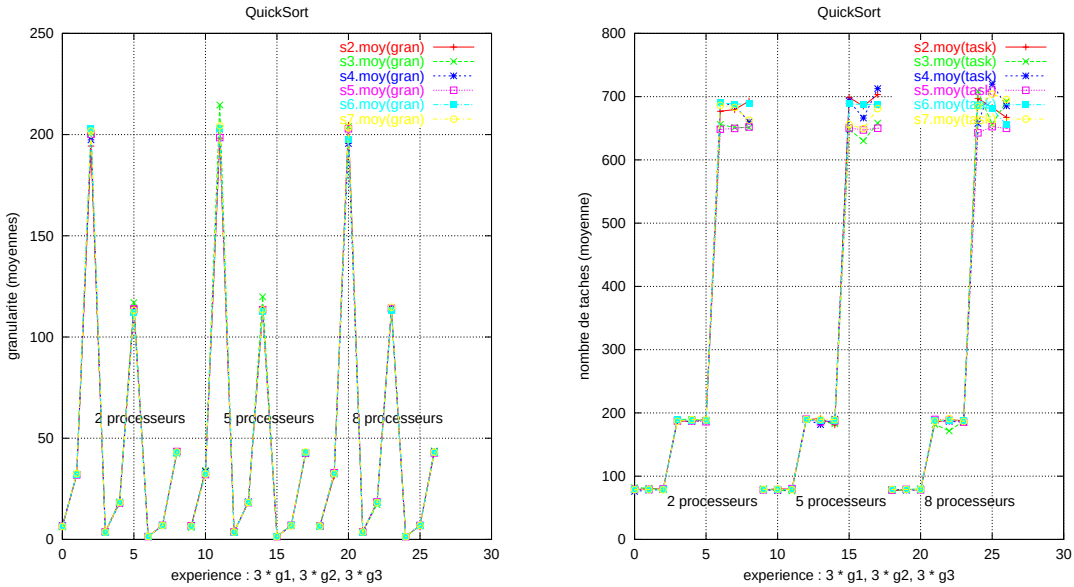


FIG. 6.5 – Valeur moyenne pour la Granularité estimée et le nombre de taches de l'application synthétique QuickSort

base (des trois premières expériences) diminuent quand le nombre de tâches est incrementé. Les valeurs ont été récupérées après l'exécution de chaque expérience.

6.5.2 Les stratégies pour la régulation dynamique de charge

Il sera impossible dans le cadre de cette thèse d'évaluer toutes les stratégies de régulation de charge. Dans la description expérimentale qui précède, nous avons noté que le nombre de variables est très grand. Nous débutons ce travail dans le cadre d'une implémentation centralisée. Nous voulons trouver pour un algorithme donné les paramètres pour lesquels il se comporte correctement. Nous resterons toujours avec des stratégies très simples et ferons la comparaison avec des algorithmes sans élément d'information comme les algorithmes cycliques et aléatoires.

S1 : séquentiel c'est l'exécution séquentielle de l'application. Pour notre cas, ce sera une simulation de l'exécution du travail total réalisé pour l'application synthétique de référence. Nous l'avons calculé pour chaque expérience à partir du modèle de calcul de la machine parallèle et le total des opérations réalisées effectivement en parallèle.

S2 : *random* c'est un algorithme décentralisé, qui ne prend pas en compte la charge des processeurs. Lors de la création de chaque nouvelle tâche, le site de réception est choisit de façon aléatoire. Le placement n'est pas remis en cause.

Observations :

– c'est un algorithme sans coût d'implantation.

– si le nombre de tâches est très grand et la différence de calcul entre les tâches est petite, il donne de bons résultats.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. **MAXT** Seuil pour le placement local ou l'exportation.

S3 : *modulo décentralisé* c'est un algorithme décentralisé, qui ne prend pas en compte la charge des processeurs. Lors de la création d'une nouvelle tâche, le site de réception est choisit selon la formule $i \, modulo \, p$, i étant le numéro d'ordre de la tâche et p le nombre total de processeur. Chaque processeur a un compteur propre.

Observations :

– c'est un algorithme sans coût d'implémentation.
– si la variation des coûts de calcul entre les tâches est petite, cet algorithme établit une bonne répartition.
– il ne prend pas en compte d'information sur l'état de charge des processeur, et il ne peut pas donner de résultats optimaux, s'il y a une grande variation entre les coûts.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. Cet ordonnanceur n'utilise pas le seuil *MAXT*.

S4 : *cyclique centralisé* c'est un algorithme centralisé très simple, qui ne prend pas en compte la charge des processeurs. Les processeurs peuvent garder des tâches lors de la création et si leur charge est supérieure au seuil *MAXT* elle est envoyée au contrôleur central. Le contrôleur central va placer les tâches tout de suite (il n'a pas de liste de réserve). Le processeur récepteur est choisi selon la formule $i \, modulo \, p$ processeur (p nombre de processeurs). Le placement n'est pas remis en cause.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. **MAXT** Seuil pour le placement local ou l'exportation.

S5 : *centralisé sans réserve* c'est un algorithme centralisé sans réserve, ce qui veut dire qu'au moment de la réception des tâches, il va tout de suite les placer. Il choisit le processeur récepteur en utilisant les informations sur l'état de charge, le moins chargé sera choisi. Lors de la création, la tâche peut être placée localement ou envoyée au contrôleur. L'indice de charge est l'indice complexe, défini dans la section 5.3.2. Le protocole pour la collecte des informations de charge est un protocole par messages volontaires. Chaque processeur envoi son état au collecteur si sa charge a varié *EVEN* événements par rapport au dernier message.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. **MAXT** Seuil pour le placement local ou l'exportation.
3. **EVEN** Seuil pour l'envoi des informations de charge au collecteur.
4. **LocalLoad** L'indice de charge utilisé est l'indice complexe.

S6 : *centralisé avec réserve* c'est un ordonnanceur centralisé avec une liste de réserve des tâches. Le contrôleur va placer la tâche dans le processeur le moins charge jusqu'à un seuil *JOBT* et puis va attendre une nouvelle activation. L'ordonnanceur est activé par la réception des nouvelles tâches ou des nouvelles informations sur l'état de charge. L'indice de charge est l'indice complexe, défini dans la section 5.3.2. Le protocole pour la collecte des informations de charge est un protocole par messages volontaires. Chaque processeur envoi son état au collecteur si sa charge a varié de *EVEN* événements par rapport à la dernière valeur informée.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. **MAXT** Seuil pour le placement local ou l'exportation.
3. **EVEN** Seuil pour l'envoi des informations de charge au collecteur.
4. **LocalLoad** L'indice de charge utilisé est l'indice complexe.
5. **JOBT** Nombre maximal de tâches que le contrôleur central va envoyer aux processeurs qui ont un charge faible.

S7 : *global-centralisé* c'est un ordonnanceur avec une liste de réserve de *Jobs*, il a un fonctionnement identique à celui de l'algorithme centralisé avec réserve S6, sauf qu'à la place d'utiliser l'indice complexe, cette ordonnanceur est seulement activé par les informations sur des tâches quand elles sont finies par que l'indice de charge est l'indice simple, défini dans la section 5.3.2. Le protocole pour la collecte des informations de charge est un protocole par messages volontaires. Chaque processeur envoi son état au collecteur si sa charge a varié *EVEN* événements par rapport à la dernière valeur informée.

Paramètres à faire varier :

1. **MRUN** Nombre de tâches qui débutent leur exécution en même temps. C'est le nombre de processus légers concurrents.
2. **MAXT** Seuil pour le placement local ou l'exportation.
3. **EVEN** Seuil pour l'envoi des informations de charge au collecteur.
4. **LocalLoad** L'indice de charge utilisé est l'indice simple simple.
5. **JOBT** Nombre maximal des tâches que le contrôleur va envoyer au processeur qui a activé l'ordonnanceur central.

Bilan sur les paramètres des ordonnanceurs

Nous rapportons dans le tableau 6.2 les paramètres qui caractérisent les différents ordonnanceur : *random, modulo décentralisé, cyclique centralisé, placeur central, centralise avec réserve, global avec réserve.*

	Paramètres					
	MRUN	MAXT	JOBT	EVENT	INDCHARGE	RESERVE
s1 : séquentiel	non	non	non	non	non	non
s2 : random	oui	oui	non	non	non	non
s3 : modulo desc.	oui	non	non	non	non	non
s4 : cyclique cent.	oui	oui	non	non	non	non
s5 : placeur central	oui	oui	non	oui	complexe	non
s6 : central-reserve	oui	oui	oui	oui	complexe	oui
s7 : global central	oui	oui	oui	oui	simple	oui

TAB. 6.2 – Les facteurs (paramètres) pour les stratégies d'équilibrage de charge

Domaines de chaque facteur

Les domaines des facteurs de l'ordonnanceur sont limités par des contraintes techniques et les observations d'une première expérimentation faite avec P. Rouchon lors de son DEA [Rou96].

Élément d'information Quand l'ordonnanceur prend en compte les informations du collecteur pour la prise de décisions, cas des algorithmes, S5, S6, S7, la politique du protocole est volontaire et le flux des messages est contrôle par la valeur d'*EVEN*. Chaque processeur envoi son état au collecteur si sa charge a varié de *EVEN* événements par rapport à la dernière valeur informée. Si *EVEN* est nulle, le processeur informe sa charge à chaque événement qui change l'indice de charge. Une grande valeur (100) pour *EVEN* implique que le collecteur ne sera jamais informé des événements qui se passent dans les processeurs. Suite à nos premières expériences [Rou96], le domaine est fixé entre [0,…,5]. Aussi pour les trois stratégies qui prennent en compte les informations sur l'état de charge, deux indices de charge différents sont considérés (voir section 5.5) : un indice simple (la fin des tâches) et un indice complexe.

Élément de contrôle Idéalement nous aimerions équilibrer la charge à 1 tâche près entre les processeurs. Pour cela les seuils ne devront pas être trop grands. Nous allons fixer les domaines des seuils :

- $MAXT \in [0 \ldots 2]$ où la valeur 0 indique qui il n'y a pas de placement local, et le valeur 2 un maximum de placements locaux, avant de l'envoyer au contrôleur central.
- $JOBT \in [1 \ldots 3]$ où 1 est le minimum possible. Si une valeur était fixé à 0, ce processeur n'enverrait jamais de travail.

Les résultats rapportés par Ginzbourg [Gin97] pour les performances d'ATHAPASCAN0 sur l'exécution concurrente de " threads " invite à utiliser cette fonctionnalité d'exécution concurrente, propre à toutes les systèmes basés sur la multiprogrammation légère. Le domaine de *MRUN* est fixé entre [1 ...10] où 1 est le nombre minimum des processus légers dans le processeur dédié à l'exécution de l'application. Nous notons que dans toute application ATHAPASCAN1 existe un nombre fixe de threads dédiés aux fonctionnalités de l'interface et de l'ordonnanceur.

6.6 La planification expérimentale

S'il était possible de faire toutes les combinaisons des paramètres avec un nombre très grand de modalités, on voit vite que le nombre d'expériences serait gigantesque (les paramètres peuvent prendre des valeurs différentes lors de chaque exécution). Pour diminuer le nombre d'expériences, nous avons fait appel aux plans factoriels (voir section 6.1.4).

Pour notre étude sur l'évaluation expérimentale des algorithmes de répartition dynamique, nous avons fait la planification de trois phases pour l'expérimentation : la première phase qui correspond au filtrage de facteurs (on détermine à partir de tous les facteurs, ceux qui ont une plus grand influence), une phase de modélisation qui correspond à la caractérisation quantitative de l'algorithme (on a établi les relations entre les facteurs prédominants et on a quantifié cette influence) et une dernière phase pour comparer les stratégies.

6.6.1 Résumons

- Nous avons 4 modèles d'application synthétique (*Fusion, QuickSort, Jacobi* et *JacobiCreu*) chacun avec 3 modalités pour la granularité et 3 modalités pour le nombre de tâches. Cela nous donne comme charge synthétique un jeu d'essai de 36 applications parallèles (**SYNA**).
- Pour la machine parallèle, nous avons à notre disposition 9 processeurs, dont un sera utilisé pour installer le contrôleur centralisé et 8 nœuds de calcul. Nous avons décidé de faire les expériences sur deux, cinq et huit nœuds de calcul : cela fait 3 modalités pour le facteur du nombre de processeurs (**NPROC**).
- Pour les stratégies de régulation étudiées les paramètres qui varient sont divers comment montre le tableau 6.2. Nous avons choisit les modalités suivantes.

1. EVEN = 0, 1, 5 ; un total de 3 modalités.
2. MRUN = 1, 2, 5, 10 ; un total de 4 modalités.
3. MAXT = 0, 1, 2 ; un total de 3 modalités.
4. JOBT = 1, 2, 3 ; un total de 3 modalités.

Cela nous donne un total de 30240 expériences. Etant donné la quantité d'expériences à réaliser, il nous importe d'automatiser leur exécution. D'un premier étalonnage des applications synthétiques nous obtenons des temps minimaux de 1 seconde et des temps

stratégie	combinaisons des modalités	total
s2	$3_{NPROC} * 36_{SYNA} * 4_{MRUN} * 3_{MAXT}$	1296
s3	$3_{NPROC} * 36_{SYNA} * 4_{MRUN}$	432
s4	$3_{NPROC} * 36_{SYNA} * 4_{MRUN} * 3_{MAXT}$	1296
s5	$3_{NPROC} * 36_{SYNA} * 4_{MRUN} * 3_{MAXT} * 3_{EVEN}$	3888
s6	$3_{NPROC} * 36_{SYNA} * 4_{MRUN} * 3_{MAXT} * 3_{EVEN} * 3_{JOBT}$	11664
s7	$3_{NPROC} * 36_{SYNA} * 4_{MRUN} * 3_{MAXT} * 3_{EVEN} * 3_{JOBT}$	11664
TOTAL		30240

TAB. 6.3 – Nombre de combinaisons des facteurs pour chaque algorithme

maximaux de 7 minutes. Si une expérience n'est pas finie après 10 minutes nous considérons qu'il y a un problème et cette expérience est arrêtée et recommencée.

6.7 Observations

L'exécution des programmes synthétiques est tracée et des informations de performances sont collectées pendant l'exécution. Pour cela nous avons instrumenté les applications synthétiques. Le type et la quantité de données collectées dépendent des objectifs.

Notre problème est que à cause du caractère aléatoire et irrégulier des applications on ne peut avoir une connaissance à priori du total du travail réalisé et du total des données transmises pendant l'exécution. Rappelons que les valeurs données en entrée pour les programmes synthétiques sont des directives pour l'exécutif, mais nous ne connaissons pas les coûts réels. Il nous a été impossible de tout mesurer, sachant que plus les traces sont détaillées, plus leur collecte est intrusive [Mai96].

Pour éviter de transférer les observations au cours de l'exécution d'une application, nous avons crée et installé au début de l'exécution un objet accumulateur (classe C++ accumulateur) dans chaque processeur, c'est un objet statique et seulement quand l'application est finie, nous récupérons les observations.

On a décidé de mesurer le temps total d'exécution de l'application parallèle (makespan ou t_{par}) comme mesure globale, puis dans chaque processeur on a compté le nombre total des tâches de calcul exécutées (*NbTask*), les tâches génératrices de travail (nommées nœuds internes : *NbInter*), le total des opérations effectuées (opérations entières UCPU), le temps, cumulé pendant lequel le processeur calcule (TCPU), le total des données communiquées par l'application (ceux dont le processeur était la source).

A la fin de chaque exécution nous avons fait un traitement sur les observations mesurées pour pouvoir obtenir des paramètres comme l'accélération, l'efficacité, le rapport entre les données transmises et le nombre de communications, la granularité et le coût des communications. La description du traitement sera faite dans le chapitre suivant.

Chapitre 7

L'expérimentation et l'analyse des mesures

Dans ce chapitre nous allons analyser les expériences faites sur une machine parallèle réelle pour un ensemble d'algorithmes d'équilibrage de charge. Nous allons séparé l'analyse en deux phases : la première permettra la description de la charge applicative avec un nombre réduit de variables, la deuxième phase nous permettra d'extrapoler des modèles linéaires du comportement de chaque ordonnanceur.

A partir d'une première série d'expériences et d'observations, on a réalisé la planification expérimentale basée sur des plans factoriels fractionnaires à trois niveaux. Pour cela, trois mois de calcul ont été nécessaires, et les ressources ont été utilisées jour et nuit. Toutes les expériences (30,000 environ) prévues n'ont pas abouti, nous présentons un bilan général dans la section 7.2.

Pour les analyses de nos expériences nous avons utilisé de façon complémentaire deux méthodes statistiques multidimensionnelles, l'analyse en composantes principales et la régression multiple, que nous allons introduire dans la première partie de ce chapitre.

Nos références essentielles sont deux travaux, Kitajima [Kit94] et Jacqmot [Jac96b] qui ont une problématique similaire. Tous les deux ont suivi une démarche systématique pour la planification expérimentale et l'analyse des résultats. Pour l'analyse statistique de leurs des observations (les bases de données sont caractérisées par un nombre important des variables et des mesures) ils utilisent des méthodes différentes.

Kitajima a fait une étude sur des algorithmes d'ordonnancement statiques, et utilise l'analyse en composantes principales afin d'éliminer des informations redondantes (filtrage de variables) pour la description des applications parallèles.

Jacqmot a fait une étude sur des stratégies d'équilibrage de charge dans les systèmes d'exploitation et utilise l'ajustement pour la méthode des moindres-carrés afin déterminer les facteurs (paramètres) plus significatifs des ses algorithmes. Elle approche le phénomène par un modèle linéaire du deuxième degré.

Rappelons notre problème : nous avons 6 stratégies d'équilibrage dynamique de charge. Nous avons appliqué à ces régulateurs une charge "synthétique" dynamique et pour chacune des stratégies, nous avons fait varier les différents paramètres. Nous avons récu-

péré un nombre important d'observations. Certaines de ces observations se rapportent à des applications dynamiques et donnent des informations sur ces applications mêmes.

Avec ces observations et les valeurs des entrées des paramètres de l'ordonnanceur pour chaque exécution, nous avons construit notre base d'observations. La ligne i dans la base de données contient toute l'information sur l'expérience i, l'ordonnanceur utilisé, les valeurs des paramètres, le modèle de base de la charge synthétique, les observations faites sur l'application exécutée et un indice de performance.

Faire les analyses avec l'ensemble des variables qui représentent les applications synthétiques est très coûteux et d'interprétation difficile. Elles peuvent donner des erreurs dues à la redondance de l'information. Mais quelles variables garder et quelles autres éliminer ? Et pourquoi ? Dans la section trois nous procédons au crible des variables qui modélisent l'application parallèle en utilisant l'analyse en composantes principales . Ensuite, après le filtrage des variables de l'application, nous avons procédé à l'analyse de chaque stratégie de régulation de charge par rapport à ses propres paramètres et à la façon dont elle a répondu face à la charge synthétique. Dans ce cas nous avons utilisé la régression multiple, qui nous permet de déterminer les facteurs qui ne sont pas significatifs. Un modèle linéaire a été obtenu [1]. Et finalement, nous avons comparé les ordonnanceurs.

Nous proposons l'utilisation de l'analyse en composantes principales et de la régression multiple de façon complémentaire pour explorer et analyser nos observations expérimentales [2]. Mais sans un outil statistique spécifique pour l'analyse exploratoire multidimensionnelle, faire ses analyses est impossible. Nous avons utilisé SPADN [3].

7.1 Présentation des méthodes statistiques utilisées

La liaison entre deux variables **quantitatives** peut être analysée de deux façons : soit une analyse de **corrélation**, soit une analyse de **régression**.

Pour distinguer les deux démarches, considérons les exemples suivants :

- nous observons deux variables pour différentes applications parallèles irrégulières : la taille finale du problème résolu et le nombre de communications générées. On voudrait savoir si ces valeurs observées sont indépendantes ou liées. Ce problème est entièrement symétrique par rapport aux deux variables. Ici il s'agit d'un problème de corrélation.
- maintenant nous voulons établir la relation qui existe entre la taille d'un message envoyé et le temps mesuré pour l'envoi du message. Les deux variables jouent cette fois un rôle dissymétrique ; la variable taille est la cause et la variable temps l'effet. L'information que nous voulons établir est la régression de la variable temps par rapport à la taille du message.

La différence principale entre ces deux exemples est, que dans le premier, les valeurs

[1] Le modèle linéaire est le plus simple car seuls les effets principaux sont considérés et il se prête bien à une interprétation graphique.

[2] En effet, cette démarche est recommandée quand il existe un nombre très grand de variables [Leb95].

[3] Nous remercions le groupe de SPADN pour l'aide, les conseils et les modifications qu'ils ont bien voulu faire sur le logiciel pour pouvoir supporter notre base des données.

observées des deux variables sont des variables aléatoires, tandis que dans le second, la variable taille est contrôlée par nous, le temps mesuré étant seul aléatoire.

Si certaines situations méritent sans ambiguïté le nom de corrélation ou de régression, d'autres peuvent recevoir l'une ou l'autre des deux dénominations et être qualifiées d'intermédiaires [Sch95].

Mais nous devons établir des corrélations entre un nombre important de variables et puis calculer leur influence sur les performances observées.

7.1.1 Statistique exploratoire multidimensionnelle

Le développement des techniques statistiques d'exploration multidimensionnelle, procède de l'effet conjugué de plusieurs facteurs : la possibilité de traiter des tableaux de données complexes et de grande taille, le regain d'intérêt suscité par de tels tableaux, la possibilité d'effectuer des opérations algorithmiques complexes.

Schématiquement, ces techniques comprennent deux familles principales de méthodes : les méthodes faisant appel à l'algèbre linéaire, désignées en France sous le nom de méthodes factorielles, et les techniques de classification automatique. L'approche des méthodes de classification consiste à faire des groupements les moins arbitraires possibles.

Les techniques factorielles sont destinées à fournir des représentations et des réductions, complémentaires, de l'information contenue dans de volumineux tableaux de données numériques. Elles reposent toutes sur une propriété mathématique des tableaux (ou matrices) rectangulaires : la décomposition en valeurs singulières (décomposition d'Eckart et Young).

Le tableau de données doit présenter une certaine homogénéité de forme et de contenu. Les lignes $(i = 1 \ldots n)$ représentent les n observations (ou individus) et les colonnes $(j = 1 \ldots p)$ sont les p variables, qui peuvent être des mesures (variables **quantitatives** ou numériques) ou des attributs (variables **qualitatives ou nominales**).

Il est toujours possible de calculer des distances entres les lignes et les colonnes d'un tableau rectangulaire de valeurs numériques. Ces distances peuvent s'interpréter en termes de corrélations ou de similarités. En revanche, il n'est pas possible de visualiser ces distances de façon immédiate : il est nécessaire de procéder à des transformations ou à des approximations pour en obtenir une représentation plane. C'est là l'une des tâches dévolues à l'analyse factorielle au sens large : opérer une réduction de dimension de certaines représentations "multidimensionnelles".

La propriété de décomposition concernera le tableau de données lui-même, et non pas seulement la matrice de corrélation ou un tableau de distances construit à partir des données. Elle a ceci de remarquable qu'elle implique de façon similaire les lignes et les colonnes du tableau, et donc en général les observations (lignes) et les variables (colonnes).

Parmi les méthodes d'analyse factorielle se trouvent :

1. Analyse en correspondances simples
2. Analyse en correspondances multiples
3. Analyse en composantes principales

L'analyse en correspondances simples est utilisée quand le tableau des informations est formé par des valeurs nominales (qualitative) à deux niveaux, normalement des 1 et des 0. L'analyse en correspondances multiple est utilisée pour des bases d'observations de variables nominales (qualitative) avec plusieurs niveaux. L'analyse en composantes principales est utilisée pour des bases d'observations de variables continues (quantitative). Nous avons utilisé l'analyse en composantes principales à cause de la nature continue de nos variables.

Analyse en composantes principales (A.C.P.)

L'analyse en composantes principales (**A.C.P**) est la plus simple des méthodes utilisées en analyse descriptive multivariée. Conçue pour la première fois par Karl Pearson en 1901, elle a été intégrée à la statistique mathématique par Harold Hotelling en 1933. Elle s'applique aux tableaux de type "variable-individus", dont les colonnes représentent des variables à valeurs numériques, et les lignes des individus, des observations, des objets, etc. Les proximités entre les variables s'interprètent en termes de corrélation et les proximités entre individus s'interprétant en termes de similitudes globales des valeurs observées [Leb95].

Deux représentations géométriques existent : le nuage des individus et le nuage des variables, toutes les deux sur un nouvel espace, les axes factoriels.

Dans un espace de dimension q, on considère n individus et on tente de les représenter graphiquement. Il s'agit donc de projeter ces individus sur un ensemble de dimension inférieure, pour nous un plan, afin de visualiser les résultats. La réduction de la dimension de l'espace fait perdre des informations sur les individus. L'ACP cherche à projeter les individus sur un espace de dimension réduite qui fournira une " bonne représentation " des individus. Ce " meilleur espace de projection " sera aussi proche que possible du nuage des points et déformera le moins possible sa structure. Pour cela chacun des deux axes est une combinaison linéaire axes des variables. Chaque variable participe donc plus ou moins aux axes du plan de projection. L'ACP permet de représenter les individus mais aussi les variables en fonction de leur importance pour distinguer les individus.

Le coefficient de corrélation est une valeur qui nous permet d'estimer (ou d'observer) l'indépendance ou la corrélation entre deux variables, sa valeur est toujours comprise entre -1 et +1. Plus la valeur (en valeur absolue) est petite, plus forte est la probabilité que les deux variables soient indépendantes. La formule du coefficient de corrélation est :

$$r = \frac{\sum(x - m_x)(y - m_y)}{\sqrt{\sum(x - m_x)^2 \sum(y - my)^2}}$$

où m_x et m_y désignent les moyennes observées des x et y.

Dans le nuage des variables, les variables fortement corrélées avec un axe vont contribuer à la définition de cet axe. Cette corrélation se lit directement sur le graphique du nuage des variables, tel que celui de la figure 7.1. Les variables qui présentent les plus fortes coordonnées sont situées à proximité du cercle de corrélation (cercle unitaire). Le cosinus de l'angle sous lequel on voit deux variables est le coefficient de corrélation.

Selon la qualité de l'ajustement, cette propriété sera plus ou moins bien conservée en projection. Il faut éviter d'interpréter la distance entre deux variables qui ne sont pas proches du cercle de corrélation [Leb95].

Remarque, l'analyse en composantes principales ne traduit que les liaisons linéaires entre les variables. Un coefficient de corrélation faible entre deux variables signifie donc que celles-ci sont indépendantes linéairement alors qu'il peut exister une relation de degré supérieur à 1 (liaison non linéaire).

Pour la représentation des individus, la position des points dans le nuage est donnée par l'ensemble des distances entre tous les points. Chaque point contribue à l'inertie des axes, mais quelques-uns plus que les autres. Les individus qui contribuent le plus à la détermination de l'axe sont les plus excentrés et l'examen des coordonnées factorielles ou la lecture du graphique suffisent à interpréter les facteurs dans ce cas.

7.1.2 La régression multiple

La régression multiple vise à expliquer ou prédire une variable continue (dite variable **dépendante** ou **endogène**) à l'aide d'un ensemble de variables dites **explicatives** (ou **exogènes**). On réserve en général le nom de régression multiple au cas où les variables explicatives sont continues. Lorsque celles-ci sont des variables nominales, on parle d'analyse de la variance et pour un ensemble de variables mixtes, d'analyse de la covariance [Leb95].

Description du problème de la régression multiple

On dispose d'un ensemble de n observations sur lesquelles ont été effectuées $p+1$ mesures des variables : $\mathbf{y}, \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_p}$. On veut expliquer ou prévoir $\mathbf{y}$ à l'aide des variables explicatives ou prédicteurs, $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_p}$, lesquels sont supposés connus sans erreur.

On cherche à approcher $\mathbf{y}$ par une combinaison linéaire des variables explicatives ($\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_p}$). Pour cela un modèle est posé :

$$y_i = \alpha_0 + \alpha_1 x_{i1} + \alpha_2 x_{i2} + \ldots \alpha_p x_{ip} + \varepsilon_i$$

où $\alpha_0, \alpha_1, \alpha_2, \ldots \alpha_p$, sont les coefficients inconnus du modèle, α_0 est le terme constant et ε_i est le résidu représentant l'écart entre la valeur observée y_i et la partie " expliquée " de l'observation ($\alpha_0 + \alpha_1 x_{i1} + \alpha_2 x_{i2} + \ldots \alpha_p x_{ip}$).

On dispose pour évaluer les coefficients inconnus du modèle, d'un système de n équations linéaires ayant $n+p+1$ inconnues. Le système admet donc une infinité de solutions. Soient $a_0, a_1, \ldots, a_p$ les coefficients correspondant à une des solutions possibles. On cherchera la solution qui minimise un critère, le plus souvent la minimisation de la somme des carrés des écarts ou moindres carrés $min\{\sum \varepsilon_i^2\}$.

7.2 Bilan général sur l'expérimentation

Nous nous sommes placés dans un contexte expérimental de 30240 expériences. Différents événements se sont produits et toutes les expériences n'ont pas abouti. Dans un premier passage très rapide sur nos mesures nous avons éliminé les points aberrants et refait si possible les expériences éliminées, à l'exception de deux cas (pour les stratégies s6 et s7, dont un ensemble partiel à été éliminé des analyses) que nous verrons dans la section 7.5.3.

Il n'était pas envisageable de faire des répétitions de toutes les mesures, cela aurait prit beaucoup de temps. Pour les mesures qui nous ont "semblé" un peu hors norme, nous avons refait l'expérience et si la différence était significative, nous avons décidé de laisser les deux résultats. Cela explique la différence entre le total des mesures prévues et le total rapporté et utilisé pour l'analyse. La quantité de données des résultats expérimentaux est d'approximativement 18,000 expériences avec une durée minimale de 1 seconde et maximale de 10 minutes. Dans la table 7.1 nous rapportons pour les expériences qui vont être analysées (que nous avons gardé), les valeurs des paramètres de l'ordonnanceur et le total des expériences pour chacun des ordonnanceurs.

	Paramètres								
stratégie	APSY	PROC	MRUN	MAXT	JOBT	EVEN	FACT	RESER	TOTAL
s2: random	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	{1, 2, 3}	1	10	3	non	1302
s3: modulo	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	1	1	10	3	non	429
s4: cyclique	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	{1, 2, 3}	1	10	3	non	1294
s5: central	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	{1, 2, 3}	1	{1, 2, 6}	1	non	3941
s6: réserve	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	{1, 2, 3}	{1, 2, 3}	1	1	oui	6994
s7: global	1 . . .39	{2, 5, 8}	{1, 2, 5, 10}	{1, 2, 3}	{1, 2, 3}	1	2	oui	3821
ENSEMBLE									17781

TAB. 7.1 – Les expériences à analyser

7.2.1 Notation

Reprenons les notations utilisées pour les analyses :

Paramètres pour les graphes synthétiques

1. *DATA* : total des données transmises pendant l'exécution.
2. *CPUS* : total des opérations faites (travail fait par l'application).
3. *CCOM* : coût total des communications (microsecondes).
4. *TSEQ* : coût total des opérations, en temps (microsecondes).
5. *TTAS* : nombre total de nœuds ou tâches.
6. *GRAN* : granularité, rapport entre le coût total de calcul et le coût total des communications.

7. *ARRET* : nombre total des communications.
8. *LOCH* : le plus long chemin du graphe ou chemin critique.
9. *FLOT* : la moyenne des données transmises dans chaque arrêt.
10. *PVIR* : le parallélisme virtuel.
11. *SYNA* : les identificateurs des applications synthétiques.

Paramètres pour les stratégies d'équilibrage

1. *EVEN* : seuil pour l'envoi de l'information au collecteur.
2. *FACT* : indice de charge utilisé.
3. *MAXT* : seuil pour l'exécution locale ou l'envoi au contrôleur.
4. *JOBT* : nombre de tâches que les processeurs de calcul vont recevoir du contrôleur.
5. *MRUN* : nombre de tâches concurrentes (il faut le voir comme le nombre de threads de l'application qui vont s'exécuter en concurrence).
6. *ORDO* : identificateur de l'ordonnanceur utilisé pour l'expérience.

Paramètres de la machine parallèle

1. *PROC* : nombre de processeurs utilisés pendant l'exécution.

Indices des performances

1. *TPAR* : valeur mesurée, du temps total d'exécution parallèle.
2. *ACCE* : l'accélération, valeur calculée à partir de *TPAR* et *TSEQ*.
3. *EFFI* : l'efficacité, valeur calculée à partir de l'accélération divisée par le nombre de processeurs *PROC* utilisés pour l'exécution.

Re-codage de la base de données

SPADN a son dictionnaire pour la représentation de valeurs. Une valeur égale à zéro qui dans nos expériences peut avoir une signification spéciale, pour SPADN est interprétée comme une absence de mesure ou de valeur. Pour cette raison, nous avons dû coder les valeurs des paramètres des ordonnanceurs. Par rapport aux valeurs établies au chapitre 6, les changements sont les suivants :

- Pour *EVEN* nous avons mis les valeurs 1,2,6 et 10 [4] à la place de 0, 1, 5 et 100.
- Pour *MAXT* à la place de 0,1 et 2 nous avons mis 1, 2, 3.
- Pour *FACT* trois valeurs ont été codées : 1 pour l'indice simple, 2 pour l'indice complexe et 3 pour l'absence d'indice.

Donc notre base de données est formée par 21 variables (colonnes) et 17781 expériences (lignes)

[4] Nous avons changé aussi la valeur de 100 pour éviter d'avoir trop d'écart et faussent les analyses

La variable nominale *SYNA*

SYNA est une variable nominale qui identifie chacun des modèles de notre charge synthétique, avec 36 niveaux valides (ou modalités) et trois niveaux (10, 20, 30) non utilisés. Pour les applications du type *Fusion*, *SYNA* prend les valeurs de 1 à 9, pour les *QuickSort*, elle prend les valeurs de 11 à 19, pour le type *Jacobi*, les valeurs 21 à 29 et pour les *JacobiCreu*, elle prend les valeurs de 31 à 39. Chaque groupe est divisé en groupes de trois. La première partie de chaque groupe correspond aux expériences avec peu de tâches (moins de 80), la deuxième correspond aux expériences avec un nombre moyen de tâches (environ 200) et la dernière correspond aux expériences avec un nombre important de tâches (environ 1000). Finalement pour chaque groupe de trois expériences, la première correspond à celle qui a la plus petite granularité, la deuxième la granularité moyenne et la troisième la plus forte granularité. Par exemple, l'application 26 (*AP26*) correspond à l'application de type *Jacobi*, avec un nombre moyen de tâches et avec la plus grande granularité.

7.2.2 Statistiques sommaires sur l'efficacité et les applications synthétiques

Dans le tableau 7.2, nous rapportons les statistiques sommaires sur l'efficacité (moyenne, écart-type, le minimum et le maximum) pour chaque application synthétique. Ici l'ensemble des résultats a été considéré.

Commentaires : Nous avons choisi d'utiliser l'efficacité comme indice de performance et cela pour pouvoir comparer toutes les expériences, puisque nous avons aussi fait varier le nombre de processeurs. Rappelons que l'efficacité est une valeur positive toujours plus petite ou égale à 1.

1. La première colonne est la modalité de *SYNA*. La deuxième colonne (effectif) correspond au total d'expériences fait globalement pour chaque application.
2. La première remarque est que la moyenne de l'efficacité est assez basse (par rapports aux résultats obtenus avec des stratégies statiques), nous ne devons pas oublier que nous cherchons à établir les valeurs les mieux adaptées pour chacune des stratégies. Cependant nous considérons que la moyenne globale de l'efficacité (0.556) est un bon indice de la qualité du choix du domaine expérimental.
3. Dans tout l'ensemble, deux applications attirent notre attention *AP27* et *AP37*, qui ont une moyenne "très faible" de moins de 0.2. Ces deux applications sont de type série-parallèle, avec une petite granularité et un grand nombre de tâches (environ 1000).
4. Nous avons obtenu quelques bonnes efficacités, la dernière colonne nous donne les valeurs maximales, pour 24 des 36 applications, nous avons obtenu des efficacités supérieures à 0.8.

		EFFICACITE			
IDENT	EFFECTIF	MOYENNE	ECART TYPE	MINIMUM	MAXIMUM
AP01	407	0.498	0.138	0.112	0.719
AP02	407	0.691	0.164	0.191	0.98
AP03	409	0.705	0.166	0.189	0.992
AP04	407	0.648	0.132	0.209	0.861
AP05	407	0.728	0.142	0.212	0.98
AP06	407	0.736	0.144	0.218	0.992
AP07	406	0.681	0.091	0.446	0.848
AP08	391	0.763	0.105	0.472	0.968
AP09	398	0.756	0.113	0.418	0.98
AP11	407	0.509	0.141	0.145	0.788
AP12	407	0.587	0.155	0.202	0.923
AP13	405	0.612	0.161	0.184	0.952
AP14	404	0.496	0.114	0.141	0.718
AP15	401	0.621	0.134	0.238	0.896
AP16	402	0.643	0.147	0.183	0.957
AP17	429	0.403	0.065	0.198	0.524
AP18	498	0.605	0.11	0.313	0.833
AP19	459	0.662	0.125	0.362	0.95
AP21	590	0.45	0.131	0.169	0.713
AP22	623	0.621	0.147	0.273	0.962
AP23	619	0.65	0.154	0.292	0.995
AP24	626	0.49	0.122	0.228	0.714
AP25	599	0.735	0.132	0.309	0.957
AP26	574	0.754	0.158	0.334	0.992
AP27	572	0.184	0.083	0.076	0.333
AP28	595	0.42	0.211	0.139	0.809
AP29	579	0.511	0.264	0.158	0.935
AP31	547	0.362	0.122	0.144	0.611
AP32	552	0.491	0.139	0.243	0.805
AP33	552	0.551	0.146	0.244	0.876
AP34	551	0.459	0.132	0.104	0.687
AP35	543	0.638	0.12	0.364	0.878
AP36	550	0.7	0.144	0.349	0.963
AP37	614	0.175	0.08	0.074	0.322
AP38	551	0.4	0.19	0.139	0.718
AP39	493	0.474	0.213	0.158	0.799
ENSEMBLE	17781	0.556	0.211	0.074	0.995

TAB. 7.2 – Efficacité globale pour chaque application parallèle

7.2.3 Le post-traitement des observations des applications synthétiques

Un graphe est un objet complexe qui contient une structure qu'il est très difficile de décrire par des paramètres. Il ne peut pas y avoir d'équivalence entre un graphe et les valeurs d'un n-uplet de paramètres. Si pour chaque graphe, il n'existe qu'un seul n-uplet de valeurs de ces paramètres, par contre pour chaque n-uplet on peut trouver une infinité de graphes de structures différentes. Néanmoins on peut espérer trouver un ensemble de paramètres globaux pour décrire les applications parallèles du jeu d'essai [5]. Il existe deux sortes de paramètres :

– les paramètres quantitatifs qui caractérisent les valeurs associées aux nœuds de cal-

[5] N'oublions pas qu'il s'agit des simulations des applications parallèles irrégulières dynamiques.

cul et aux arcs du graphe par exemple.

– les paramètres structuraux qui rendent compte de la structure du graphe.

Parmi les paramètres quantitatifs, on trouve le nombre de tâches ou nœuds (*TTAS*), le nombre total d'opérations (*CPUS*), le coût de l'exécution de toutes les opérations en microsecondes ou temps séquentiel (*TSEQ*), le coût des communications en microsecondes (*CCOM*) et le nombre total des communications (*ARRETS*).

A partir de ces paramètres simples mais influençant les stratégies, on peut définir d'autres paramètres : le grain du graphe qui est le rapport entre le coût des calculs et le coût des communications (*GRAN*). Un autre type de paramètres concerné est le chemin critique du graphe (*LGCH*), il s'agit de la longueur du plus long chemin du graphe. Ce chemin relie forcément l'unique tâche source et l'unique tâche puits pour les graphes étudiés. Le flot des données (*FLOT*) que nous avons définit comme le rapport du total des données transmises (*DATA*) sur le nombre de communications (*ARRETS*). Le parallélisme virtuel (*PVIR*) est le dernier paramètre structurel. Il est défini comme le nombre total de tâches divisé par le nombre de tâches appartenant au chemin critique.

Il est impossible de tout observer lors de l'exécution des applications, nous avons limité les observations à certains éléments à partir desquels les paramètres décrits auparavant sont calculés. Chaque processeur fait des observations, compte le nombre de tâches exécutées localement (*TASK*), le nombre d'opérations réalisées (*OPER*), le nombre de données transmises (*DATT*) et il garde la plus grande profondeur de graphe (*CHEMIN*) qu'il a exécuté.

Lors de la collecte des observations à la fin de l'exécution un ensemble de traitements doit être fait.

1. L'indice de performance que nous avons mesuré est le temps d'exécution de l'application (*TPAR*), le temps parallèle a été mesuré dans le processeur où l'application a débutée et où on attend la fin.
2. *TTAS*, est la somme des toutes les tâches qui ont été exécutées sur chaque processeur,

$$TTAS = \sum_{i=1}^{p} TASK_i$$

où p est le nombre de processeurs.

3. *CPUS*, est la somme de toutes les opérations arithmétiques calculées,

$$CPUS = \sum_{i=1}^{p} OPER_i$$

où p est le nombre de processeurs.

4. *DATA*, est la somme de toutes les données transmises pendant l'exécution,

$$DATA = \sum_{i=1}^{p} DATT_i$$

où p est le nombre de processeurs.

5. *TSEQ*, le temps séquentiel est estimé à partir du modèle de calcul que nous avons réalisé au préalable et qui a été présenté dans la section 6.4.1 et du total des opérations *CPUS* valeur en secondes:

$$TSEQ(sec) = 3.408761e-07 * CPUS + 6.337159e-05$$

6. *LGCH*, le chemin critique est la valeur la plus grande parmi toutes les observations faites par les processeurs :

$$max(CHEMIN_i) \;\; avec \;\; (i = 1, \ldots, p)$$

7. *ARRETS*, est un paramètre qui indique le nombre d'arcs qu'a le graphe de l'application et non le nombre effectif des communications "physiques" faites entre les processeurs. Ce paramètre est directement corrélé au nombre de tâches dans l'application. Par exemple pour les arborescences, chaque tâche de calcul aura au moins deux arcs, un en entrée et un autre en sortie. Si c'est une tâche génératrice de parallélisme, elle aura au moins un arc en entrée d'appel, deux arcs en sorties pour les deux tâches filles, deux arcs en entrée des résultats des filles et un retour d'elle-même vers son appelant, cela fait un total de trois arcs en entrée et trois en sortie. *ARRETS* est le compteur des arcs de sortie.
8. *CCOM*, comme ATHAPASCAN1 ne permet pas de mesurer directement les communications, nous avons fait une estimation de ce coût à partir du nombre total des communications (*ARRETS*) et du nombre total de données (*DATA*) transmises par les processeurs, donc :

$$CCOM(sec) = 3.582428e-08 * DATA + 1.391197e-03 * ARRETS$$

   Cette approximation ne prend pas en compte l'engagement possible du réseau.
9. *FLOT*, est le rapport entre les données transmises et le nombre des communications dans le graphe de l'application :

$$FLOT = \frac{DATA}{ARRETS}$$

10. *GRAN*, la granularité est calculée à partir "des valeurs estimées" du coût de calcul et du coût des communications

$$GRAN = \frac{TSEQ}{CCOM}$$

11. *PVIR*, le parallélisme virtuel est calculé à partir du nombre total de tâches *TTAS* et du chemin critique *LGCH* :

$$PVIR = \frac{TTAS}{LOCH}$$

12. *ACCE*, l'accélération est calculée à partir des valeurs mesurées *TPAR* et de la valeur estimée *TSEQ*.

$$ACCE = \frac{TSEQ}{TPAR}$$

13. *EFFI*, l'efficacité est calculée à partir de l'accélération et du nombre de processeurs *PROC* :

$$EFFI = \frac{ACCE}{PROC}$$

Le tableau 7.3, rapporte les statistiques sommaires "globales" de toutes les variables mesurées et calculées sur l'ensemble des observations faites lors de l'expérimentation.

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES
EFFECTIF TOTAL : 17781

Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
TSEQ	17781	67.435	87.816	1.31	538.469
CPUS	17781	**********	*********	**********	*********
DATA	17781	327057.8124	99507.438	5360.000**	********
FLOT	17781	327.032	246.927	27.37	740.925
CCOM	17781	1.334	1.233	0.225	4.377
GRAN	17781	99.696	156.693	1.278	741.752
TTAS	17781	475.803	439.406	80	1573
LGCH	17781	6.41	1.849	5	10
PVIR	17781	73.468	68.002	13.857	200
ARRT	17781	950.768	878.647	160	3144
Paramètres de l'ordonnanceur					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
PROC	17781	5.049	2.47	2	8
EVEN	17781	2.979	3.436	1	10
MAXT	17781	1.98	0.82	1	3
JOBT	17781	1.607	0.802	1	3
FACT	17781	2.445	0.766	1	3
MRUN	17781	4.496	3.496	1	10
Indice de performance					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
TPAR	17781	28.785	39.174	0.293	320.652
ACCE	17781	2.549	1.366	0.479	7.332
EFFI	17781	0.556	0.211	0.074	0.995

TAB. 7.3 – Les statistiques sommaires

7.3 Analyse des applications parallèles synthétiques

Nous avons once paramètres pour décrire les applications parallèles. Du fait de leur définition, beaucoup de ces paramètres représentent les mêmes informations sur les graphes. Nous avons besoin de limiter le nombre de paramètres pour pouvoir espérer déduire quelque chose lors de l'analyse des stratégies d'équilibrage de charge.

Nous allons maintenant procéder à une analyse pour déterminer les variables ayant un impact significatif et qui décrivent le mieux les applications en évitant des redondances d'information. Cette étape est importante car elle doit nous permettre de réduire le nombre de variables qu'il faut prendre en compte pour l'analyse des stratégies d'équilibrage de charge.

7.3.1 Criblage des paramètres de nos applications

Le problème que nous nous posons est de savoir quelles variables éliminer et pour quelles raisons. Nous montrerons l'utilisation de la méthode de l'ACP pour le criblage des paramètres des graphes synthétiques. L'ACP réalisée est normalisée, pour pouvoir comparer les variables de différentes natures (temps, unités, octets, etc.).

Dans notre étude les variables sont les paramètres et les individus chaque expérience faite avec les graphes du jeu d'essai. Nous cherchons à garder les variables qui sont indépendantes entre elles et à éliminer les variables qui ont la plus forte corrélation. Par indépendance, il faut comprendre ne représentant pas les mêmes caractéristiques du jeu d'essai. Dire que les variables y et x sont indépendantes implique que la variable y doit avoir, en moyenne, la même valeur quelque soit x et inversement. Le choix va donc se faire par l'élimination des variables orientées dans la même direction qui sont les plus fortement corrélées.

MATRICE DES CORRELATIONS PERMUTEE SUIVANT LE PREMIER AXE DE L'ANALYSE

	CCOM	ARRT	TTAS	PVIR	DATA	LGCH	TSEQ	CPUS	FLOT	GRAN
CCOM	1.00									
ARRT	1.00	1.00								
TTAS	1.00	1.00	1.00							
PVIR	0.89	0.88	0.88	1.00						
DATA	0.51	0.50	0.50	0.85	1.00					
LGCH	0.48	0.49	0.49	0.07	-0.40	1.00				
TSEQ	0.23	0.24	0.24	0.12	0.00	0.26	1.00			
CPUS	0.23	0.24	0.24	0.12	0.00	0.26	1.00	1.00		
FLOT	0.04	0.03	0.03	0.42	0.75	-0.82	-0.09	-0.09	1.00	
GRAN	-0.33	-0.33	-0.33	-0.32	-0.21	-0.21	0.47	0.47	0.10	1.00
	CCOM	ARRT	TTAS	PVIR	DATA	LGCH	TSEQ	CPUS	FLOT	GRAN

TAB. 7.4 – Matrice des corrélations de toutes les variables utilisées pour décrire les applications

La matrice de corrélation de toutes les variables explicatives pour les graphes synthétiques est rapportée dans le tableau 7.4 ; nous observons qu'il existe de fortes corrélations entre les variables à cause de leur mode de calcul. Pour aider l'utilisateur à faire un choix plus juste dans l'élimination des variables, SPADN fournit aussi une seconde matrice (que nous ne rapportons pas ici) avec les tests d'hypothèses sur le coefficient de corrélation.

La valeur propre (ou inertie liée à un facteur) est la variance des coordonnées des individus sur l'axe correspondant. C'est un indice de dispersion du nuage des individus dans la direction définie par l'axe. Dans une analyse normée (nous avons utilisé une analyse normée), la somme des inerties est égale au nombre de variables et donc l'inertie moyenne vaut 1. Si les données sont peu structurées (les variables ne sont pas fortement corrélées entre elles), le nuage a une forme " régulière ". Dans ce cas, les valeurs propres sont régulièrement décroissantes et l'analyse factorielle ne fournira pas de résultats intéressants. Les pourcentages d'inertie des axes définissent les " pouvoirs explicatifs " des facteurs : ils représentent la part de la variance (ou inertie) totale prise en compte par chaque facteur. Le tableau 7.5 montre les valeurs propres. Nous observons que les trois premiers axes représentent 94.18 % des individus sont expliqués. Cela pourra nous donner une idée que : les variables sont fortement corrélées et que suffira d'un nombre très petit des axes pour les expliquer.

LES 10 PREMIERES VALEURS PROPRES

NUMERO	VALEUR PROPRE	POURCENT.	POURCENT. CUMULE
1	4.6145	46.15	46.15
2	2.6776	26.78	72.92
3	2.1258	21.26	94.18
4	0.4180	4.18	98.36
5	0.1259	1.26	99.62
6	0.0358	0.36	99.98
7	0.0012	0.01	99.99
8	0.0011	0.01	100.00
9	0.0000	0.00	100.00
10	0.0000	0.00	100.00

TAB. 7.5 – Les 10 premières valeurs propres

La figure 7.1 est le nuage des variables sur les deux premiers axes. Nous observons, comme il était attendu, la forte corrélation entre différentes variables, *CCOM*, *ARRT* et *TTAS*, que nous pouvons comparer puisqu'elles se trouvent proches du cercle de corrélation. Aussi nous observons une corrélation négative entre *LGCH* et *FLOT*.

CPUS et *TSEQ* " semblent " proches dans le graphique, mais comme elles ne sont pas proches du cercle de corrélation, pour les comparer, nous devons voir dans quels axes elles sont les mieux représentées. La table 7.6 rapporte les coordonnés des variables sur les cinq axes principaux et leur corrélation. Notons que *CPUS* et *TSEQ* sont fortement

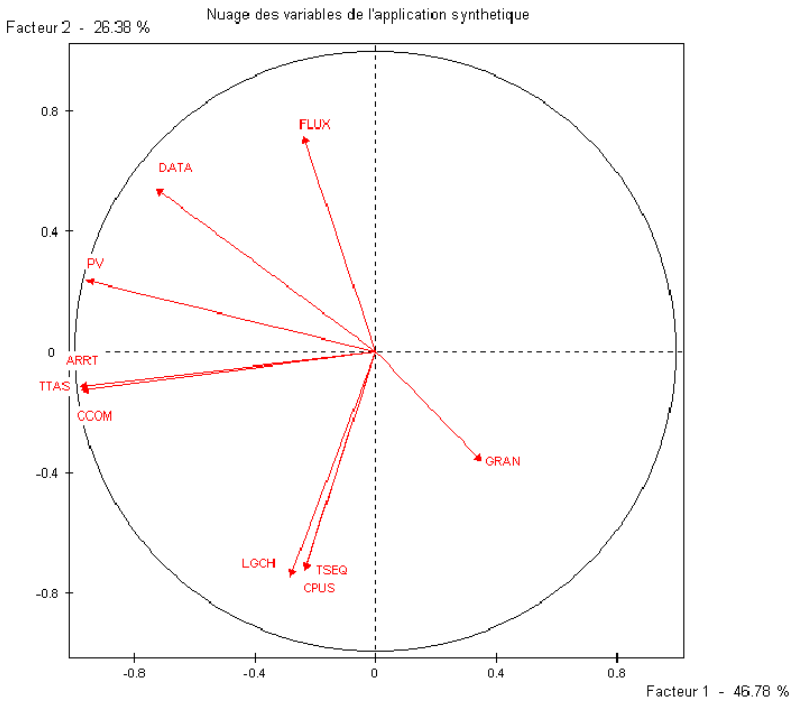


FIG. 7.1 – Nuage des variables des applications synthétiques

corrélées avec les axes factoriels 2 et 3.

COORDONNEES DES VARIABLES SUR LES AXES 1 A 5
VARIABLES ACTIVES

VARIABLES	COORDONNEES					CORRELATIONS VARIABLE-FACTEUR				
IDEN	1	2	3	4	5	1	2	3	4	5
TSEQ	-0.28	-0.66	0.67	0.22	-0.01	-0.28	-0.66	0.67	0.22	-0.01
CPUS	-0.28	-0.66	0.66	0.21	-0.01	-0.28	-0.66	0.66	0.21	-0.01
DATA	-0.65	0.64	0.33	0.09	0.24	-0.65	0.64	0.33	0.09	0.24
FLOT	-0.17	0.78	0.57	0.00	-0.18	-0.17	0.78	0.57	0.00	-0.18
CCOM	-0.98	-0.09	-0.12	-0.11	-0.07	-0.98	-0.09	-0.12	-0.11	-0.07
GRAN	0.33	-0.31	0.72	-0.53	0.06	0.33	-0.31	0.72	-0.53	0.06
TTAS	-0.98	-0.10	-0.12	-0.11	-0.07	-0.98	-0.10	-0.12	-0.11	-0.07
LGCH	-0.36	-0.77	-0.50	-0.05	0.08	-0.36	-0.77	-0.50	-0.05	0.08
PVIR	-0.94	0.30	0.09	-0.02	0.09	-0.94	0.30	0.09	-0.02	0.09
ARRT	-0.98	-0.10	-0.12	-0.11	-0.07	-0.98	-0.10	-0.12	-0.11	-0.07

TAB. 7.6 – Coordonnées et corrélations des variables avec les axes factoriels

La matrice de corrélation et le graphique des nuages des variables sont les deux sources utiles pour procéder à l'élimination des variables " redondantes ", c'est un travail à faire pas à pas. Cela signifie d'abord éliminer une variable, puis appliquer une ACP sur les variables restantes et enfin observer le résultat produit. A chaque pas les choix antérieurs peuvent être remis en question. C'est un procédé empirique dans lequel la bonne connaissance de la nature des variables a un rôle important.

Les variables représentatives pour les applications parallèles

Nous avons procédé à l'élimination des variables et nous sommes arrivés à garder les trois variables suivantes pour représenter notre jeu d'essai :

1. la granularité *GRAN*
2. le flot des données *FLOT*
3. le parallélisme virtuel *PVIR*

Il est intéressant de noter que ces paramètres sont de nature complexe. Un autre aspect que nous voulons noter est que la variable granularité utilisée ici, n'est pas la même que celle utilisée pour calibrer nos applications synthétiques :*GRAN* est de nature aléatoire et dépend des valeurs d'entrée qui peuvent varier d'une application à une autre.

Nous présentons les résultats obtenus après l'ACP sur l'ensemble des mesures effectuées avec les trois variables conservées et nous avons inclus l'indice de performances : l'efficacité. Le tableau 7.7 présente la matrice des corrélations des paramètres gardés pour la description de la charge synthétique et l'efficacité. La dernière ligne du tableau apporte une information globale du lien entre les paramètres et le critère d'évaluation.

MATRICE DES CORRELATIONS

	PVIR	FLOT	GRAN	EFFI
PVIR	1.00			
FLOT	0.42	1.00		
GRAN	-0.32	0.10	1.00	
EFFI	-0.37	-0.45	0.25	1.00
	PVIR	FLOT	GRAN	EFFI

TAB. 7.7 – La matrice de corrélation pour les trois variables conservées : *PVIR*, *FLOT*, *GRAN* et l'indice de performance *EFFI*

Regardons maintenant la répartition des individus sur les axes (voir tableau 7.8). Cette information est indispensable pour interpréter la projection. Les deux premiers axes représentent 75.28 % de la population des mesures.

La figure 7.2 montre le nuage des variables conservées et de l'indice de performance sur les deux axes principaux. Les trois variables conservées sont assez bien représentées sur le cercle unitaire.

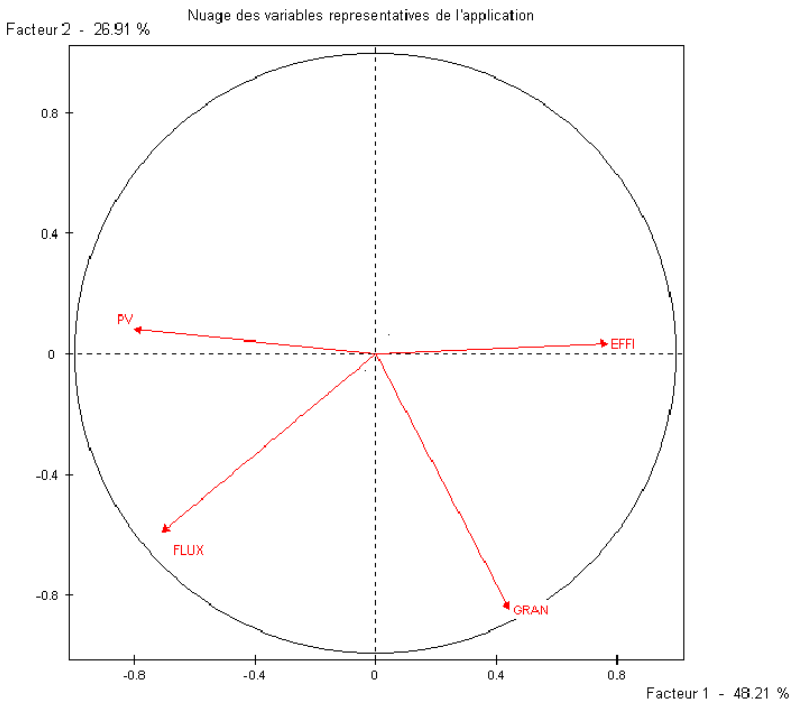


FIG. 7.2 – Nuage des variables des applications synthétiques après criblage

La figure 7.3 est le nuage des individus (nous avons fait une représentation simultanée des deux nuages (celui des variables et celui des individus) mais il faut se garder d'établir une relation de grandeur, le nuage des variables nous sert pour nous indiquer l'information plus importante représentée par chaque axe.

Nous avons inclus la variable nominale *SYNA* qui n'intervient pas dans l'analyse en composantes principales mais qui nous permet de retrouver les applications de charge

NUMERO	VALEUR PROPRE	POURCENT.	POURCENT. CUMULE
1	1.9131	47.83	47.83
2	1.0980	27.45	75.28
3	0.6286	15.71	90.99
4	0.3603	9.01	100.00

TAB. 7.8 – Les quatre premières valeurs propres pour l'ensemble des expériences, après criblage des variables des applications synthétiques

synthétique. Chacune de ses modalités (36 modalités, une pour chaque application synthétique) a une étiquette. Cet identificateur sera placé au centre de gravité de toutes les expériences qui correspondent à la modalité.

Dans le nuage nous avons crée des groupes (quatre). Les groupes B, C et D sont les plus différenciés (éloignés du centre). Le groupe A est proche du centre des axes principaux. Ce sont les groupes plus éloignés qui nous donnent le plus d'information.

Les trois groupes (B, C et D) ont la caractéristique commune d'inclure le même type d'application : série-parallèle.

Dans le groupe B nous retrouvons les applications avec le plus grand nombre de tâches (approx. 1000). Nous remarquons à l'extrême les deux applications 27 et 37 qui ont les pires efficacités. Comme elle ne sont pas dans la direction de l'axe 2, ces applications doivent être décrites par des *PVIR* et *FLOT* grands.

Le groupe C (en-dessous du cercle de corrélation) inclut les applications série-parallèles 26 et 36 avec un granularité d'entrée grande et un total de tâches moyennes (200 approx.).

Le groupe D, le plus éloigné en bas inclut les applications avec une granularité forte, et un petit nombre de tâches (80 approx.).

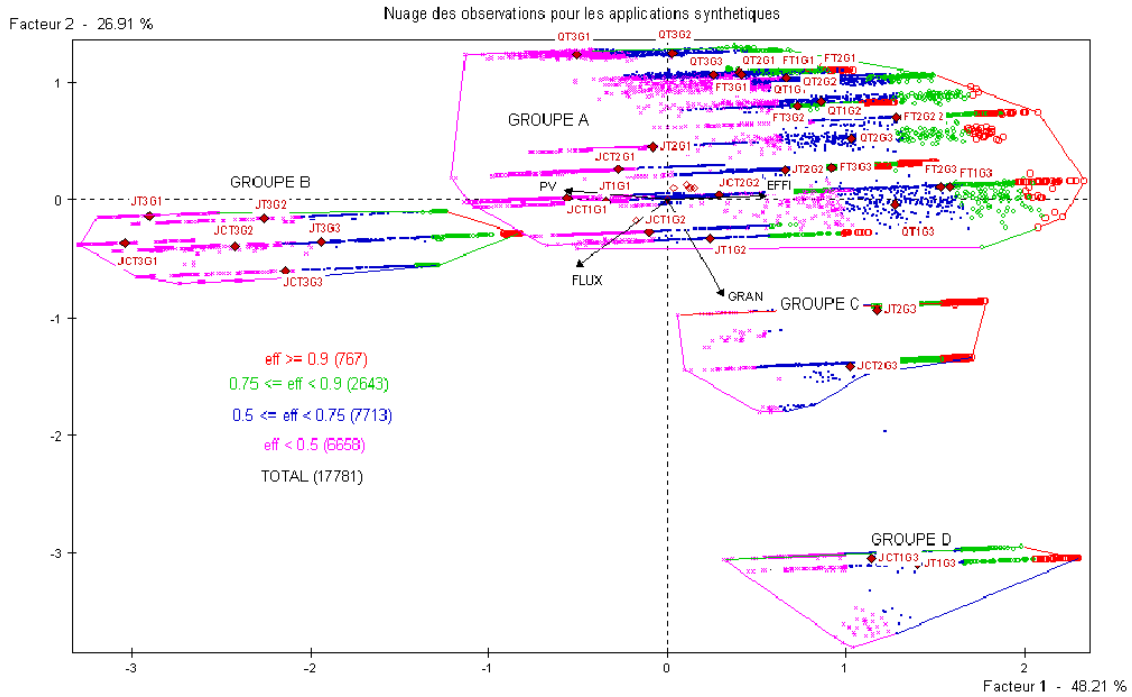


FIG. 7.3 – Nuage des individus

Nous rapportons pour les trois variables conservées les statistiques sommaires (moyenne, écart type, minimum et maximum) pour chaque application synthétique.

***FLOT* :** Pour les applications de type Fusion (1 …9) et Jacobi (21 …29) qui sont nos applications régulières, les valeurs de *FLOT* sont constantes (l'écart type est zéro). Le *FLOT* pour l'application JacobiCreu commence à avoir une variation (nous avions choisi lors de la création de notre jeu d'essai des données réparties aléatoirement entre les fils de la découpe en suivant une loi normale, selon un seuil assez proche des données reçues). Le type d'application avec une plus grande variabilité pour *FLOT* est QuickSort (11 …19).

FLOT					
SYNA	EFFECTIF	MOYENNE	ECART TYPE	MINIMUM	MAXIMUM
AP01	407	60.952	0	60.952	60.952
AP02	407	60.952	0	60.952	60.952
AP03	409	60.952	0	60.952	60.952
AP04	407	70.551	0	70.551	70.551
AP05	407	70.551	0	70.551	70.551
AP06	407	70.551	0	70.551	70.551
AP07	406	83.256	0	83.256	83.256
AP08	391	83.256	0	83.256	83.256
AP09	398	83.256	0	83.248	83.256
AP11	407	89.634	6.242	77.142	111.43
AP12	407	89.277	5.743	76.602	110.484
AP13	405	89.294	6.418	74.819	119.913
AP14	404	61.455	3.226	54.548	71.297
AP15	401	61.26	3.478	52.783	76.59
AP16	402	61.276	3.44	53.964	76.394
AP17	429	30.136	1.096	27.592	34.693
AP18	498	30.053	1.012	27.37	34.759
AP19	459	30.095	1.036	27.845	33.878
AP21	590	505	0	505	505
AP22	623	505	0	505	505
AP23	619	505	0	505	505
AP24	626	325	0	325	325
AP25	599	325	0	325	325
AP26	574	325	0	325	325
AP27	572	645	0	645	645
AP28	595	645	0	645	645
AP29	579	645	0	645	645
AP31	547	488.388	0.023	488.322	488.456
AP32	552	488.389	0.023	488.328	488.467
AP33	552	488.39	0.023	488.333	488.456
AP34	551	401.556	0.419	400.274	402.603
AP35	543	401.612	0.446	400.326	402.851
AP36	550	401.645	0.452	400.383	402.906
AP37	614	740.354	0.146	739.933	740.925
AP38	551	740.348	0.144	739.966	740.697
AP39	493	740.355	0.146	739.955	740.722
ENSEMBLE	17781	327.032	246.927	27.37	740.925

TAB. 7.9 – FLOT pour chaque application

GRAN : C'est pour les applications du type Fusion que l'on observe pour les trois groupes de 3 les différents niveaux de granularité. Pour les autres applications, les trois granularités sont seulement différenciées pour le premier groupe. Quand le nombre de tâches augmente, la granularité diminue et cela est dû au fait que nous n'augmentons pas le travail entre chaque groupe, seulement le nombre de tâches.

GRAN					
SYNA	EFFECTIF	MOYENNE	ECART TYPE	MINIMUM	MAXIMUM
AP01	407	3.755	0.014	3.732	3.836
AP02	407	74.792	0.221	74.624	78.749
AP03	409	186.659	0.136	186.438	188.388
AP04	407	9.987	0.282	9.874	11.258
AP05	407	74.967	2.597	74.021	86.025
AP06	407	187.251	6.759	185.002	215.724
AP07	406	8.951	0.303	8.841	10.139
AP08	391	66.634	1.517	66.264	75.284
AP09	398	166.864	3.855	165.514	188.956
AP11	407	6.532	0.587	5.135	8.384
AP12	407	32.372	2.671	25.125	42.821
AP13	405	201.15	17.71	156.811	265.932
AP14	404	3.666	0.202	3.043	4.236
AP15	401	18.326	1.03	15.209	21.126
AP16	402	113.483	6.622	95.777	135.416
AP17	429	1.397	0.044	1.278	1.562
AP18	498	6.952	0.201	6.38	7.631
AP19	459	42.952	1.242	39.132	46.69
AP21	590	7.297	0.065	7.255	8.601
AP22	623	72.813	0.271	72.623	77.226
AP23	619	605.562	3.884	604.597	690.489
AP24	626	4.698	0.023	4.674	5.105
AP25	599	50.696	0.218	50.578	54.806
AP26	574	278.825	23.077	272.215	390.149
AP27	572	1.559	0.007	1.551	1.709
AP28	595	14.351	1.641	13.91	21.285
AP29	579	54.595	2.509	54.067	83.827
AP31	547	6.524	0.174	6.451	7.766
AP32	552	65.536	3.283	64.534	88.062
AP33	552	599.247	28.557	591.034	741.752
AP34	551	5.569	0.513	5.32	8.602
AP35	543	53.123	2.573	52.21	63.521
AP36	550	332.761	17.091	327.083	439.141
AP37	614	1.651	0.066	1.627	1.993
AP38	551	14.818	0.73	14.594	17.875
AP39	493	57.654	2.941	56.75	68.591
ENSEMBLE	17781	99.696	156.693	1.278	741.752

TAB. 7.10 – GRAN pour chaque application

PVIR					
SYNA	EFFECTIF	MOYENNE	ECART TYPE	MINIMUM	MAXIMUM
AP01	407	18.143	0	18.143	18.143
AP02	407	18.143	0	18.143	18.143
AP03	409	18.143	0	18.143	18.143
AP04	407	31.875	0	31.875	31.875
AP05	407	31.875	0	31.875	31.875
AP06	407	31.875	0	31.875	31.875
AP07	406	102.3	0	102.3	102.3
AP08	391	102.3	0	102.3	102.3
AP09	398	102.3	0	102.3	102.3
AP11	407	22.327	1.135	17.857	25.286
AP12	407	22.449	1.05	18.143	26.143
AP13	405	22.39	1.234	13.857	25.571
AP14	404	47.01	1.827	31.375	51.125
AP15	401	46.783	2.151	31.125	51.625
AP16	402	46.743	2.366	18.375	52.125
AP17	429	134.611	10.131	86.5	157.3
AP18	498	134.839	9.617	80.5	156.5
AP19	459	133.981	8.637	97.5	156.3
AP21	590	16	0	16	16
AP22	623	16	0	16	16
AP23	619	16	0	16	16
AP24	626	40	0	40	40
AP25	599	40	0	40	40
AP26	574	40	0	40	40
AP27	572	200	0	200	200
AP28	595	200	0	200	200
AP29	579	200	0	200	200
AP31	547	18	0	18	18
AP32	552	18	0	18	18
AP33	552	18	0	18	18
AP34	551	35	0	35	35
AP35	543	35	0	35	35
AP36	550	35	0	35	35
AP37	614	190	0	190	190
AP38	551	190	0	190	190
AP39	493	190	0	190	190
ENSEMBLE	17781	73.468	68.002	13.857	200

TAB. 7.11 – *PVIR* pour chaque application

PVIR : Comme nous l'avions établi, seul le Quicksort à un comportement irrégulier dans la création du nombre de tâches (rappelons que le *PVIR* est directement corrélé au nombre total des tâches), pour les autres trois types d'application Fusion, Jacobi et JacobiCreu cette variable est constante.

7.4 Analyse des stratégies d'équilibrage dynamique de charge

Les stratégies de répartition dynamique vont être évaluées entre elles. Nous entendons par-là que l'évaluation consiste à mesurer l'impact de quelques options d'implémentations (paramètres) sur un jeu de programmes test. Cette démarche permet de s'approcher de la " meilleure " paramétrisation possible d'une classe de répartiteurs. La plupart du temps, on constate que les paramètres les meilleurs ne sont pas les mêmes pour tous les programmes testés. C'est pourquoi, la validité d'une méthode dépend de l'existence d'une paramétrisation "satisfaisante" pour tous les jeux de test.

Pour chaque stratégie d'équilibrage de charge, nous avons fait varier différents paramètres, nous sommes intéressés à l'influence (si elle existe) de ces paramètres. Pour la charge d'entrée du système, nous avons maintenant trois paramètres qui représentent le jeu d'essai appliqué pour chaque ordonnanceur : le FLOT de données (*FLOT*) ou rapport entre le total des données transmises et le nombre total des communications, la granularité (*GRAN*) qui a été définie comme le rapport entre le coût de calcul et le coût de communication, et finalement le parallélisme virtuel (*PVIR*) qui est le rapport entre le nombre total de tâches divisé par le chemin critique observé.

Avec seulement ces trois variables pour le jeu d'essai, nous allons procéder à l'analyse des stratégies. Nous montrons l'analyse globale pour une seule des stratégies : l'ordonnanceur *random*. Reprenons le système que nous essayons d'approcher (voir figure 7.4), nous avons des paramètres d'entrée et des observations. Les paramètres d'entrée sont de deux natures différentes pour l'application (*GRAN*, *FLOT* et *PVIR*) et pour l'ordonnanceur (*EVEN, MAXT, JOBT, MRUN*). Est-ce que les paramètres de l'ordonnanceur ont une influence sur l'indice de performance observé (*EFFI*) ? Comment se comporte l'ordonnanceur pour un jeu d'entrée de *GRAN*, *FLOT*, *PVIR* ?

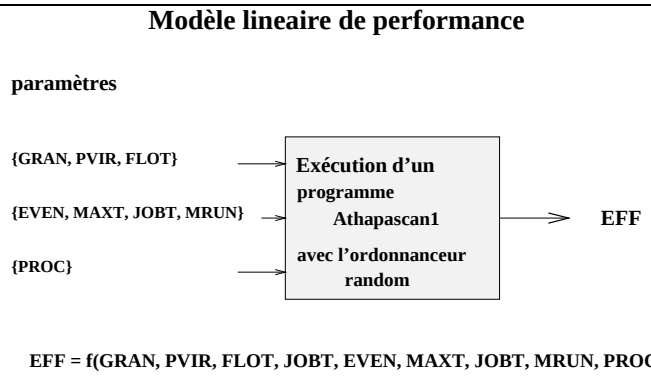


FIG. 7.4 – Schéma de la démarche d'analyse de mesures

Pour cette analyse nous avons utilisé la régression multiple, pour essayer de comprendre l'influence des différents paramètres, nous commençons par l'ordonnanceur *Random*.

7.4.1 L'ordonnanceur *Random*

Pour la stratégie d'équilibrage *Random*, on a fait varier deux paramètres, *MRUN* et *MAXT*. Rappelons que *MRUN* est le nombre de tâches qui peuvent débuter leur exécution de façon concurrente (cela signifie que l'exécuteur lancera jusqu'à *MRUN* processus légers de façon concurrente) et *MAXT* est un seuil, qui est utilisé au moment de la création d'une nouvelle tâche, (le processeur peut décider en fonction de sa charge locale et de la valeur de *MAXT* de garder la tâche pour l'exécuter localement où l'envoyer au contrôleur central pour leur placement.). Les valeurs qui sont prises dans l'expérience ces deux paramètres sont :

- *MRUN* : 1 (le minimun), 2, 5 et 10 processus légers
- *MAXT* : 1, 2, 3 [6]

	PVIR	FLOT	MRUN	MAXT	GRAN	EFFI
PVIR	1.00					
FLOT	0.42	1.00				
MRUN	0.00	0.00	1.00			
MAXT	0.01	0.00	-0.01	1.00		
GRAN	-0.32	0.10	0.00	0.00	1.00	
EFFI	-0.32	-0.50	0.02	0.10	0.18	1.00
	PVIR	FLOT	MRUN	MAXT	GRAN	EFFI

TAB. 7.12 – Matrice de corrélation pour l'ordonnanceur *Random*

Avec ces deux paramètres et les variables pour les applications synthétiques, nous avons appliqué une analyse en composantes principales pour voir si nous n'avions pas d'informations redondantes. La matrice de corrélation est rapportée (voir tableau 7.12) . Nous notons qu'il n'existe pas de corrélation directe entre les variables de l'ordonnanceur et de l'application synthétique. La dernière ligne correspond à la corrélation entre les différents paramètres et l'indice de performance : l'efficacité. Le *PVIR* et le *FLOT* ont une corrélation négative avec l'efficacité, la granularité une corrélation positive et les deux paramètres de l'ordonnanceur ont très peu de corrélation avec l'indice de performance.

Nous rapportons le nuage des variables dans les axes 1 et 2 (voir figure 7.5). Comme la projection des paramètres de l'ordonnanceur sur les deux axes principaux est trop petite, nous ne pouvons rien dire sur son effet sur l'efficacité.

Notre problème suivant est un problème de régression. Nous voulons savoir comment les différentes variables modifient l'efficacité. Nous allons appliquer la corrélation multiple.

[6] A cause du recodage, et pour éviter de donner un niveau égal à zéro, nous avons changé ces valeurs, les valeurs originales 0, 1 et 2. La valeur zéro signifie que toutes les tâches devrons être envoyées au contrôleur et que les placements locaux ne sont pas permis.

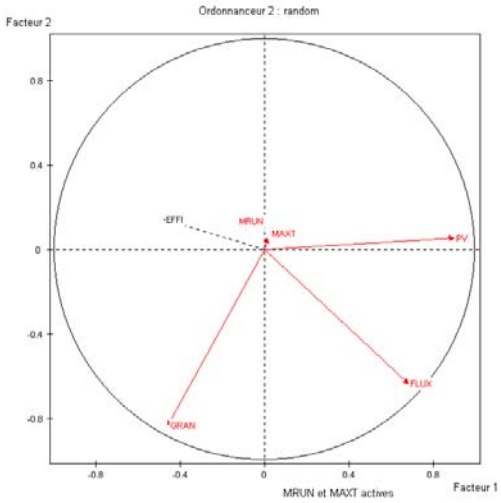


FIG. 7.5 – Nuages des variables pour l'ordonnanceur *Random*

Un modèle linéaire

Nous ne connaissons pas la loi que suit le phénomène et nous essayons de l'approcher par un modèle linéaire, qui est le plus le simple et qui nous donnera l'information sur les effets principaux des variables utilisées comme explicatives [7]. Le modèle est :

$$\begin{aligned} eff_i =& \alpha_0 + \alpha_1 * MAXT_i + \alpha_2 * MRUN_i \\ &+ \alpha_3 * FLOT_i + \alpha_4 * GRAN_i + \alpha_5 * PVIR_i + \varepsilon_i \end{aligned}$$

Nous rapportons l'analyse faite par SPADN (voir tableau 7.13). Nous avons utilisé le modèle linéaire (V16 = V6 + V10 + V17 + V19 + V22), le nom de chaque variable correspond au nom interne que SPADN a donné aux variables de la base de données.

[7] Nous avons essayé d'appliquer un modèle quadratique. En principe ce le modèle nous aurait permis d'établir en plus des effets principaux, les interactions entre les variables ; mais un problème technique dû à SPAD ne nous a pas permis de l'utiliser. SPADN est un outil très puissant pour l'analyse multidimensionnelle, surtout dans les secteurs d'agronomie, biologie, et économie. Il n'était pas prévu tout à fait pour des problèmes où les valeurs des variables peuvent varier en ordre de grandeur de manière considérable. Le problème est que nous avons des valeurs qui peuvent être très petites, de l'ordre des microsecondes avec une précision de 6 décimales (pour les temps parallèles, par exemple) et des valeurs très grandes, comme le nombre de données transmises ou le total des opérations. Le service de SPADN nous a fait des modifications sur le traitement des bases de données, pour la prise en compte des décimales pour les statistiques de base. Cependant il faudra faire encore des modifications pour l'ensemble de l'outil pour une meilleure précision.

```
ANALYSES DE VARIANCE ET REGRESSIONS MULTIPLES
MODELE  1
DEFINITION
:----- MODELE
V16 = V6 + V10 + V17 + V19 + V22
STATISTIQUES SOMMAIRES SUR LES VARIABLES DU MODELE
STATISTIQUES SUR LES   6 VARIABLES CONTINUES
+-------------------------------------------------+-----------+-----------+
|       VARIABLE                       EFFECTIF   |   MOYENNE |   MINIMUM |
| NUM . IDEN - LIBELLE                   POIDS    | ECART-TYPE|   MAXIMUM |
+-------------------------------------------------+-----------+-----------+
|   6 . MAXT - MAXT                      1302     |      2.01 |      1.00 |
|                                        1302.00  |      0.82 |      3.00 |
+-------------------------------------------------+-----------+-----------+
|  10 . MRUN - MRUN                      1302     |      4.51 |      1.00 |
|                                        1302.00  |      3.50 |     10.00 |
+-------------------------------------------------+-----------+-----------+
|  16 . EFFI - EFFI                      1302     |      0.54 |      0.08 |
|                                        1302.00  |      0.20 |      0.94 |
+-------------------------------------------------+-----------+-----------+
|  17 . FLOT - FLOT                      1302     |    289.17 |     27.74 |
|                                        1302.00  |    247.02 |    740.61 |
+-------------------------------------------------+-----------+-----------+
|  19 . GRAN - GRAN                      1302     |     95.12 |      1.30 |
|                                        1302.00  |    147.87 |    608.67 |
+-------------------------------------------------+-----------+-----------+
|  22 . PVIR - PVIR                      1302     |     71.97 |     16.00 |
|                                        1302.00  |     65.57 |    200.00 |
+-------------------------------------------------+-----------+-----------+

ESTIMATION / COEFFICIENTS
AJUSTEMENT DES MOINDRES CARRES (AVEC TERME CONSTANT)
 1302 INDIVIDUS,  6 PARAMETRES (CONSTANTE EN QUEUE).
  IDEN                    COEFFICIENT  ECART-TYPE STUDENT  PROBA. V.TEST
                                                     1296
CRITERE(S)
  MAXT                       0.0250       0.006   4.495   0.000   4.48
  MRUN                       0.0012       0.001   0.921   0.357   0.92
  FLOT                      -0.0004       0.000  19.140   0.000 -17.96
  GRAN                       0.0003       0.000   8.514   0.000   8.40
  PVIR                      -0.0001       0.000   1.548   0.122  -1.55
  CONSTANTE                  0.5817       0.015  38.183   0.000  31.25

TEST D'AJUSTEMENT GLOBAL
SOMME DES CARRES DES ECARTS ..... SCE =           34.6873
COEFFICIENT DE CORRELATION MULTIPLE ... R   =       0.5611   R2 =       0.3148
VARIANCE ESTIMEE DES RESIDUS ...... S2  =            0.0268   S  =       0.1636
TEST DE NULLITE SIMULTANEE DES COEFFICIENTS DES  5 VARIABLES :
   FISHER =   119.076           DEG.LIB =  5 1296
   P.CRIT =    0.0000           V.TEST  =   21.60
ANALYSE DE LA VARIANCE (SANS EFFET DE REPETABILITE)
NB. AJUSTEMENT DES MOINDRES CARRES SUR LE MODELE RENDU  DE
    PLEIN RANG  PAR LES  CONTRAINTES :  SOMMES  NULLES DES
    COEFFICIENTS.  LES  SOMMES DE  CARRES CORRESPONDENT  A
    L'ANNULATION SUCCESSIVE DE GROUPES DE COEFFICIENTS SUR
    LE MODELE DE PLEIN RANG
SOURCE                     SOMME      FISHER  DEGRES DE    PROBA.   VALEUR
                         DES CARRES             LIBERTE    CRITIQUE   TEST
ECARTS RESIDUELS            34.687              1296
```

TAB. 7.13 – La régression multiple pour l'ordonnanceur *Random*

La première partie correspond aux statistiques sommaires des variables du modèle. Décrivons les différentes colonnes pour l'ajustement du modèle. La première colonne donne les variables, la deuxième colonne donne les valeurs des coefficients calculés. Si nous considérons seulement ces deux informations, le modèle linéaire pour l'ordonnanceur *Random* s'écrit sous la forme :

$$\begin{aligned} eff_i =& 0.0250 * MAXT_i + 0.0012 * MRUN_i - 0.0004 * FLOT_i \\ &+ 0.0003 * GRAN_i - 0.0001 * PVIR_i + 0.5817 \end{aligned} \tag{7.1}$$

La constante nous semble très grande mais la littérature [Leb95] assure que l'une des propriétés de la régression multiple est que les estimations des coefficients autres que a_0 sont les mêmes, que les variables soient centrées a priori ou pas. Les variables sont centrées quand $a_0 = 0$ (la constante va prendre une valeur proche de la moyenne de la variable observée).

Mais l'analyse de régression multiple nous permet aussi de distinguer les paramètres qui n'ont pas d'influence (ou qui ne sont pas significatifs) pour le calcul de la variable à expliquer.

La troisième colonne rapporte (s_k) l'estimation de l'écart type du k-ième coefficient de régression a_k. Pour savoir si une variable explicative $\mathbf{x_k}$ a une influence réelle sur la variable à expliquer $\mathbf{y}$, on procède à un test d'hypothèse sur le coefficient de régression α_k. La statistique de Student s'écrit $t = \frac{a_k}{s_k}$ et est nommé variable test (dernière colonne).

L'hypothèse nulle ($\mathbf{H_0}$) est l'éventuelle non-influence qui se traduit par :

$$(\mathbf{H_0}) \quad \alpha_k = 0$$

Si $\mathbf{H}_0$ est vraie, la statistique de Student suit une loi de Student à (n-p) degrés de liberté (quatrième colonne). Soit p_c la probabilité tirée de la distribution de Student correspondant à la valeur t_c prise par t :

$$p_c = P(|t| \geq t_c)$$

Si cette probabilité est jugée " trop faible ", on rejette l'hypothèse ($\mathbf{H}_0$). Dans l'hypothèse d'un tirage au hasard, la valeur test a 95 chances sur 100 d'être comprise dans l'intervalle [-1.96 et +1,96]. Normalement on effectue le test au seuil de confiance 0,05 : si $p_c < 0.05$ on rejette l'hypothèse selon laquelle la variable $\mathbf{x_k}$ n'a pas d'influence réelle (avec moins de 5 chances sur 100 de se tromper) ; alors que si $p_c \geq 0,05$ on ne peut pas rejeter cette hypothèse [Leb95]

En pratique, si dans la dernière colonne (la variable test) nous observons une valeur entre [-1.96, +1.96], nous regardons ce que nous dit la colonne des probabilités. Et si cette valeur est plus petite que 0.05, nous devons rejeter l'hypothèse nulle ce qui veut dire que la variable a une influence. Par exemple pour *MRUN* la valeur test est 0.92, elle se trouve dans l'intervalle [-1.96, +1,96], nous regardons alors la colonne de la probabilité, dans ce cas la probabilité est plus grande que 0.05 et alors nous ne pouvons pas rejeter l'hypothèse nulle. Cela signifie que *MRUN* n'a pas d'effet principal sur le calcul de l'efficacité pour

cet ordonnanceur. La même conclusion est vraie pour la variable *PVIR* ou la valeur test est (-1.55) et la probabilité est 0.122 donc plus grande que 0.05.

Donc notre nouveau modèle après l'élimination des variables dites "non significatives" sera :

$$eff_i = \alpha_1 * MAXT_i + \alpha_2 * FLOT_i + \alpha_3 * GRAN_i$$

La dernière partie de l'estimation des coefficients du modèle (voir table 7.13) correspond à la qualité du modèle proposé : la somme des carrés des écarts (SCE), le coefficient de corrélation multiple (R), le carré du coefficient de corrélation (R2) [8] et la variance des résidus. En principe si nous devons choisir entre deux modèles nous regardons deux valeurs : la somme des carrés des écarts et le carré du coefficient de corrélation multiple, le meilleur modèle est celui qui a minimisé $\sum e_i^2$, et a maximisé R^2.

Nous avons appliqué la régression multiple pour le nouveau modèle (sans *MRUN* et *PVIR*). L'ajustement est :

$$eff_i = 0.0248 * MAXT_i - 0.0004 * FLOT_i + 0.0003 * GRAN_i + 0.5806 \qquad (7.2)$$

Les nouvelles valeurs pour le SCE sont 34.7738 et pour R2 est 0.3131. Ces valeurs ne sont pas très différentes des valeurs obtenues pour le premier modèle (SCE de 34.6873 et un R2 de 0.3148) avec une économie de deux variables. C'est pour cette raison que nous avons choisi ce deuxième modèle.

Interprétation du modèle

Le modèle linéaire que nous proposons est un modèle empirique et nous posons l'hypothèse que ce modèle représente bien le phénomène étudié. Nous sommes conscients que ce modèle est une approximation empirique du phénomène [9]. Nous allons maintenant l'interpréter.

L'**étude du signe** et de la **grandeur des coefficients** nous apportent des informations importantes. Le signe du coefficient d'une variable nous permet de savoir si une variation de la variable influe de manière positive ou négative sur la variable à prédire. Dans le deuxième modèle (equ. 7.2) les variables *MAXT* et *GRAN* influent de manière positive. Cela signifie qu'il est mieux de garder les tâches et de les exécuter localement lors de sa création que de les exporter de manière aléatoire et qu'il est mieux d'augmenter la granularité. Nous mettons en garde que nous ne pouvons extrapoler ces conclusions en dehors du domaine expérimental où notre modèle a été ajusté. La plus grande valeur que nous avons donnée à *MAXT* est 3 (rappelons qu'à cause du recodage de la base de donnée, 1 signifie ne rien garder localement, 2 est au moins une tâche et 3 signifie garder

[8] Cette valeur est aussi reportée comme SCR : ou somme des carrés dus à la Regression. Le R2 décrit le partage de la variance totale en variance " expliquée " et " résiduelle "

[9] Nous devrions valider le modèle, normalement avec de nouvelles expériences, nous n'avons pas eu cette possibilité, pour des raisons techniques et de temps, nous allons avec les risques que cela comporte le considérer valide.

deux tâches au moment de la création.), mais en général il pourra prendre des valeurs plus grandes.

Par contre *FLOT* a un signe négatif. Cela signifie que si nous voulons augmenter l'efficacité, le *FLOT* devrait être fixé à sa valeur minimale. L'ordonnanceur peut contrôler le seuil *MAXT*, mais pas le *FLOT*, parce que *FLOT* est une caractéristique de l'application. Une des fonctionnalités d'ATHAPASCAN1 est de pouvoir prendre en compte des informations (données par le programmeur ou l'ordonnanceur) pour arrêter la découpe des tâches. Quand le nombre de tâches diminue, la granularité va augmenter. Cependant il existe une corrélation entre les paramètres de l'application *GRAN* et *FLOT* (voir table 7.12), si l'un augmente l'autre aussi, si les deux augmentent à la même vitesse nous courons le risque de les neutraliser, puisque ces deux variables ont un coefficient du même ordre de grandeur. Et les valeurs qu'ils peuvent prendre se trouvent aussi dans le même ordre de grandeur (voir la table des statistiques sommaires 7.14).

L'efficacité pour l'ensemble des expériences de l'ordonnanceur *Random* (un total de 1302) est de 0.54. Selon le modèle obtenu, pour avoir une meilleure efficacité, *MAXT* et *GRAN* doivent prendre les valeurs les plus grands possibles. En effet, nous avons filtré les expériences et pris en compte seulement les observations avec un *MAXT* = 3. Cela nous donne un incrément dans la moyenne, avec 0.56 comme moyenne et un écart de 0.2. Nous avons encore filtré les expériences et pris en compte seulement les expériences avec *MAXT* = 3 et une *GRAN* > 100, dans ce cas notre moyenne d'efficacité est de 0.658 avec un écart de 0.161.

Maintenant nous avons envie de chercher les valeurs " optimales " qui nous permettraient de nous approcher d'une efficacité 1 (valeur que nous ne pouvons pas atteindre voir section 2.1).

Notre modèle a trois variables indépendantes *MAXT*, *GRAN* et *FLOT* et une variable dépendante *EFFI*. Si on cherche la valeur optimale pour *EFFI*, i.e. *EFFI* =1, alors notre modèle devient une fonction à trois variables et représente un plan dans une espace cartésien tridimensionnel. La figure 7.6 montre ce plan [10].

[10]Ce graphique a été obtenu avec le logiciel " Octave ".

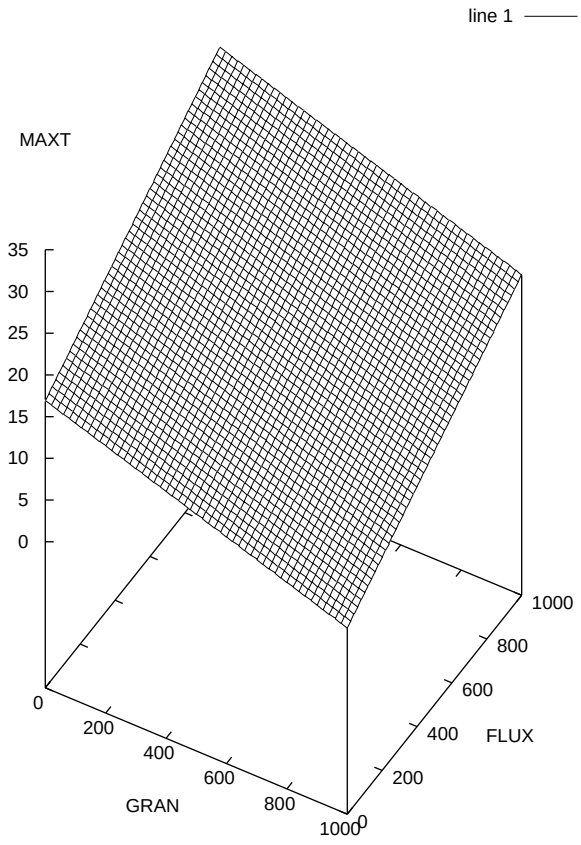


FIG. 7.6 – Modèle linéaire pour l'ordonnanceur *Random* avec *EFFI*=1

Nous pourrions déduire que pour des taches de petite granularité, il est mieux de les garder sur le processeur, surtout si le *FLOT* est grand, et que pour des tâches de plus grande granularité nous pouvons les exporter mais toujours pour des valeurs petites de *FLOT*. Par contre si cette dernière valeur devient grande, il est mieux de les garder sur le processeur qui les a créée.

La régression linéaire que nous avons calculée inclut toutes les expériences faites avec cet ordonnanceur. Nous avons séparé la population en deux sous-groupes, l'un avec les meilleures performances (eff $\geq$ 0.5) et l'autre avec les moins bonnes (eff < 0.5). Dans le tableau 7.14 nous avons rapporté les statistiques sommaires des variables *EFFI*, *FLOT*, *GRAN*, et *PVIR* pour les deux sous-populations.

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 1302					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	1302	0.539	0.197	0.078	0.943
FLOT	1302	289.167	247.017	27.742	740.613
GRAN	1302	95.117	147.870	1.304	608.667
PVIR	1302	71.968	65.569	16	200
Eff >=0.5					
EFFI	786	0.669	0.112	0.500	0.943
FLOT	786	207.426	209.692	28.319	740.591
GRAN	786	102.500	133.356	1.341	606.225
PVIR	786	62.420	53.584	16	200
Eff < 0.5					
EFFI	516	0.340	0.117	0.078	0.499
FLOT	516	413.680	247.593	27.742	740.613
GRAN	516	83.869	166.953	1.304	608.667
PVIR	516	86.512	78.258	16	200

TAB. 7.14 – Les statistiques sommaires de l'efficacité pour l'ordonnanceur *Random*

Une observation est que la valeur moyenne de la granularité est plus petite pour le groupe de moins bonne efficacité. Mais l'aspect le plus important est le rapport entre *GRAN* qui dans le modèle a un signe positif et les deux autres paramètres de l'application *FLOT* et *PVIR* [11] qui ont des signes négatifs. Pour le groupe d'eff $\geq$ 0.5, *FLOT* est deux fois plus grand que *GRAN*, par contre dans l'autre groupe la valeur moyenne de *FLOT* est cinq fois plus grande que *GRAN*. Le fait que les coefficients de *FLOT* et de *GRAN* sont du même ordre de grandeur, comme prévu par le modèle, que l'on obtient la meilleure efficacité quand le rapport entre *FLOT* et *GRAN* est plus petit.

7.4.2 L'ordonnanceur *Modulo Décentralisé*

Rappelons que *modulo* est un algorithme décentralisé, qui ne prend pas en compte la charge des processeurs. Lors de la création d'une nouvelle tâche, le site de réception est le $i \ MODULO \ p$ processeur. L'unique paramètre que nous avons fait varier est *MRUN*.

[11] *PVIR* n'est pas significatif dans le modèle.

Comme pour l'ordonnanceur *Random*, nous avons appliqué la régression multiple. Nous rapportons seulement les résultats (voir tableau 7.15), la première colonne correspond aux variables, la deuxième aux signes, puis les coefficients calculés, la probabilité de Student et finalement la variable test.

Ordonnanceur 3				
	SIG	coeff	probab.	v. test
MRUN	(+)	0.0059	0.022	2.3
FLOT	(-)	-0.0003	0	-6.48
GRAN	(+)	0.0005	0	7.28
PVIR	(-)	-0.0005	0.006	-2.73
cte		0.6089	0	22.05
effect.			moy(eff) = 0.57	

TAB. 7.15 – Modèle de base pour l'ordonnanceur *Modulo Décentralisé*

Pour cet ordonnanceur toutes les variables sont influentes, donc l'ajustement du modèle linéaire pour l'ordonnanceur s'écrit :

$$\begin{aligned} eff_i = {} & 0.005 * MRUN_i - 0.0003 * FLOT_i \\ & + 0.0005 * GRAN_i - 0.0005 * PVIR_i + 0.6089 \end{aligned} \tag{7.3}$$

Interprétation du modèle

Il s'agit d'un modèle empirique, comme le modèle de la stratégie *Random*. D'après les signes, les paramètres qui influent de façon positive sont *MRUN* et *GRAN*, les deux autres *FLOT* et *PVIR* ont une influence négative.

Le modèle nous dit que pour améliorer l'efficacité, il est bien d'augmenter le *MRUN* ou le nombre concurrent de processus légers. L'ordonnanceur modulo choisit le placement des tâches selon un placement cyclique. Chaque processeur a son propre indice et chaque processeur reçoit du travail de tous les autres processeurs. Toutes les tâches prêtes à être exécutées sont gardées dans une liste locale au processeur et l'ordonnanceur va "piocher" dans cette liste pour débuter une tâche. La valeur de *MRUN* permet de débuter une ou plusieurs tâches en même temps. Nous lui avons fixé une valeur maximale pour l'expérimentation qui peut être augmentée. Si les quatre variables étaient complètement indépendantes, pour s'approcher plus rapidement de la valeur maximale de l'efficacité, il serait nécessaire de donner des valeurs minimales à *FLOT* et à *PVIR* et les valeurs maximales à *GRAN* et à *MRUN*.

Nous divisons la population en deux sous-groupes, l'un avec un eff ≥ 0.5 et l'autre avec eff < 0.5. La table 7.16 rapporte les statistiques sommaires pour les paramètres de l'application, la première est pour toutes les expériences, puis pour les deux sous-groupes. Nous observons que pour le groupe de faible efficacité la granularité est encore plus petite que le parallélisme virtuel. Le FLOT de données est 10 fois plus grand que la granularité. Pour l'autre groupe la relation est semblable au groupe de forte efficacité de *random*. En effet notre modèle linéaire explique bien cette situation puisque le coefficient du *GRAN*,

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 429					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	429	0.570	0.216	0.078231	0.980339
FLOT	429	292.074	246.582	28.426	740.585
GRAN	429	95.263	146.458	1.327	605.316
PVIR	429	70.416	64.952	16	200
Eff >=0.5					
EFFI	279	0.699	0.124	0.505	0.980
FLOT	279	241.208	213.956	28.426	740.545
GRAN	279	126.631	160.620	1.363	605.316
PVIR	279	57.628	51.262	16	200
Eff < 0.5					
EFFI	150	0.328	0.124	0.078	0.496
FLOT	150	386.685	273.837	28.848	740.585
GRAN	150	36.917	90.145	1.327	591.456
PVIR	150	94.202	79.427	16	200

TAB. 7.16 – Les statistiques sommaires des groupes pour l'ordonnanceur *Modulo Décentralisé*

FLOT et *PVIR* sont du même ordre de grandeur. Alors la meilleure efficacité est obtenue quand les rapports entre les valeurs de *FLOT* et *GRAN* et entre les valeurs de *PVIR* et *GRAN* sont petits.

7.4.3 L'ordonnanceur *Cyclique Centralisé*

L'algorithme *Cyclique* est un algorithme centralisé très simple, qui ne prend pas en compte la charge des processeurs. Les processeurs peuvent garder des tâches lors de la création et si la charge est supérieure au seuil *MAXT*, la tâche est envoyée au contrôleur central. Le contrôleur va la placer tout de suite (il n'a pas de liste de réserve). Le processeur récepteur est choisit selon la formule $i\ MODULO\ p$, si i est la i-ème tâche à placer et p le nombre de processeurs. Le placement n'est pas remis en cause.

Les paramètres que nous avons fait varier :

1. **MRUN** Nombre de processus légers qui peuvent débuter de façon concurrente.
2. **MAXT** Seuil pour le placement local ou l'exportation.

Ordonnanceur 4				
	SIG	coeff	probab.	v. test
MAXT	(+)	0.0276	0	4.73
MRUN	(+)	0.003	0.028	2.2
FLOT	(-)	-0.0004	0	-16.15
GRAN	(+)	0.0004	0	10.98
PVIR	(-)	-0.0004	0	4.61
cte		0.6158	0	31.55
effect			moy(eff) = 0.58	

TAB. 7.17 – Modèle de base pour l'ordonnanceur *Cyclique Centralisé*

Nous appliquons une régression multiple, et les résultats sont rapportés dans le tableau 7.17. Pour cette expérience tous les paramètres sont significatifs. Donc l'ajustement du modèle linéaire pour cet ordonnanceur est :

$$\begin{aligned} eff_i = &+ 0.0276 * MAXT_i + 0.003 * MRUN_i - 0.0004 * FLOT_i \\ &+ 0.0004 * GRAN_i - 0.0004 * PVIR_i + 0.6158 \end{aligned} \tag{7.4}$$

Interprétation du modèle

L'étude du signe des paramètres montre à nouveau que l'influence de *GRAN* est positive et que les deux autres paramètres de l'application ont une influence négative. *MAXT* et *MRUN* qui sont les paramètres de l'ordonnanceur, doivent prendre les valeurs les plus grandes possibles pour avoir une meilleure efficacité. Nous nous sommes limités dans notre expérience aux valeurs 3 pour *MAXT* et 10 pour *MRUN*, mais ils pourraient prendre des valeurs plus grandes.

L'ordonnanceur est un *Cyclique Centralisé*, avec une certaine liberté pour les processeurs locaux de décider de garder des tâches. Le modèle nous indique que il est mieux de garder les tâches localement que de les envoyer au contrôleur, lequel va les placer de façon cyclique. En effet si le processeur les garde, des communications sont évitées : celles de l'envoi de la tâche de la source au contrôleur central, puis, du contrôleur central vers le récepteur, puis quand le récepteur débute la tâche, la récupération des données de la tâche mère. Autrement dit, cela évite des communications avec le processeur source de la tâche pour récupérer les données et finalement le message de retour des résultats.

La grandeur des coefficients relatifs aux paramètres de l'application est du même ordre. Comme ils sont de signe opposé, les valeurs prises par ces paramètres vont déterminer si ces influences s'annulent mutuellement ou non.

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 1294					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	1294	0.584	0.210	0.077	0.992
FLOT	1294	291.657	246.820	28.041	740.726
GRAN	1294	95.944	148.160	1.314	605.849
PVIR	1294	71.495	65.614	16	200
Eff >=0.5					
EFFI	875	0.702	0.120	0.500	0.992
FLOT	875	222.350	212.478	28.041	740.628
GRAN	875	114.097	148.507	1.362	605.849
PVIR	875	59.219	53.069	16	200
Eff < 0.5					
EFFI	419	0.337	0.128	0.077	0.500
FLOT	419	436.392	250.758	28.710	740.726
GRAN	419	58.036	140.040	1.317	605.646
PVIR	419	97.133	80.265	16	200

TAB. 7.18 – Les statistiques sommaires des groupes pour l'ordonnanceur *Cyclique Centralisé*

Nous avons subdivisé les expériences en deux groupes. Le premier avec l'eff ≥ 0.5 et l'autre avec l'eff < 05. La table 7.18 rapporte les statistiques sommaires pour les groupes. Les résultats sont similaires aux deux ordonnanceurs précédents. La remarque principale à faire sur cet ordonnanceur est qu'il a la plus grande moyenne en efficacité, pour l'ensemble des expériences. Nous ne pouvons pas dire que c'est le meilleur en absolu, puisque le nombre d'expériences fait est un sous-ensemble de toutes les expériences possibles. Cependant cet ordonnanceur a un comportement de bonne qualité.

7.4.4 L'ordonnanceur *Centralisé*

C'est un algorithme centralisé sans réserve, ce qui veut dire qu'au moment de la réception des tâches, le contrôleur va tout de suite les placer. A la différence de l'ordonnanceur cyclique, cette stratégie prend en compte les informations sur l'état de charge des processeurs. Le contrôleur choisit le processeur le moins chargé comme récepteur. Lors de sa création, la tâche peut être placée localement ou envoyée au contrôleur. L'indice de charge est l'indice complexe qui a été décrit dans la section 5.3.2.

Les paramètres que nous avons fait varier sont :

1. **MRUN** Nombre de processus concurrents
2. **MAXT** Seuil pour le placement local ou l'envoi au contrôleur.
3. **EVEN** Seuil pour l'actualisation des informations au contrôleur.

Les résultats de la régression multiple sont rapportés dans le tableau 7.19.

Ordonnanceur 5				
	SIG	coeff	probab.	v. test
EVEN	(+)	0.0006	0.604	0.52
MAXT	(+)	0.0076	0.022	2.3
MRUN	(+)	0.0016	0.041	2.04
FLOT	(-)	-0.0003	0	-26.43
GRAN	(+)	0.0003	0	16.83
PVIR	(-)	-0.0004	0	-8.26
cte		0.6346	0	53.4
effect		3941	moy(eff) = 0.56	

TAB. 7.19 – Modèle de base pour l'ordonnanceur *Centralisé*

Comme la variable test pour *EVEN* est dans l'intervalle [-1.96,+1.96] et que sa probabilité est plus grande que 0.05, la variable *EVEN* peut être considérée comme non significative. Nous procédons à son élimination du modèle, et le nouvel ajustement est rapporté dans la table 7.20.

Donc l'ajustement du modèle linaire pour l'ordonnanceur *Centralisé* est :

$$\begin{aligned} eff_i = &+ 0.0076 * MAXT_i + 0.0015 * MRUN_i - 0.0003 * FLOT_i \\ &+ 0.0003 * GRAN_i - 0.0004 * PVIR_i + 0.6346 \end{aligned} \tag{7.5}$$

Ordonnanceur 5				
	SIG	coeff	probab.	v. test
MAXT	(+)	0.0076	0.022	2.3
MRUN	(+)	0.0016	0.042	2.04
FLOT	(-)	-0.0003	0	-26.44
GRAN	(+)	0.0003	0	16.83
PVIR	(-)	-0.0004	0	-8.26
cte		0.6366	0	56.68
effect		3941	moy(eff) = 0.56	

TAB. 7.20 – Ajustement pour le modèle linaire de l'ordonnanceur *Centralisé* après élimination des variables " non significatives "

Interprétation du modèle

Tout d'abord le paramètre de l'ordonnanceur *EVEN* (qui est considéré comme " non significatif ") est un seuil pour contrôler le nombre de messages à envoyer de chaque processeur vers le collecteur central de l'information. Si *EVEN* a une valeur faible, l'information est actualisée à chaque changement de la charge locale : un message est envoyé au collecteur à chaque événement observé. Plus la valeur d'*EVEN* augmente et plus le nombre des messages diminue, mais l'estimation de la charge globale sera moins exacte. Pour cet ordonnanceur, il semble que ce paramètre n'ait pas d'influence.

Après l'élimination de ce paramètre le modèle est assez semblable à celui de l'ordonnanceur cyclique centralisé à la différence près que les valeurs des coefficients du modèle sont différentes. Les signes sont restés les mêmes. Pour cet ordonnanceur les coefficients sont plus petits (à l'exception pour *PVIR* qui reste le même). Nous avons envie de dire que pour cet ordonnanceur les paramètres de l'application *GRAN* et *FLOT* sont moins dominants mais nous avons un problème de précision [12].

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 3941					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	3941	0.566	0.202	0.076	0.992
FLOT	3941	283.474	247.013	27.370	740.763
GRAN	3941	94.061	149.200	1.282	741.752
PVIR	3941	71.992	64.513	16	200
Eff >=0.5					
EFFI	2497	0.687	0.121	0.500	0.992
FLOT	2497	217.478	210.096	27.370	740.672
GRAN	2497	110.495	142.263	1.353	721.650
PVIR	2497	60.054	52.102	16	200
Eff < 0.5					
EFFI	1444	0.351	0.121	0.076	0.500
FLOT	1444	397.596	263.897	28.025	740.763
GRAN	1444	65.644	156.468	1.282	741.752
PVIR	1444	92.635	77.408	16	200

TAB. 7.21 – Les statistiques sommaires des groupes pour l'ordonnanceur *Centralisé*

[12] L'estimation des coefficients est d'une précision de 4 décimales.

Nous avons procédé à la subdivision des expériences en deux groupes : le premier pour les applications avec une eff ≥ 0.5 et l'autre avec l'eff < 05. La table 7.21 rapporte les statistiques sommaires pour les groupes. Les deux groupes sont différenciés par le rapports entre les valeurs des trois variables de façon semblable aux autres ordonnanceurs.

7.4.5 L'ordonnanceur centralisé avec *Réserve*

Il est semblable à l'algorithme centralisé, sauf qu'il y possède une liste de réserve de tâches, gérée par le contrôleur. Le contrôleur va envoyer des groupes de tâches aux processeurs les moins chargés jusqu'à un seuil (*JOBT*) et puis va attendre de nouvelles modifications de la charge du système pour répartir les tâches de la liste. Le placement est activé par la réception de nouvelles tâches ou de nouvelles informations de la part des processeurs. Si l'ordonnanceur est activé, plus d'un processeur peut recevoir du travail et les processeurs sont ordonnés dans une liste selon leur charge. Le contrôleur va choisir dans cette liste les processeurs dont la charge est plus petite que le seuil *JOBT*. Puis il va répartir les tâches de la réserve. Le nombre de tâches que va recevoir le processeur i dépend de la différence entre *JOBT* et sa charge. Toutes les tâches ne sont pas réparties si les processeurs sont tous chargés. Le contrôleur va attendre de nouvelles modifications pour les répartir.

Les paramètres que nous avons fait varier sont [13]:

1. **MRUN** Nombre de processus légers concurrents à l'exécution.
2. **SMAX** Seuil pour le placement local ou l'exportation.
3. **JOBT** Nombre de tâches que le contrôleur va envoyer au processeur.

Ordonnanceur 6				
	SIG	coeff	probab.	v. test
MAXT	(-)	-0.001	0.699	-0.39
JOBT	(+)	0.0002	0.923	0.10
MRUN	(-)	-0.0049	0	-8.14
FLOT	(-)	-0.0004	0	-33.06
GRAN	(+)	0.0003	0	23.47
PVIR	(-)	-0.0005	0	-13.52
cte		0.7097	0	67.17
effect.		6994	moy(eff) = 0.57	

TAB. 7.22 – Modèle de base pour l'ordonnanceur avec *Réserve*

Les résultats de la régression multiple sont rapportés à la suite (voir tableau 7.22). Pour *MAXT* et *JOBT*, la variable test se trouve dans le seuil [-1.96,+1.96] et leur probabilité est supérieure à 0.05. Nous les avons éliminées une à une (c'est la recommandation

[13]Nous avons observé que ces expériences étaient très sensibles à un paramètre : *EVEN*. Si *EVEN* restait dans sa valeur minimale, aucun problème n'était observé ; par contre quand il était augmenté et lors de certaines combinaisons avec d'autres paramètres, l'application restait bloquée. Nous avons répété les expériences pour ces algorithmes mais toujours avec le même problème. Nous n'avons pas réussit à déterminer la cause de ces blocages ou à prévoir les cas " spéciaux " pour les éliminer de l'expérimentation : nous avons décidé d'éliminer toutes les expériences avec *EVEN* = 2 et 6.

pour l'élimination des variables peu significatives, elles ne doivent pas être éliminées ensemble, puisque le test d'hypothèse est individuel). Nous avons éliminé d'abord *MAXT*, et appliqué une nouvelle régression multiple. Dans les nouveaux résultats, *JOBT* avait toujours une valeur test dans le seuil [-1,96,+1.96] et une probabilité plus grande que 0.05. Nous avons aussi procédé à son élimination. L'estimation des coefficients est rapportée dans la table 7.23.

Ordonnanceur Réserve				
	SIG	coeff	probab.	v. test
MRUN	(-)	0.0049	0	-8.14
FLOT	(-)	-0.0004	0	-33.06
GRAN	(+)	0.0003	0	23.47
PVIR	(-)	-0.0005	0	-13.53
cte		0.7082	0	95.61
effect.		6994	moy(eff) = 0.54	

TAB. 7.23 – Ajustement du modèle linaire de l'ordonnanceur avec réserve après élimination des variables " non significatives "

Donc l'ajustement du modèle linaire pour la stratégie avec réserve est :

$$\begin{aligned} eff_i = & -0.0049 * MRUN_i - 0.0004 * FLOT_i \\ & + 0.0003 * GRAN_i - 0.0005 * PVIR_i + 0.7082 \end{aligned} \tag{7.6}$$

Interprétation du modèle

Après l'étude des signes, le point le plus important à noter est la valeur négative pour *MRUN*, un signe auquel nous ne nous attendions pas et qu'il nous est difficile d'expliquer. Il semble que pour cet ordonnanceur le début d'exécution concurrente de différentes tâches n'est pas bien supporté. Pour expliquer cela, nous proposons l'hypothèse suivante : *EVEN*, qui est le seuil qui contrôle la fréquence des messages échangés pour actualiser l'information de la charge, à cause des problèmes expérimentaux, est resté avec sa valeur minimale. Cela signifie que le collecteur sera informé de tout événement observé dans chaque processeur, et un plus grand nombre de tâches exécutées concurremment implique plus de communications et peut-être un engorgement qui va diminuer l'efficacité. Nous avons séparé la population en deux sous-groupes, l'une avec les meilleures performances (eff $\geq$ 0.5) et une autre avec les moins bonnes (eff < 0.5). Dans le tableau 7.24 nous avons rapporté les statistiques sommaires des variables *EFFI*, *FLOT*, *GRAN*, et *PVIR* pour les deux sous-populations.

Maintenant nous appliquons une régression multiple à chaque sous-groupe (voir table 7.25. On remarque que les deux groupes ont des signes contraires pour *MAXT* et *JOBT*, paramètres qui ont été éliminés dans le modèle global. La grandeur des coefficients est assez importante. Nous pensons que cet ordonnanceur est plus sensible aux changements de paramètres de l'application. Ce que nous pouvons dire est que pour des applications avec une faible granularité, il est mieux de garder les tâches localement mais quand la granularité est plus grande, il est mieux de les exporter.

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 6994					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	6994	0.542	0.216	0.074	0.074
FLOT	6994	387.764	235.361	27.592	27.592
GRAN	6994	106.947	169.033	1.278	1.278
PVIR	6994	77.033	72.394	13.857	13.857
Eff >=0.5					
EFFI	4269	0.684	0.122	0.500	0.995
FLOT	4269	317.411	217.036	28.070	740.722
GRAN	4269	139.321	175.790	1.435	690.489
PVIR	4269	59.069	57.799	16	200
Eff < 0.5					
EFFI	2725	0.321	0.127	0.074	0.500
FLOT	2725	497.979	220.183	27.592	740.925
GRAN	2725	56.228	143.903	1.278	610.5
PVIR	2725	105.177	83.187	13.857	200

TAB. 7.24 – Les statistiques sommaires des groupes pour l'ordonnanceur avec *Réserve*

Ordonnanceur 6			
	SIG	eff >= 0.5	eff < 0.5
MAXT	(-)	-0.0091	0.0073
JOBT	(+)	-0.0046	0.0061
MRUN	(-)	non sign (-)	non sign (-)
FLOT	(-)	-0.0001	-0.0002
GRAN	(+)	0.0001	0.0002
PV	(-)	0.0001	-0.0007
effect		4269	2725
porcent		61.03	38.96

TAB. 7.25 – Modèles des sous-groupes de l'ordonnanceur avec *Réserve*

JOBT change aussi de signe entre les deux groupes, nous pensons que c'est la raison pour laquelle ses coefficients s'annulent dans le modèle global. Nous interprétons le comportement de *JOBT* de la façon suivante : si la granularité est petite, il est bien d'envoyer en groupe sur le même processeur, par contre si la granularité est grande il faut diminuer le nombre de tâches envoyées vers chaque processeur.

La dernière remarque concerne le paramètre *PVIR* : pour le groupe avec une plus grande efficacité, le coefficient a un signe positif. Cela signifie que cet ordonnanceur aura un bon comportement en présence d'applications avec un nombre important de tâches.

7.4.6 L'ordonnanceur *Global*

Il s'agit de l'ordonnanceur avec réserve centrale et avec indice de charge simple : l'indicateur de charge est le nombre de tâches qui ont déjà été exécutées. Dans ce cas, la réserve est seulement activée quand les tâches assignées au processeur sont terminées. L'ordonnanceur envoie *JOBT* tâches au processeur qui a activé la réserve et/ou aux processeurs qui ont une charge moins grande que *JOBT*.

Les paramètres que nous avons fait varier [14] :

1. **MRUN** nombre de processus concurrents en exécution.
2. **MAXT** seuil pour le placement local ou l'exportation.
3. **JOBT** nombre de tâches que le contrôleur va envoyer au processeur.

Ordonnanceur 7				
	SIG	coeff	probab.	v. test
MAXT	(-)	-0.0544	0	-14.56
JOBT	(+)	0.008	0.03	2.18
MRUN	(+)	0.0034	0	3.94
FLOT	(-)	-0.0003	0	23.93
GRAN	(+)	0.0004	0	18.31
PVIR	(+)	0.0003	0	5.83
cte		0.6855	0	47.23
effect		3821	moy(eff) = 0.57	

TAB. 7.26 – Modèle de base pour l'ordonnanceur *Global*

Les résultats de la régression multiple sont rapportés à la suite (voir tableau 7.26). Toutes les variables sont significatives, donc l'ajustement du modèle linéaire est :

$$\begin{aligned} eff_i = & -0.0544 * MAXT_i + 0.008 * JOBT_i + 0.0034 * MRUN_i \\ & - 0.0003 * FLOT_i + 0.0003 * GRAN_i + 0.0003 * PVIR_i + 0.7082 \end{aligned} \tag{7.7}$$

Interprétation du modèle

Suite à l'étude des signes, le modèle nous donne des informations intéressantes. D'abord tout placement local par le processeur lui-même est à éviter (le signe de *MAXT* est négatif et son coefficient est très grand par rapport à la valeur que nous avons donnée à *MAXT* dans l'expérimentation). Les deux autres paramètres de l'ordonnanceur ont un signe positif.

L'autre aspect intéressant est le signe de *PVIR* que nous retrouvons pour la première fois dans le modèle global d'un ordonnanceur. Cet ordonnanceur supporte bien un nombre important de tâches. Cela signifie par exemple qu'il doit bien se comporter de façon générale pour les applications série-parallèle.

Comme dans tous les autres cas, nous avons séparé les expériences en deux sous-groupes, avec les meilleures performances (eff $\geq$ 0.5) dans le premier groupe et dans l'autre les moins bonnes (eff < 0.5). Les statistiques sommaires des variables de l'application sont rapportées dans le tableau 7.27. En plus nous avons calculé comme pour l'ordonnanceur précédent la régression multiple des deux sous-groupes (voir 7.28).

Les deux groupes sont différenciés. Le groupe d'eff $\geq$ 0.5 est décrit par un coefficient positif pour *PVIR* et pour *MRUN* (les autres deux paramètres de l'ordonnanceur ne sont pas significatifs pour le groupe). L'autre groupe a au contraire une valeur négative

[14] Comme pour l'ordonnanceur avec réserve *EVEN* a été fixé à sa valeur minimale.

STATISTIQUES SOMMAIRES DES VARIABLES CONTINUES EFFECTIF TOTAL : 3821					
Paramètres de l'application					
IDEN	EFFECTIF	MOYENNE	ECART-TYPE	MINIMUM	MAXIMUM
Ensemble					
EFFI	3821	0.570	0.211	0.085	0.992
FLOT	3821	289.605	244.958	27.845	740.663
GRAN	3821	95.565	146.781	1.282	609.071
PVIR	3821	69.986	64.775	16	200
Eff >=0.5					
EFFI	2417	0.704	0.122	0.500	0.992
FLOT	2417	240.061	223.798	27.845	740.663
GRAN	2417	111.222	147.630	1.370	606.582
PVIR	2417	65.325	57.182	16	200
Eff < 0.5					
EFFI	1404	0.338	0.107	0.085	0.500
FLOT	1404	374.896	256.086	28.095	740.632
GRAN	1404	68.612	141.302	1.282	609.071
PVIR	1404	78.016	75.420	16	200

TAB. 7.27 – Les statistiques sommaires des groupes pour l'ordonnanceur *Global*

Ordonnanceur 7			
	SIG	eff >= 0.5	eff < 0.5
MAXT	(-)	non sign (-)	-0.022
JOBT	(+)	non sign (+)	non sign (+)
MRUN	(+)	0.0044	non sign (+)
FLOT	(-)	-0.0001	-0.0001
GRAN	(+)	0.0002	0.0002
PV	(+)	0.0003	-0.0003
effect		2417	1405
porcent		63.24	36.76

TAB. 7.28 – Modèles des sous-groupes de l'ordonnanceur *Global*

pour *PVIR* et garde comme paramètre de l'ordonnanceur *MAXT* avec un signe négatif. Le groupe d'eff $\geq$ 5 a un rapport plus faible entre *FLOT* et *GRAN* que l'autre groupe et supporte bien le parallélisme virtuel (*PVIR* positif) au contraire du deuxième groupe (*PVIR* négatif).

7.5 Comparaison des stratégies d'équilibrage de charge

Nous allons présenter deux comparaisons statistiques pour l'ensemble des stratégies, par rapport à chacune des applications synthétiques (36), la première avec l'ensemble des observations et la deuxième après avoir filtré les expériences selon les valeurs données par la régression multiple.

7.5.1 Comparaison avec l'ensemble des mesures

Dans le tableau suivant nous rapportons les moyennes de l'efficacité pour chaque application synthétique pour chacun des ordonnanceurs, ainsi que l'ordonnanceur qui a la

meilleure moyenne et celui qui a la pire moyenne. Nous devons être conscients que les moyennes sont obtenues à partir d'expériences dans lesquelles nous avons fait varier des paramètres pour essayer d'étudier leur influence et comprendre leurs effets. Pourtant, nous pouvons avoir une idée globale sur l'ensemble de l'expérience. Quelques remarques sur ce premier tableau :

1. La première remarque concerne les deux applications, numérotées 27 et 37, qui correspondent respectivement à un *Jacobi* et un *Jacobi Creux* (toutes deux série-parallèles), avec toutes les deux une petite granularité et un grand nombre de tâches (environ 1000 tâches). Pour ces deux applications l'efficacité moyenne pour tous les ordonnanceurs est la plus mauvaise (eff < 0.2).
2. Pour toutes les applications de petite granularité (les numéros 1, 4, 7, 11, 14, 17, 21, 24, 27, 31, 34, 37), la moyenne de l'efficacité est plus basse comparée aux deux autres du groupe. Rappelons que nous avons classé nos applications par groupes de trois (voir définition de *SYNA* 7.2.1).
3. Pour 14 applications, l'ordonnanceur *Random* est le pire, et pour 9 applications c'est l'ordonnanceur *Modulo*. L'ordonnanceur *Random* a la moyenne la plus basse de l'ensemble, (voir dernière ligne.)
4. Pour 12 applications, l'ordonnanceur cyclique centralisé est le meilleur et c'est le cas pour 12 applications pour l'ordonnanceur global.
5. Notons le cas particulier des ordonnanceurs cyclique et global qui, soit ont de bonnes moyennes, soit ont de mauvaises moyennes.
6. Pour les applications de type diviser pour paralléliser (1 …19), le meilleur ordonnanceur est le cyclique, et le pire est l'ordonnanceur modulo.
7. Pour les applications série-parallèle (21 …39), le meilleur est l'ordonnanceur modulo et le pire est le *random*.

TAB. 7.29 – Statistique sommaire de l'efficacité pour chaque stratégie

STATISTIQUES SOMMAIRES DE L'EFFICACITE										
Num	Nom		ORD 2	ORD 3	ORD 4	ORD 5	ORD 6	ORD 7	meill	pire
		POIDS	36	12	36	108	107	108		
1	FT1G1	MOYENNE	0.512	0.444	0.538	0.516	0.525	0.439	ord 4	ord 7
		ECART-TYPE	0.094	0.189	0.126	0.12	0.108	0.168		
		POIDS	36	12	36	107	108	108		
2	FT1G2	MOYENNE	0.675	0.682	0.762	0.703	0.713	0.641	ord 4	ord 7
		ECART-TYPE	0.116	0.144	0.126	0.127	0.135	0.224		
		POIDS	36	12	36	108	109	108		
3	FT1G3	MOYENNE	0.665	0.704	0.77	0.726	0.726	0.654	ord 4	ord 7
		ECART-TYPE	0.133	0.153	0.136	0.131	0.125	0.227		
		POIDS	36	12	36	107	108	108		
4	FT2G1	MOYENNE	0.654	0.628	0.681	0.655	0.659	0.62	ord 4	ord 7
		ECART-TYPE	0.093	0.154	0.116	0.121	0.106	0.168		
		POIDS	36	12	36	108	107	108		
5	FT2G2	MOYENNE	0.727	0.712	0.764	0.732	0.73	0.713	ord 4	ord 3
		ECART-TYPE	0.098	0.115	0.113	0.12	0.119	0.195		
		POIDS	36	12	36	107	108	108		
6	FT2G3	MOYENNE	0.718	0.738	0.772	0.741	0.735	0.725	ord 4	ord 2
		ECART-TYPE	0.124	0.116	0.104	0.121	0.121	0.196		
		POIDS	36	12	36	106	108	108		
7	FT3G1	MOYENNE	0.673	0.681	0.691	0.673	0.672	0.696	ord 7	ord 6
		ECART-TYPE	0.091	0.103	0.088	0.091	0.092	0.088		
		POIDS	36	12	36	104	99	104		
8	FT3G2	MOYENNE	0.733	0.74	0.764	0.745	0.754	0.802	ord 7	ord 2
		ECART-TYPE	0.097	0.111	0.087	0.101	0.111	0.101		
		POIDS	34	12	36	107	101	108		
9	FT3G3	MOYENNE	0.709	0.742	0.772	0.75	0.721	0.806	ord 7	ord 2
		ECART-TYPE	0.115	0.114	0.095	0.107	0.115	0.102		
		POIDS	36	12	36	107	108	108		
11	QT1G1	MOYENNE	0.501	0.469	0.517	0.517	0.528	0.488	ord 6	ord 3
		ECART-TYPE	0.141	0.138	0.141	0.126	0.124	0.164		
		POIDS	36	12	36	107	108	108		
12	QT1G2	MOYENNE	0.576	0.546	0.609	0.589	0.611	0.561	ord 6	ord 3
		ECART-TYPE	0.139	0.122	0.125	0.15	0.138	0.186		
		POIDS	36	12	36	106	107	108		
13	QT1G3	MOYENNE	0.604	0.596	0.646	0.609	0.629	0.592	ord 4	ord 7
		ECART-TYPE	0.138	0.139	0.136	0.147	0.146	0.197		
		POIDS	36	12	36	108	108	104		
14	QT2G1	MOYENNE	0.497	0.451	0.516	0.494	0.508	0.484	ord 4	ord 3
		ECART-TYPE	0.117	0.142	0.109	0.109	0.101	0.125		
		POIDS	35	12	35	107	107	105		
15	QT2G2	MOYENNE	0.616	0.595	0.64	0.608	0.637	0.618	ord 4	ord 3
		ECART-TYPE	0.12	0.151	0.113	0.131	0.125	0.151		
		POIDS	35	12	34	108	106	107		
16	QT2G3	MOYENNE	0.627	0.612	0.665	0.628	0.653	0.651	ord 4	ord 3
		ECART-TYPE	0.128	0.126	0.129	0.14	0.137	0.172		
		POIDS	42	12	35	121	128	91		
17	QT3G1	MOYENNE	0.411	0.395	0.418	0.397	0.398	0.41	ord 4	ord 5
		ECART-TYPE	0.067	0.088	0.061	0.066	0.061	0.061		
		POIDS	42	11	38	161	137	110		
18	QT3G2	MOYENNE	0.605	0.561	0.608	0.584	0.609	0.63	ord 7	ord 3
		ECART-TYPE	0.1	0.092	0.102	0.12	0.105	0.115		
		POIDS	37	11	37	146	119	109		
19	QT3G3	MOYENNE	0.669	0.638	0.679	0.635	0.655	0.701	ord 7	ord 5
		ECART-TYPE	0.119	0.126	0.114	0.13	0.124	0.115		
		POIDS	36	12	36	108	288	110		

continu dans la page suivante

commence dans la page précédente

Num	Nom		ORD 2	ORD 3	ORD 4	ORD 5	ORD 6	ORD 7	meill	pire
21	JT1G1	MOYENNE	0.394	0.431	0.449	0.452	0.466	0.427	ord 6	ord 2
		ECART-TYPE	0.142	0.127	0.141	0.132	0.12	0.142		
		POIDS	36	12	36	108	323	108		
22	JT1G2	MOYENNE	0.534	0.662	0.656	0.617	0.628	0.619	ord 3	ord 2
		ECART-TYPE	0.154	0.114	0.139	0.138	0.139	0.166		
		POIDS	36	12	36	105	323	107		
23	JT1G3	MOYENNE	0.545	0.705	0.693	0.653	0.657	0.641	ord 3	ord 2
		ECART-TYPE	0.168	0.122	0.151	0.151	0.146	0.163		
		POIDS	36	12	36	111	323	108		
24	JT2G1	MOYENNE	0.454	0.485	0.493	0.507	0.501	0.451	ord 5	ord 7
		ECART-TYPE	0.131	0.136	0.127	0.116	0.116	0.129		
		POIDS	36	12	36	108	301	106		
25	JT2G2	MOYENNE	0.658	0.833	0.78	0.755	0.743	0.692	ord 3	ord 2
		ECART-TYPE	0.156	0.067	0.101	0.109	0.122	0.158		
		POIDS	36	12	36	95	288	107		
26	JT2G3	MOYENNE	0.615	0.889	0.667	0.795	0.777	0.717	ord 3	ord 2
		ECART-TYPE	0.199	0.062	0.198	0.113	0.133	0.179		
		POIDS	36	12	36	108	272	108		
27	JT3G1	MOYENNE	0.176	0.174	0.176	0.177	0.187	0.191	ord 7	ord 3
		ECART-TYPE	0.083	0.084	0.084	0.084	0.083	0.077		
		POIDS	36	12	36	95	308	108		
28	JT3G2	MOYENNE	0.39	0.402	0.403	0.379	0.402	0.522	ord 7	ord 5
		ECART-TYPE	0.199	0.215	0.217	0.184	0.219	0.178		
		POIDS	34	11	35	104	304	91		
29	JT3G3	MOYENNE	0.46	0.469	0.487	0.5	0.485	0.643	ord 7	ord 2
		ECART-TYPE	0.242	0.261	0.251	0.264	0.27	0.211		
		POIDS	36	12	36	107	248	108		
31	JCT1G1	MOYENNE	0.348	0.353	0.366	0.368	0.358	0.37	ord 7	ord 2
		ECART-TYPE	0.136	0.134	0.137	0.128	0.113	0.123		
		POIDS	36	12	36	107	252	109		
32	JCT1G2	MOYENNE	0.457	0.52	0.506	0.486	0.487	0.511	ord 3	ord 2
		ECART-TYPE	0.144	0.142	0.14	0.151	0.128	0.144		
		POIDS	36	12	36	108	252	108		
33	JCT1G3	MOYENNE	0.514	0.595	0.583	0.544	0.55	0.557	ord 3	ord 2
		ECART-TYPE	0.159	0.116	0.153	0.152	0.14	0.146		
		POIDS	36	12	36	107	252	108		
34	JCT2G1	MOYENNE	0.442	0.482	0.48	0.476	0.474	0.404	ord 3	ord 7
		ECART-TYPE	0.136	0.136	0.128	0.122	0.117	0.155		
		POIDS	35	12	36	108	246	106		
35	JCT2G2	MOYENNE	0.605	0.725	0.683	0.613	0.65	0.622	ord 3	ord 2
		ECART-TYPE	0.131	0.081	0.104	0.133	0.101	0.137		
		POIDS	36	12	36	107	251	108		
36	JCT2G3	MOYENNE	0.659	0.818	0.766	0.674	0.722	0.655	ord 3	ord 7
		ECART-TYPE	0.144	0.082	0.124	0.162	0.126	0.151		
		POIDS	36	12	36	108	314	108		
37	JCT3G1	MOYENNE	0.17	0.169	0.171	0.172	0.174	0.186	ord 7	ord 3
		ECART-TYPE	0.081	0.082	0.082	0.081	0.081	0.075		
		POIDS	36	12	36	108	252	107		
38	JCT3G2	MOYENNE	0.38	0.39	0.39	0.378	0.374	0.497	ord 7	ord 6
		ECART-TYPE	0.181	0.189	0.192	0.18	0.195	0.155		
		POIDS	36	12	36	106	214	89		
39	JCT3G3	MOYENNE	0.447	0.461	0.462	0.443	0.46	0.562	ord 7	ord 5
		ECART-TYPE	0.204	0.217	0.22	0.21	0.221	0.165		
		POIDS	1302	429	1294	3941	6994	3822		
	ENSEMBLE	MOYENNE	0.539	0.57	0.584	0.564	0.542	0.569	ord 4	ord 2
		ECART-TYPE	0.197	0.216	0.21	0.202	0.216	0.212		

7.5.2 Comparaison après filtrage

Nous avons pris en considération seulement les meilleures valeurs des paramètres des ordonnanceurs. Nous n'avons pas filtré les paramètres de l'application. Rappelons que dans tous les modèles, les variables de l'application ont un rôle important.

1. La première remarque est que tous les ordonnanceurs ont amélioré leur efficacité (voir dernier ligne du tableau), mais pas de façon spectaculaire.
2. Sur les deux applications, 27 et 37, qui correspondent respectivement à un *Jacobi* et à un *Jacobi Creux* les moyennes d'efficacités sont restées assez basses.
3. Pour toutes les applications de petite granularité (les numéros 1, 4, 7, 11, 14, 17, 21, 24, 27, 31, 34, 37) la moyenne continue à être plus basse par rapport aux autres deux de chaque groupe (voir définition de *SYNA* 7.2.1).
4. Pour 18 applications, l'ordonnanceur avec réserve est le pire, et pour 10 applications c'est l'ordonnanceur *Random* qui a la moyenne la plus basse de l'ensemble, (voir dernière ligne).
5. Pour 18 applications, l'ordonnanceur cyclique centralisé est le meilleur et c'est le cas de 9 applications pour l'ordonnanceur modulo et pour l'ordonnanceur avec réserve.
6. Notons le cas particulier des ordonnanceurs avec réserve qui soit ont des bonnes moyennes, soit ont des mauvaises moyennes.
7. Pour les applications de type diviser pour paralléliser (1 …19), le meilleur ordonnanceur est le cyclique centralisé, et le pire est l'ordonnanceur avec réserve.
8. Pour les applications série-parallèles (21 …39), les meilleurs sont l'ordonnanceur avec réserve et l'ordonnanceur cyclique centralisé et le pire est l'ordonnanceur *random*.

La statistique sommaire de l'efficacité pour chaque stratégie après filtrage est reportée dans le tableau suivant.

TAB. 7.30 – Statistique sommaire de l'efficacité pour chaque stratégie, après filtrage

STATISTIQUES SOMMAIRES DE L'EFFICACITE										
Num	Nom		ORD 2	ORD 3	ORD 4	ORD 5	ORD 6	ORD 7	meill	pire
1	FT1G1	POIDS	12	3	3	9	26	3	ord 4	ord 6
		MOYENNE	0.521	0.577	0.588	0.559	0.498	0.567		
		ECART-TYPE	0.098	0.083	0.046	0.095	0.096	0.115		
2	FT1G2	POIDS	12	3	3	8	27	3	ord 4	ord 6
		MOYENNE	0.691	0.685	0.852	0.724	0.67	0.785		
		ECART-TYPE	0.114	0.078	0.088	0.121	0.113	0.136		
3	FT1G3	POIDS	12	3	3	9	28	3	ord 4	ord 6
		MOYENNE	0.71	0.782	0.897	0.77	0.692	0.838		
		ECART-TYPE	0.119	0.157	0.082	0.145	0.106	0.097		
4	FT2G1	POIDS	12	3	3	8	27	3	ord 4	ord 6
		MOYENNE	0.677	0.727	0.797	0.698	0.607	0.677		
		ECART-TYPE	0.076	0.067	0.043	0.115	0.09	0.089		
5	FT2G2	POIDS	12	3	3	9	26	3	ord 4	ord 6
		MOYENNE	0.763	0.783	0.887	0.804	0.66	0.855		

continu dans la page suivante

commence dans la page précédente

Num	Nom		ORD 2	ORD 3	ORD 4	ORD 5	ORD 6	ORD 7	meill	pire
		ECART-TYPE	0.086	0.078	0.059	0.092	0.089	0.059		
6	FT2G3	POIDS	12	3	3	9	27	3	ord 4	ord 6
		MOYENNE	0.775	0.829	0.863	0.814	0.664	0.819		
		ECART-TYPE	0.078	0.077	0.064	0.108	0.096	0.092		
7	FT3G1	POIDS	12	3	3	7	27	3	ord 4	ord 6
		MOYENNE	0.698	0.738	0.791	0.705	0.607	0.706		
		ECART-TYPE	0.069	0.069	0.041	0.088	0.068	0.055		
8	FT3G2	POIDS	12	3	3	8	22	3	ord 4	ord 6
		MOYENNE	0.771	0.783	0.87	0.809	0.675	0.829		
		ECART-TYPE	0.072	0.112	0.059	0.088	0.07	0.063		
9	FT3G3	POIDS	12	3	3	8	25	3	ord 4	ord 6
		MOYENNE	0.756	0.802	0.86	0.835	0.631	0.839		
		ECART-TYPE	0.081	0.102	0.067	0.095	0.087	0.066		
11	QT1G1	POIDS	12	3	3	9	27	3	ord 4	ord 6
		MOYENNE	0.549	0.523	0.611	0.572	0.483	0.57		
		ECART-TYPE	0.114	0.091	0.142	0.115	0.107	0.058		
12	QT1G2	POIDS	12	3	3	9	27	3	ord 4	ord 6
		MOYENNE	0.63	0.614	0.704	0.676	0.549	0.662		
		ECART-TYPE	0.12	0.107	0.112	0.133	0.104	0.077		
13	QT1G3	POIDS	12	3	3	9	27	3	ord 4	ord 6
		MOYENNE	0.68	0.648	0.821	0.695	0.564	0.705		
		ECART-TYPE	0.114	0.164	0.088	0.15	0.101	0.094		
14	QT2G1	POIDS	12	3	3	9	27	3	ord 4	ord 6
		MOYENNE	0.547	0.522	0.616	0.554	0.456	0.546		
		ECART-TYPE	0.085	0.116	0.07	0.104	0.087	0.045		
15	QT2G2	POIDS	12	3	3	9	26	3	ord 4	ord 6
		MOYENNE	0.669	0.692	0.767	0.712	0.552	0.673		
		ECART-TYPE	0.096	0.128	0.095	0.123	0.085	0.047		
16	QT2G3	POIDS	11	3	3	9	26	3	ord 4	ord 6
		MOYENNE	0.668	0.695	0.808	0.707	0.558	0.749		
		ECART-TYPE	0.1	0.118	0.063	0.157	0.098	0.082		
17	QT3G1	POIDS	13	3	4	10	29	3	ord 4	ord 6
		MOYENNE	0.434	0.426	0.465	0.423	0.357	0.425		
		ECART-TYPE	0.051	0.093	0.038	0.054	0.047	0.034		
18	QT3G2	POIDS	19	2	4	14	35	2	ord 4	ord 6
		MOYENNE	0.634	0.57	0.715	0.704	0.523	0.679		
		ECART-TYPE	0.085	0.044	0.065	0.092	0.07	0.03		
19	QT3G3	POIDS	12	2	4	12	31	2	ord 4	ord 6
		MOYENNE	0.692	0.801	0.806	0.761	0.555	0.728		
		ECART-TYPE	0.101	0.08	0.119	0.131	0.073	0.063		
21	JT1G1	POIDS	12	3	3	9	72	3	ord 6	ord 2
		MOYENNE	0.402	0.478	0.445	0.435	0.494	0.461		
		ECART-TYPE	0.141	0.113	0.117	0.132	0.138	0.146		
22	JT1G2	POIDS	12	3	3	9	81	3	ord 6	ord 2
		MOYENNE	0.535	0.628	0.609	0.562	0.703	0.639		
		ECART-TYPE	0.168	0.11	0.113	0.133	0.147	0.184		
23	JT1G3	POIDS	12	3	3	9	81	3	ord 6	ord 2
		MOYENNE	0.552	0.678	0.643	0.597	0.733	0.643		
		ECART-TYPE	0.154	0.111	0.123	0.137	0.158	0.172		
24	JT2G1	POIDS	12	3	3	10	80	3	ord 3	ord 2
		MOYENNE	0.458	0.543	0.501	0.51	0.496	0.524		
		ECART-TYPE	0.122	0.103	0.119	0.117	0.112	0.106		
25	JT2G2	POIDS	12	3	3	9	75	3	ord 3	ord 2
		MOYENNE	0.662	0.854	0.741	0.702	0.792	0.819		
		ECART-TYPE	0.143	0.026	0.083	0.104	0.101	0.078		
26	JT2G3	POIDS	12	3	3	8	72	3	ord 3	ord 2
		MOYENNE	0.607	0.903	0.624	0.746	0.825	0.83		
		ECART-TYPE	0.22	0.004	0.198	0.109	0.106	0.116		

continu dans la page suivante

commence dans la page précédente

Num	Nom		ORD 2	ORD 3	ORD 4	ORD 5	ORD 6	ORD 7	meill	pire
		POIDS	12	3	3	9	68	3		
27	JT3G1	MOYENNE	0.177	0.155	0.153	0.158	0.23	0.19	ord 6	ord 4
		ECART-TYPE	0.084	0.076	0.078	0.078	0.074	0.086		
		POIDS	12	3	3	8	77	3		
28	JT3G2	MOYENNE	0.395	0.287	0.28	0.28	0.678	0.457	ord 6	ord 5
		ECART-TYPE	0.204	0.146	0.148	0.142	0.093	0.202		
		POIDS	12	2	3	9	71	3		
29	JT3G3	MOYENNE	0.47	0.356	0.322	0.316	0.856	0.522	ord 6	ord 5
		ECART-TYPE	0.241	0.179	0.156	0.161	0.054	0.214		
		POIDS	12	3	3	9	62	3		
31	JCT1G1	MOYENNE	0.368	0.409	0.348	0.346	0.379	0.387	ord 3	ord 5
		ECART-TYPE	0.134	0.142	0.127	0.128	0.102	0.149		
		POIDS	12	3	3	9	63	3		
32	JCT1G2	MOYENNE	0.451	0.535	0.489	0.459	0.519	0.506	ord 3	ord 2
		ECART-TYPE	0.144	0.173	0.142	0.152	0.1	0.161		
		POIDS	12	3	3	9	63	3		
33	JCT1G3	MOYENNE	0.515	0.587	0.553	0.519	0.56	0.562	ord 3	ord 2
		ECART-TYPE	0.165	0.128	0.149	0.179	0.121	0.161		
		POIDS	12	3	3	9	63	3		
34	JCT2G1	MOYENNE	0.449	0.55	0.486	0.471	0.464	0.481	ord 3	ord 2
		ECART-TYPE	0.125	0.106	0.123	0.13	0.113	0.129		
		POIDS	12	3	3	9	61	3		
35	JCT2G2	MOYENNE	0.613	0.785	0.688	0.596	0.659	0.713	ord 3	ord 5
		ECART-TYPE	0.129	0.068	0.111	0.146	0.056	0.104		
		POIDS	12	3	3	9	63	3		
36	JCT2G3	MOYENNE	0.664	0.826	0.743	0.633	0.767	0.766	ord 3	ord 5
		ECART-TYPE	0.15	0.058	0.121	0.164	0.096	0.117		
		POIDS	12	3	3	9	78	3		
37	JCT3G1	MOYENNE	0.172	0.151	0.148	0.154	0.216	0.184	ord 6	ord 4
		ECART-TYPE	0.082	0.075	0.076	0.075	0.073	0.084		
		POIDS	12	3	3	9	63	3		
38	JCT3G2	MOYENNE	0.381	0.291	0.283	0.285	0.608	0.44	ord 6	ord 4
		ECART-TYPE	0.184	0.143	0.147	0.148	0.053	0.191		
		POIDS	12	3	3	9	55	2		
39	JCT3G3	MOYENNE	0.445	0.324	0.317	0.316	0.702	0.367	ord 6	ord 5
		ECART-TYPE	0.202	0.155	0.159	0.159	0.025	0.053		
		POIDS	439	105	111	326	1738	105		
	ENSEMBLE	MOYENNE	0.561	0.599	0.615	0.573	0.587	0.616	ord 7	ord 2
		ECART-TYPE	0.2	0.225	0.239	0.225	0.193	0.211		

7.5.3 Commentaires généraux

Maintenant nous allons faire une description globale et comparative des paramètres des applications et des ordonnanceurs par rapport aux modèles linéaires obtenus. Rappelons ici que les paramètres des applications sont *GRAN*, *FLOT* et *PVIR*.

GRAN qui est présent dans tous le modèles, nous donne une information sur le modèle d'application Athapascan. Nous pouvons dire que, comme le modèle de base de programmation est l'appel de procédure a distance, il est important que le coût de calcul de la tâche soit plus grand que le coût de communication. C'est un aspect à considérer lors de la construction des applications ATHAPASCAN1.

FLOT qui est l'autre paramètre de l'application significatif pour tous les ordonnanceurs, a toujours un signe négatif. C'est un autre paramètre qui peut être expliqué par le modèle de base utilisé par Athapascan puisqu'il représente le total des données transmises. Ce paramètre nous indique l'importance de la prise en compte de la localité des

données. Il faudra construire un ordonnanceur qui prenne en compte la localité des données, puisque les ordonnanceurs que nous avons présenté ne le font pas.

PVIR nous soulignons le cas de l'ordonnanceur global où le signe de *PVIR* est positif. Cela signifie que cet ordonnanceur aura un bon comportement avec les applications série-parallèles.

MRUN , qui est un paramètre commun à tous les ordonnanceurs, a une influence différente selon l'ordonnanceur. Pour l'ordonnanceur *Random*, il n'est pas significatif, et pour l'ordonnanceur avec *Réserve*, il a un signe négatif. Pour tous les autres ordonnanceurs il a un signe positif, dans ces cas, il exploite bien la multiprogrammation, puisque ce paramètre décrive le début concurrent des processus légers.

MAXT pour des ordonnanceurs très simples sans réserve centralisée est un paramètre important et positif, cela signifie que pour ces ordonnanceurs, c'est un bon choix de garder les tâches plutôt que les envoyer avec un placement qui n'est pas suffisamment pertinent. Par contre pour l'ordonnanceur avec réserve, il est non significatif et pour l'ordonnanceur global qui a aussi une réserve centralisée, il est même négatif. Ces deux ordonnanceurs sont les plus complexes et font de meilleurs choix du site de placement des tâches.

JOBT le paramètre *JOBT* est présent seulement dans les ordonnanceurs avec réserve, mais il est significatif seulement pour l'ordonnanceur *Global*, de signe positif mais peu influent.

EVEN nous voulons élargir ici la discussion sur le paramètre *EVEN*. Au début de l'expérimentation les ordonnanceurs avec réserve et global avaient ce paramètre. Mais après nous avons dû le fixer à sa valeur la plus basse à cause de problèmes d'expérimentation. Les ordonnanceurs avec *Réserve* et Global ont la caractéristique commune d'avoir une réserve centrale. La distribution des tâches de la réserve est activée par la réception des informations provenant des processeurs. S'il n'y a pas de message, les tâches ne sont pas envoyées. Nous pourrions peut être résoudre ce problème avec une activation temporisée mais à cause de contraintes techniques cela n'a pas été possible jusqu'à présent. [15].

7.6 Conclusions

Dans ce chapitre nous avons analysé les expériences faites sur une machine parallèle réelle pour un ensemble d'algorithmes d'équilibrage de charge. Nous avons séparé l'analyse en deux phases : la première a permis la description de la charge applicative avec un nombre réduit de variables, la deuxième nous a permis d'extrapoler des modèles

[15] En effet, au moment de l'expérimentation, la version que nous utilisions d'ATHAPASCAN0 n'avait pas la possibilité de supporter les interruptions temporisées.

linéaires (en fonction des paramètres des applications et des ordonnanceurs) du comportement de chaque ordonnanceur. L'utilisation du modèle linéaire (après validation) pourra être utilisée dans les cas de stratégies d'équilibrage de charge adaptatives. Pour le cas spécifique d'ATHAPASCAN, qui a la possibilité de prendre en compte des informations pour la découpe récursive d'une tâche ou des informations sur la localité des données, nous envisagons l'utilisation du modèle pour donner des directives plus rationnelles. Par exemple dans le cas où il est possible d'avoir des informations (même partielles) en temps réel sur les valeurs des paramètres des applications, le modèle peut préciser les valeurs optimales à donner aux paramètres de l'ordonnanceur.

Conclusions et Perspectives

Bilan

L'objectif de ce travail était de proposer une méthodologie pour l'évaluation quantitative des stratégies d'équilibrage dynamique de charge dans un système de calcul parallèle. Notre démarche s'est orientée vers la construction d'une plate-forme expérimentale qui inclut : un ordonnanceur modulaire, une bibliothèque d'algorithmes de répartition dynamique de charge, des modèles d'applications parallèles synthétiques, un jeu d'essai (ou charge synthétique) et des éléments pour la prise de traces et leur traitement.

Pour l'expérimentation deux phases doivent être considérées : la planification expérimentale et l'analyse des résultats. Tout problème expérimental est unique et il faut procéder de façon très rigoureuse dans toutes les étapes. La planification est primordiale et l'objectif doit être clairement précisé.

Pour l'analyse des résultats, nous avons dégagé deux problèmes : la description de l'application irrégulière exécutée avec un nombre réduit de variables observées et la détermination de l'influence significative des paramètres des stratégies d'équilibrage de charge. Nous proposons l'utilisation complémentaire de deux méthodes statistiques multidimensionnelles : l'analyse en composantes principales et la corrélation multiple.

Notre objectif n'était pas de proposer un algorithme général mais de donner des éléments pour pouvoir évaluer d'une façon plus objective les performances des algorithmes de répartition dynamique dans un système complexe dédié aux applications parallèles. Pour comprendre le phénomène et établir les limites du domaine expérimental, il est nécessaire d'avoir une connaissance qualitative du problème. Dans notre cas de l'ordonnancement d'une application parallèle où il ne serait pas possible d'avoir des connaissances à priori de l'application, il faut savoir les mesurer.

Nous avons dirigé notre travail vers l'étude des algorithmes d'équilibrage dynamique de charge qui existent dans la littérature. L'aspect le plus important était de dégager les fonctionnalités globales d'un ordonnanceur. Nous avons participé à la construction de l'ordonnanceur du système ATHAPASCAN. Il a été construit entre les couches du système : l'interface applicative ATHAPASCAN-1a et le noyau *multi-threadé* ATHAPASCAN-0.

Pour l'étude quantitative des algorithmes de répartition dynamique de charge, nous avons implémenté des options pour le contrôle des paramètres de l'ordonnanceur, instrumenté l'application pour la prise des traces et, à la fin de chaque exécution, un traitement sur les observations qui décrivent l'application exécutée a été nécessaire. Le jeu d'essai d'applications parallèles synthétiques est inspiré des modèles ANDES [Kit94] mais avec

la différence essentielle d'avoir un caractère dynamique et aléatoire pour simuler des applications irrégulières.

En restant dans un cadre simple, nous avons procédé à l'étude quantitative de différents algorithmes, nous avons étudié 6 ordonnanceurs : deux stratégies décentralisées, *Random* et *Modulo* et quatre ordonnanceurs centralisés que nous avons nommé : *Cyclique*, *Central sans réserve*, *Central avec Réserve* et *Global*. Chacun des ordonnanceurs est décrit par un ensemble de paramètres et nous nous sommes intéressés à établir l'influence de ces paramètres.

Nous avons planifié et fait presque 18000 expériences pour les six ordonnanceurs avec quatre modèles d'applications synthétiques sur une machine parallèle (SP2-IBM). Pour chaque stratégie, nous avons fait varier simultanément ses paramètres selon un plan factoriel à trois niveaux (cela signifie que chaque paramètre prendra trois valeurs différentes). Ce type de plan nous permet d'étudier l'effet principal de chacun des paramètres. Pour chaque modèle, des applications synthétiques, deux paramètres ont été considérés : le nombre de tâches et la granularité, cela nous donne neuf applications pour chaque modèle et donc un total de 36 expériences.

Les méthodes d'analyse de données comme la régression linéaire, les histogrammes, la comparaison des moyennes et des écarts types, des graphiques bidimensionnels sont nécessaires pour la compression et l'interprétation des résultats, mais dans le cas d'un nombre important de variables, ces outils ne sont pas suffisants.

La première phase de l'analyse de nos expériences a été une *analyse en composantes principales (ACP)*. Cette méthode nous a permis de décrire de façon satisfaisante les applications avec un nombre réduit de variables. Cette étape est très importante parce qu'elle nous a permis d'éliminer dans l'analyse les informations redondantes (ou fortement corrélées). Avec cet ensemble réduit de variables pour représenter notre charge applicative nous avons procédé à l'analyse de chaque stratégie d'équilibrage de charge par rapport à ses paramètres. Un modèle linéaire à été obtenu grâce à la technique de régression multiple. Le modèle linéaire a été choisi pour sa facilité d'interprétation des résultats. L'ajustement du modèle nous donne des informations sur l'importance de chaque paramètre dans notre domaine expérimental. En particulier nous avons déterminé si un paramètre était significatif ou non, et dans le premier cas s'il avait une influence positive ou négative sur l'efficacité. Nous remarquons l'importance d'obtenir des modèles à partir des mesures réelles, puisque cela peut permettre de trouver les meilleurs paramètres pour maximiser les performances.

L'interprétation des modèles obtenus nous a permis de comprendre le comportement de chaque ordonnanceur (et des ses paramètres) par rapport à la charge applicative. En effet, nous avons pu noter que la détermination des paramètres appropriés dépend des caractéristiques du programme (granularité des tâches, flux de données, parallélisme virtuel). Les valeurs optimales des paramètres pour une application donnée ne le sont pas systématiquement pour une autre application.

La démarche que nous avons suivie nous permet d'avoir une vision globale du comportement de chaque stratégie dans le domaine expérimental.

Perspectives

Il n'existe, à l'heure actuelle, que peu de travaux sur l'évaluation quantitative des algorithmes d'équilibrage de charge considérant en même temps l'influence de différents paramètres. La plupart des travaux font varier un à un les différents paramètres en laissant fixes les autres facteurs, mais le problème principal est que cette méthodologie ne permet de dégager que des conclusions partielles du phénomène puisque les interactions entre les paramètres sont ignorées.

Parmi les perspectives de notre travail, les plus importantes sont :

Pour l'analyse et l'exploitation des résultats

Tout d'abord, il faut réaliser une validation expérimentale de notre approche. Cela concerne les modèles linéaires que nous avons obtenus (par exemple voir si pour des valeurs de paramètres différentes de l'ordonnanceur, on obtient l'efficacité prévue dans le modèle) et essayer d'utiliser notre approche avec des applications réelles.

Après validation, nous pourrons utiliser le modèle de deux façons, de façon prédictive et pour la recherche des valeurs optimales qui doivent être affectées aux paramètres des stratégies pour optimiser les performances du système. L'utilisation de ces modèles devrait nous permettre la modification rationnelle des valeurs des paramètres et nous envisageons cette modification en cours d'exécution. Il s'agit alors d'adapter les paramètres de l'ordonnanceur selon les performances partielles obtenues.

Les modèles nous ont permis d'établir l'importance des variables qui décrivent l'application, en particulier la granularité. L'utilisation du modèle pourra nous permettre de donner des directives plus rationnelles à l'ordonnanceur d'ATHAPASCAN pour la découpe des graphes *Split-Compute-Merge*.

Pour l'étude de nouvelles stratégies d'équilibrage de charge

Dans le cadre de cette thèse, nous avons étudié des stratégies dans un cadre centralisé. Avec notre approche nous avons obtenu des résultats satisfaisants. Nous aimerions élargir notre approche pour étudier d'autres stratégies, en particulier celles qui prennent en compte la localité des données et le contrôle de la granularité.

Il serait également pertinent de procéder à l'étude des stratégies qui ont un caractère décentralisé et où l'autonomie des nœuds est plus importante (et en conséquence la mise à jour de l'état de la charge plus complexe). Dans ce type d'algorithmes se trouvent les stratégies actives à la réception et ou à l'envoi [WLR93].

Pour la plate-forme expérimentale

Il est important que tout système pour le calcul parallèle offre des outils pour l'évaluation des performances. Les outils que nous avons développés dans le cadre de cette thèse sont intégrés à l'environnement ATHAPASCAN, mais des modifications et adaptations sont essentielles.

Une extension immédiate consiste à porter les applications synthétiques irrégulières sur la nouvelle plate-forme du système ATHAPASCAN, ainsi que développer de nouveaux modèles. La construction du jeu d'essai est essentielle. Nous voulons faire remarquer ici que l'utilisation de la charge synthétique est non seulement utile pour l'évaluation de performances mais nous a été très utile pour la mise au point de l'ordonnanceur du système.

Pour évaluer les performances de nos applications, nous avons instrumenté le code de l'application. Le problème est que pour chaque nouveau modèle, il est nécessaire de faire cette instrumentation. Intégrer la prise de trace dans le noyau exécutif d'ATHAPASCAN nous devrait permettre de limiter leur intrusion dans les performances.

Pour conclure, l'expérience acquise au cours du développement de cette thèse nous a montré la difficulté de travailler sur des machines parallèles et la nécessité d'avoir des environnements de programmation qui permettent à l'utilisateur d'exploiter leur performance.

Bibliographie

[All80] A.O. Allen. Queueing Models of Computer Systems. *IEEE Transactions on Computers*, pages 13–24, 1980.

[Bal91] R. Balter, J.P. Banâtre et S. Krakowiak, editor. *Construction des systèmes d'exploitation répartis*. INRIA, 1991.

[BBBa94] D. Bayley, E. Barszcz, J. Barton, and al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, mars 1994.

[Ber91] B. Bernard, D. Stéve et M. Simatic. Placement et Migration de Processus dans les Systèmes Répartis Faiblement Couplés. *Technique et Science Informatique*, 10(5):375–392, 1991.

[Ber97] P.E. Bernard. *Parallélisation et multiprogrammtion pour une application irrégulière de dynamique moléculaire opérationnelle*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et de Calcul de l'IMAG, 1997.

[BGP96] J. Briat, I. Ginzburg, and M. Pasin. Athapascan-0b User Manual. Technical report, LMC-IMAG, 1996.

[BGR96] J. Briat, T. Gautier, and J.L. Roch. On Line Scheduling. In *Parallel Programming Environments for High Performance Computing*, pages 95–108, 1996.

[BL94a] R.D. Blumofe and C.E. Leiserson. Scheduling Multithread Computation by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science*, pages 256–368, Santa Fe, New Mexico, November 1994.

[BL94b] R. Butler and E. Lusk. Monitors, Message and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20:547–564, 1994.

[Bok81] S.H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, 30:207–214, 1981.

[Bou94] P. Bouvry. *Placement de tâches sur ordinateurs parallèles à mémoire distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et de Calcul de l'IMAG, 1994.

[Bou98] E. Bouzgarrou. *Parallélisation de la methode du Branch and Cut pour resoudre le probleme du voyageur de commerce*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et de Calcul de l'IMAG, 1998.

[Bra90] B. Braschi. *Principes de base des algorithmes d'ordonnancement de liste et affectation de priorités aux tâches*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et de Calcul de l'IMAG, 1990.

[Bri95] J. Briat, M. Christaller et M. Rivière. Athapascan-0 : vers une expression du parallélisme à l'aide de décompositions en procédures parallèles et procédures barrière. In *Actes de RenPar'7*, pages 217–220, 1995.

[Bri96] J. Briat, T. Gautier et J.L. Roch. Application irrégulière et ordonnancement en ligne. In *Placement dynamique et répartiotion de charge: application aux systèmes parallèles et répartis*, pages 81–105, 1996.

[Cal94] R.E. Callan. *Building Object-Oriented Systems*. Computational Mechanics Publications, Boston, 1994.

[Cal95] C. Calvin. *Minimisation du sur-coût des communications dans la parallélisation des algorithmes numériques*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire de Modélisation et de Calcul de l'IMAG, 1995.

[Car88] J. Carlier et P. Chrétienne. *Problèmes d'ordonnancement*. MASSON, 1988.

[CDR98] G.H. Cavalheiro, Y. Denneulin, and J.L. Roch. A General Modular Specification for Distributed Schedulers. In *Europar'98*, Southampton, England, 1998. Springer.

[CG72] E.G. Coffmann and R.L. Graham. Optimal Scheduling for Two-Processor Systems. *Acta Informatica*, 1:200–213, 1972.

[CGMS94] N.J. Carrero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The Linda Alternative to Message-Passing Systems. *Parallel Computing*, 20:633–655, 1994.

[Chr94] P. Chrétienne. Ordonnancement et parallélisme. In *Ecole d'ete parallélisme du CNRS*, 1994.

[Chr96] M. Christaller. *Athapascan-0: vers un support exécutif pour applications parallles irrégulières efficacement portables*. PhD thesis, Université Joseph Fourier, Grenoble I, 1996.

[CK88] T. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[CL95] M. Cosnard and M. Loi. Automatic Task Graph Generation Techniques. *Parallel Processing Letters*, 5:527–538, 1995.

[CMKG99] H. Cai, O. Maquelin, P. Kakulavarapu, and G.R. Gao. Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-Grain Multithreaded Execution Model. In *MTEAC 99*, pages 1–10, 1999.

[Col95] J.N. Collins. *Un environnement pour la programmation distribuée à grain indéterminé*. PhD thesis, Univerité de Mons-Hainaut, Belgium, 1995.

[CRCG95] M.R. Castañeda Retiz, M. Christaller, and T. Gautier. Control Parallelism on Top of PVM: The Athapascan Environment. In *EURO-PVM 95*, 1995.

[CRHW94] R. Calkin, R.Hempel, H.C. Hoppe, and P. Wypior. Portable Programming With the PARMACS Message-Passing Library. *Parallel Computing*, 20:615–632, 1994.

[CT95] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.

[Dag96] P. Dagnelie. *Plans d'expériences. Applications à l'entreprise*. October 1996.

[Den95] Y. Denneulin et R. Namyst. PM2 : Parallel Multithreaded Machine. un support d'exécution pour applications irrégulieres. In *RenPar'7*, pages 208–212, 1995.

[EG89] K. Efe and B. Groselj. Minimizing Control Overheads in Adaptative Load Sharing. In *9th. Internation Conference on Distributed Computing Systems*, pages 307–315, 1989.

[ELZ86] D.L. Eager, E.D. Lazwska, and J. Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptative Load Sharing. *Performance Evaluation*, 6:53–68, 1986.

[FK94] J. Flower and A. Kolawa. Express is Not Just a Message Passing System. current and future directions in Express. *Parallel Computing*, 20:597–614, 1994.

[Fly72] M.J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[Fol92] B. Folliot. *Méthodes et outils de partage de charge pour la conception et la mise en œvre d'applications dans le systèmes répartis hétérogénes*. PhD thesis, Université Paris VI, 1992.

[Fon94] C. Fonlupt. *Distribution dynamique de données sur machines SIMD*. PhD thesis, Université des Sciences et Technologies de Lille, 1994.

[Fos95] I.T. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[Fri89] S. Frida et G. Pujolle. *Modèles de systèmes et de réseaux. Performance*, volume 1. Eyrolles, Paris, 1989.

[FS95] A. Ferrari and V.S. Sunderam. TPVM: Distributed Concurrent Computing With Lightweight Process. *IEEE High Performance Distributed Computing*, 4:211–218, 1995.

[FST96] I.T. Foster, C. Skesselman, and S. Tueckle. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[FWR+95] H. Franke, C.E. Wu, M. Riviere, P. Pattnaik, and M. Snir. MPI Programming Environment for IBM SP1/SP2. In *International Conference on Multimedia Computing and Systems*, pages 127–135, 1995.

[FZ86] D. Ferrari and S. Zhou. A Load Index for Dynamic Load Balancing. In *Fall Joint Computer Conference*, pages 684–690, Dallas Texas, 1986.

[FZ87] D. Ferrari and S. Zhou. An Empirical Investigation of Load Indices for Load Balancing Appplications. *Performance Evaluation*, 1987.

[Gin97] I. Ginzburg. *Athapascan-0b: Intégration efficace et portable de multiprogrammation légére et de communications*. PhD thesis, Institut National Polytechnique de Grenoble, 1997.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability : A guide to the Theory of NP-Completeness*. W. H. Freman and Company, New York, 1979.

[GRCD98] F. Galilee, J.L. Roch, G.H. Cavalheiro, and M. Doreille. Athapacan-1: Online Building Data Flow Graph in a Parallel Language. In *Pact '98*, Paris, France, 1998.

[Gui95] F. Guinand. *Ordonnancement avec communicatins pour architectures multiprocesseurs dans divers modèles d'exécution*. PhD thesis, Institut National Polytechnique de Grenoble, 1995.

[HB95] M. Haines and W. Böhm. An Initial Comparison of Implicit and Explicit Programming Styles for Distributed Memory Multiprocessors. In *28th Annual Hawaii International Conference on System Sciences*, pages 379–389, 1995.

[HCAL89] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling Precedence Graphs in Systems With Interprocessor Communication Times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.

[Hém94] F. Hémery. *Etude de la repartition dynamique d'activités sur architectures décentralisées*. PhD thesis, Université des sciences et Technologies de Lille, 1994.

[Hig93] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[Jac96a] C. Jacqmot. *Load Management in Distributed Computing Systems : Towards Adaptative Strategies*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgique, 1996.

[Jac96b] C. Jacqmot et E. Milgrom. Evaluation empirique des performances d'un système informatique: application à l'équilibrage de charge. In B. Folliot, G. Bernard, J. Chassin de Kergomeaux et C. Roucairol, editor, *Placement Dynamique et Répartition de Charge : application aux système parallèles et répartis*, pages 127–139, Presqu'île de Giens, 1996. École Française de Parallélisme, Réseaux et Systèmes.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[JM94] C. Jacmot and E. Milgrom. A Systematic Approache to Load Distribution Strategies for Dstributed Systems. In *International Conference on Decentralized and Distributed Systems ICDDS'93*, pages 15–17. Institut National Polytechnique de Grenoble, 1994.

[Jug96] M. Juganaru, L. Carraro et I. Sakho. Une analyse probabiliste de la complexité en temps d'un algorithme de régulation de charge. In B. Folliot, G. Bernard, J. Chassin de Kergomeaux et C. Roucairol, editor, *Placement Dynamique et Répartition de Charge: application aux systémes parallèles et répartis*, pages 213–218, Presqu'ile de Giens, 1996. École Française de Parallélisme, Réseaux et Systèmes.

[Kan96] S.E. Kannat. *Régulation dynamique de charge dans les systèmes logiques parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, 1996.

[Kit94] J.P. Kitajima. *Modèles Quantitatifs d'Algorithmes Parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, 1994.

[KL87] P. Krueger and M. Livny. A Comparison of Preemptive and Non-Preemptive Load Distributing. In *8th. International Conference on Distributed Computing Systems*, September 1987.

[Kon97] J.C. Konig et J.L. Roch. Machines virtuelles et techniques d'ordonnancement. In *Icare '97*, pages 95–123, 1997.

[KP94] J.P. Kitajima and B. Plateau. Modeling Parallel Program behaviour in *ALPES*. *Information and Software Technology*, 36:457–464, 1994.

[Kun91] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[Leb95] L. Lebart, A. Morineau et M. Piron. *Statistique exploratoire multidimensionelle*. Dunod, Paris, 1995.

[LK87] F.C. Lin and R.M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, 13:32–38, 1987.

[LM82] M. Livny and M. Melman. Load Balancing in Homogeneous Broadcast Distributed Systems. In *ACM Computer Network Performance Symposium*, April 1982.

[Mai96] E. Maillet. *Le tracage logiciel d'applications paralléles : conception et ajustement de qualité*. PhD thesis, Institut National Polytechnique de Grenoble, 1996.

[McB94] O.A. McBryan. An Overview of Message Passing Environments. *Parallel Computing*, 20:417–444, 1994.

[McC95] W.F. McColl. Scalable Computing. *Lecture Notes in Computer Science*, 1000:46–61, 1995.

[Mes95] Message Passing Interface Forum, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard*, June 1995.

[ML87] M.W. Mutka and M. Livny. Scheduling Remote Processing Capacity in a Workstation Processor Bank Computing System. In *7th. International Conference on Distributed Computing Systems*, pages 2–9, Berlin, 1987.

[Mon91] D.C. Montgomery. *Design and Analysis of Experiments*. Wiley, 1991.

[Mon96] T. Monteil. *Etude de nouvelles approches pour les communications, l'observation et le placement de tâches dans l'environnement de programmation parallèle LANDA*. PhD thesis, LAAS, 1996.

[MS91] S. Madala and J.B. Sinclair. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Transaction on Parallel and Distribued Systems*, 2(1):105–116, January 1991.

[Mue93] F. Mueller. A Library Implementation of POSIX Threads under UNIX. In *Winter USENIX Conference*, pages 29–41, San Diego, CA, 1993.

[Pa94] B. Plateau and al. Présentation d'APACHE. Rapport APACHE 1, IMAG, Grenoble, December 1994.

[PD96] G.R. Perrin and A. Darte, editors. *The Data Parallel Programming Model: foundations, HPF realisation, and scientific applications*. Springer Verlag, 1996.

[Pie88] P. Pierce. The NX/2 Operating System. In *3rd Conference on Hypercube Concurrent Computers and Application*, pages 284–390. ACM Press, 1988.

[Ree93] D.A. Reed. Preformance Instrumentation Techniques for Parallel Systems. *Performance Evaluation of Computer and Communication Systems*, pages 463–490, 1993.

[Riv97] M. Riviere. *Concepts structurants pour la mise en œuvre d'applications irrégulières : Application au support exécutif parallèle Athapascan0$_{mp}$*. PhD thesis, Institut National Polytechnique de Grenoble, 1997.

[Rou96] P. Rouchon. Plateforme d'évaluation quantitative pour les stratégies de régulation dynamique de charge. Technical report, Rapport de DEA Informatique. INPG-ENSIMAG, June 1996.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[Sch95] D. Schwartz. *Mèthodes statistiques à l'usage des mèdecins et des biologistes*. Flammarion, Paris, 1995.

[Sch96] O. Schiavo. Intégration d'un ordonnanceur dans athapascan-1. Technical report, Rapport de DEA Mathématiques Appliquées, Université Joseph Fourier, June 1996.

[Sni92] M. Snir. Scalable Parallel Computers and Scalable Parallel Codes : from Theory to Practice. In *Parallel Architectures and their Efficient Use*, volume LNCS, pages 176–184. Springer-Verlag, 1992.

[SSD+94] A. Skjellum, S.G. Smith, N.E. Doss, A.P. Leung, and M. Morari. The Design and the Evolution of Zipcode. *Parallel Computing*, 20:565–596, 1994.

[ST85] C.C. Shen and W.H. Tsai. A Graph Matching Approach to Optimal Task Assignement in Distributed Computing Systems Using a Minimax Criterion. *IEEE Transactions on Computers*, 1985.

[Ste97] B.O. Stein et J. Chassin de Kergommeaux. Environnement de visualisation de progammes parallèles basés sur le fils d'exécution. In *RenPar'9*, Lausanne, Suisse, 1997.

[Stu88] M. Stumm. The Desing and Implementation of a Decentralised Scheduling Facility for a Workstation Cluster. In *2th. IEEE Conference on Computer Workstations*, pages 12–22, Mars 1988.

[Sun90] V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–339, December 1990.

[Tal91] E.G. Talbi. Un algorithme d'allocation dynamique de processus sur un réseau de transputers. *La lettre du Transputer*, 11:7–20, September 1991.

[Tal94] E.G. Talbi. Allocation dynamique de processus dans les systèmes distribués et parallèles. *Lecture Notes in Computer Science*, September 1994.

[Tal95] E.G. Talbi. Allocation dynamique de processus dans les systèmes distrubués et parallèles. Technical report, LIFL, 1995.

[Tar92] E. Tarnvik. Dynamo: a Portable Tool for Dynamic Load Balancing on Distributed Memory Multicomputers. In *Parallel Porcessing, CONPAR'92*, pages 484–490. Springer, 1992.

[Tho87] A. Thomasian. A Performance Study of Dynamic Load Balancing in Distributed Systmes. In *7th. International Conference on Distributed Computing Systems*, pages 178–184, 1987.

[TL89] M.M. Theimee and K.A. Lantz. Finding Idle Machines in a Workstation-Based Distributed Systems. *IEEE Transactions on Software Engineering*, 15:1444–1458, 1989.

[Val90] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103, 1990.

[WLR93] M.H. Willebeek-Lemair and A.P. Reeves. Strategies for Dynamic Load Balancing on High Parallel Computers. *IEEE Transactions on Parallel and Distribued Systems*, 4(9):979–993, September 1993.

[YG92] T. Yang and A. Gerasoulis. A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors. *Proc. of SHPCC'92*, pages 350–357, 1992.

[Zho88] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.

Étude quantitative des mécanismes d'équilibrage de charge dans les systèmes de programmation pour le calcul parallèle

Résumé :

Cette thèse se concentre sur l'évaluation des performances des mécanismes d'équilibrage de charge. Pour l'utilisation efficace d'une architecture parallèle, il est nécessaire de développer des techniques de régulation de charge appropriées. Nous étudions en détail le problème de l'ordonnancement dynamique d'une application parallèle. Les fonctionnalités d'un ordonnanceur générique sont analysées et son implémentation dans le système Athapascan est décrit. Athapascan est un environnement de programmation pour les applications parallèles irrégulières. La structure de l'ordonnanceur permet l'implémentation de différents algorithmes d'équilibrage de charge.
Pour étudier les différentes stratégies d'équilibrage et comparer leurs performances nous proposons une méthodologie. Nous avons construit des modèles de programmes synthétiques avec un caractère dynamique et aléatoire, à partir desquels nous avons établi un jeu d'essai. Nous avons choisi d'étudier les effets simultanés des différents paramètres des ordonnanceurs et de la charge synthétique. Une planification factorielle a été choisie parce qu'elle permet une vision globale de l'influence des différents paramètres.
Les tests sont effectués sur une machine SP1-IBM. Deux méthodes d'analyse de données multivariée sont utilisées, l'analyse en composantes principales et la régression multiple. L'interprétation des modèles linéaires obtenus permet de comprendre le comportement de chaque ordonnanceur et l'influence de ses paramètres par rapport à la charge applicative.

Mots clés : Calcul parallèle, ordonnancement dynamique, algorithmes d'équilibrage de charge, évaluation de performance.

Quantitative study of load-balancing mechanisms in parallel computing systems

Abstract:

The aim of this thesis is the performance evaluation of the the load-balancing mechanisms. The development of load balancing techniques is necessary in order to obtain an effective use of a parallel architecture. We study the problem of the dynamic scheduling of a parallel application. We start with the analysis of the functionalities of a generic scheduler and its implementation in the Athapascan system. Athapascan is a multithreaded system for parallel applications. The structure of the schedule gives the possibility of different algorithms of load balancing.
We propose a methodology for a quantitative evaluation of the different algorithms of load balancing. We built a benchmark of synthetic algorithms with dynamic and random features. We study the combined effects of all the parameters of the schedule and of the synthetic load. A factorial design has been choosen because it permits a global view of the influence of the different parameters.
Our benchmark has been carried out on a SP1-IBM computer. We used two different methods to analyze our results: the principal components analyses (ACP) and the multilinear regression. The linear models obtained from the analysis give us the possibility of understanding the behavior of every schedule and the influence of its parameters with respect to the synthetic load.

Keywords: Parallel computing, dynamic scheduling, algorithms of load balancing, performance evaluation