50 376
1998
428

NUMERO D'ORDRE :

USTL

ANNEE : 1998

LIFL Laboratoire d'Informatique Fondamentale de Lille

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Yves Denneulin

Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin

Thèse soutenue le 16 janvier 1998, devant la commission d'examen :

Président :	S. TISON	LIFL
Directeur de Thèse :	J.M. GEIB	LIFL
Rapporteurs :	B. FOLLIOT	LIP6
	J.L. ROCH	LMC
	B. PLATEAU	LMC
Examinateurs :	T. MAUTOR	PR/SM
	D. EL BAZ	LAAS
	J.F. MÉHAUT	LIFL

D 030 175740 8

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 03.20.43.47.24 Fax. 03.20.43.65.66

the 2000 OLTF

Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin

Yves DENNEULIN

15 décembre 1997

UNIVERSITÉ DES SC. ET TECHN.
BIBLIOTHÈQUE UNIVERSITAIRE
LILLE

Table des matières

1 Introduction **11**
1.1 ESPACE 11
1.2 PM^2 11
1.3 Contributions de la thèse et organisation du document 12
1.4 Caractérisation de l'irrégularité 13
1.4.1 Caractérisation de l'irrégularité 13
1.4.1.1 Irrégularité faible 14
1.4.1.2 Irrégularité moyenne 14
1.4.1.3 Irrégularité forte 16

2 Granularité et problèmes hautement irréguliers **19**
2.1 Granularité et parallélisation des problèmes hautement irréguliers 19
2.1.1 Présentation des problèmes hautement irréguliers 20
2.1.2 La granularité des activités 20
2.1.3 Granularité du support d'exécution 23
2.1.4 Dualité entre les granularités applicative et exécutive 23
2.2 Étude de parallélisation à gros grain 24
2.2.1 Impact de la granularité applicative 24
2.2.2 Transition du grain fin vers le gros grain 26
2.2.3 Utilisation d'une structure de données globale 27
2.2.4 Schéma centralisé 28
2.2.4.1 Principe 28
2.2.4.2 Avantages 28
2.2.4.3 Inconvénients 29
2.2.4.4 Performances 29
2.2.5 Schéma distribué 31
2.2.5.1 Principe 31
2.2.5.2 Avantages 33
2.2.5.3 Inconvénients 33
2.2.5.4 Exemple d'implantation 34
2.2.6 Approches hybrides 38
2.2.6.1 Approche hiérarchique 39
2.2.6.2 Nommage centralisé et stockage réparti 40
2.2.7 Conclusion sur l'approche à gros grain 42
2.3 Conclusion 43

3 Virtualisation et ordonnancement **45**
3.1 Environnements à granularité fine et applications hautement irrégulières 46
3.1.1 Le principe des supports à grain fin 46
3.1.2 Avantages 48
3.1.3 Conclusion 49
3.2 Processus légers et distribution 49
3.2.1 Communication directe 50
3.2.1.1 TPVM 51
3.2.1.2 CHANT 51
3.2.1.3 Discussion sur les environnements à passage de message 52
3.2.2 Invocation distante de services 53
3.2.2.1 NEXUS 55
3.2.2.2 ATHAPASCAN-0b 56
3.2.2.3 Discussion sur les environnements à invocation distante de services . 57
3.2.3 Conclusion 59
3.3 Présentation de la virtualisation 60
3.3.1 Objectif 60
3.3.1.1 La décomposition en activités 60
3.3.1.2 La gestion des activités 63
3.4 Utilisation de la mobilité dans les systèmes distribués 65
3.4.1 ARIADNE 65
3.4.2 MILLIPEDE 66
3.4.3 EMERALD 68
3.4.4 PM^2 69
3.4.5 Conclusion 70
3.5 Ordonnancement distribué d'activités à grain fin 71
3.5.1 Virtualiser $\Longrightarrow$ Ordonnancer 71
3.5.2 Ordonnancement dynamique pour activités de granularité fine 72
3.5.2.1 MILLIPEDE 73
3.5.2.2 CILK 75
3.5.2.3 ATHAPASCAN-1 78
3.5.3 Discussion sur les ordonnanceurs à grain fin 81
3.6 Conclusion 83

4 Generic Threads Load Balancer **85**
4.1 Ordonnancement des applications hautement irrégulières 86
4.1.1 Particularités des applications hautement irrégulières 86
4.1.1.1 Absence de dépendances exploitables pour l'ordonnancement 86
4.1.1.2 Influence de l'ordonnancement sur la quantité de travail 87
4.1.1.3 Degré de parallélisme dépendant de l'application 89
4.1.2 Conséquences sur les caractéristiques d'un ordonnanceur générique 90
4.1.3 Conclusion 92
4.2 Ordonnancer = Placer + Migrer 93
4.2.1 Migration et ordonnancement 93
4.2.2 Ordonnancement avec migration = Équilibrage dynamique de charge 94
4.3 L'approche GTLB 94
4.3.1 Objectifs 95

4.3.2 Caractéristiques 95
4.4 Fonctionnement 97
4.4.1 Bornage du coût relatif de l'ordonnancement 101
4.5 Implantation 102
4.5.1 Caractéristiques d'une politique 102
4.5.2 Définition de la politique d'ordonnancement 103
4.5.3 Interface 104
4.5.3.1 Choix d'implantation 104
4.5.3.2 Déclaration du critère et des opérations associées 104
4.5.4 Détails sur l'implantation 106
4.5.4.1 Fonctionnement des activités de régulation 106
4.5.4.2 Fonctionnalités particulières 107
4.5.4.3 Paramètres de fonctionnement 109
4.5.5 Intégration dans une application 110
4.6 Exemple d'utilisation 110
4.6.1 Critère utilisé 110
4.6.2 Implantation de cette politique 110
4.6.3 Résultats expérimentaux sur une application témoin 114
4.7 Conclusion 115

5 Priorités et applications irrégulières distribuées 117
5.1 Présentation des priorités 117
5.1.1 Dans les systèmes d'exploitation 117
5.1.2 Dans les applications irrégulières 118
5.2 Priorités et parallélisme à grain fin 119
5.2.1 Priorités dans SOLARIS 119
5.2.2 Priorités dans MARCEL 119
5.2.3 Utilisation des priorités exécutives dans les applications irrégulières 121
5.2.4 Comment affecter une priorité à une tâche? 122
5.2.5 Application sur le problème du voyageur de commerce 124
5.2.5.1 Description de l'implantation 125
5.2.5.2 Résultats expérimentaux 125
5.3 Priorités et distribution 126
5.3.1 Présentation du problème 126
5.3.1.1 Définition 127
5.3.1.2 Le critère de qualité (ϵ) 128
5.3.2 Le régulateur LBMP 129
5.3.2.1 Spécialisation de GTLB 129
5.3.2.2 Résultats expérimentaux 134
5.4 Conclusion 135

6 Conclusion et perspectives 137
6.1 Travaux réalisés 138
6.1.1 Ordonnanceur générique 138
6.1.2 Utilisation des priorités 138
6.2 Perspectives 138
6.2.1 Évolution du régulateur 139

6.2.2 Validation sur de nouvelles applications 139
6.2.3 Multi-programmation légère et priorités 140

A Description des applications étudiées **141**
A.1 Description de l'algorithme Branch&Bound 141
A.2 L'affectation quadratique 141
A.3 Le voyageur de commerce (Traveling Salesperson Problem) 142

Table des figures

2.1 Graphe de relation entre les tâches d'une application hautement irrégulière 20
2.2 Agglomérat d'activités élémentaires en activités applicatives 22
2.3 Courbe typique de l'influence de la granularité sur les performances d'une application hautement irrégulière d'exploration arborescente 25
2.4 Influence de la granularité sur le temps d'exécution de la résolution d'un problème d'affectation quadratique de taille 15 26
2.5 Schéma général de parallélisation d'applications hautement irrégulières dans un environnement à gros grain 27
2.6 Schéma centralisé d'implantation d'une structure de données parallèle globale 28
2.7 Mesures d'accélération pour la résolution du célèbre problème d'affectation quadratique Nugent de taille 16 30
2.8 Schéma distribué d'implantation d'une structure de données parallèle globale 32
2.9 Schéma de communication complet pour l'implantation distribuée d'une structure de données globale sur une architecture parallèle sans mémoire partagée 34
2.10 Les états 35
2.11 Répartition des nœuds pour la résolution d'un problème d'affectation quadratique dans un contexte à grosse granularité de parallélisme avec un schéma de conception entièrement distribué 37
2.12 Schéma hiérarchisé d'implantation d'une structure de données parallèle globale . . . 39
2.13 Schéma d'implantation d'une structure de données parallèle globale utilisant le principe d'indirection 41

3.1 Processus légers encapsulés dans un processus lourd 46
3.2 Schéma d'implantation d'une application hautement irrégulière sur un support d'exécution proposant des activités de granularité fine 47
3.3 Intégration des processus légers et de la distribution par l'invocation de services distants 54
3.4 Illustration de l'utilisation du mécanisme des ports et des étiquettes dans l'environnement ATHAPASCAN-0b 57
3.5 Exemple d'implantation d'une application hautement irrégulière avec un environnement proposant l'appel de procédure distant comme paradigme de distribution . . . 58
3.6 Le principe de virtualisation 61
3.7 Les trois modes possibles d'invocation de services à distance. 62
3.8 Le concept de clonage léger. 62
3.9 Le concept de migration. 65
3.10 Un système d'ordonnancement 71
3.11 Le programme parallèle CILK calculant le n^{ime} élément de la suite de Fibonacci . . . 76

3.12 Graphe de dépendances de l'application CILK calculant Fibo(4) 77
3.13 Graphe de dépendances sur les variables utilisé pour ordonnancer les activités 79
3.14 Positionnement des environnements étudiés sur les trois axes {Réactivité, Granularité, Paramétrisation} 82

4.1 Exemple d'ordonnancement avec respect de l'importance des activités 88
4.2 Multiprogrammation légère et multiprogrammation lourde 96
4.3 Structure d'une application virtualisée utilisant notre ordonnanceur 97
4.4 Inondation d'un processeur par l'afflux de processus légers migrés pour compenser un déséquilibre 99
4.5 Fonctionnement complet de l'ordonnanceur générique GTLB 100
4.6 Modélisation du coût maximal de l'ordonnanceur 102
4.7 Accélération obtenue avec l'ordonnanceur pour une application synthétique de cent activités 115
4.8 Accélération obtenue avec l'ordonnanceur pour une application synthétique de deux cents activités 116

5.1 Affectation du processeur à deux processus légers, un de priorité un et un de priorité dix 120
5.2 Exemple d'exploration spéculative d'un arbre de recherche 121
5.3 Exemple de situation où les priorités locales ne sont plus respectées dans un contexte global 127
5.4 Exemple de situation où il n'est pas possible de satisfaire la propriété P (la valeur de P_3 est différente de la figure 5.3) 128
5.5 Correction d'une situation déséquilibrée afin de satisfaire les priorités 130
5.6 Accélération moyenne obtenue sur 10 problèmes du TSP de taille 16 pour deux valeurs de ϵ 135

Liste des tableaux

2.1 Correspondance entre granularité exécutive et applicative 23
2.2 Performances de la résolution des problèmes de taille 16 d'Eschermann et Wunderlich ainsi que d'autres problèmes classiques de taille 20 avec une approche centralisée . . 31
2.3 Action entreprise à la réception d'un message de changement d'état 36
2.4 Performances de la résolution des problèmes de taille 16 d'Eschermann et Wunderlich et d'autres problèmes de taille 20 avec une approche centralisée et une approche distribuée 38
2.5 Classification des différents schémas de parallélisation d'une structure de données globale distribuée 42

3.1 Expression du produit de deux matrices, à gauche avec l'invocation distante de service, à droite avec le clonage léger. 63

4.1 Ensemble des opérations à définir sur la charge 105
4.2 Fonctions nécessaire à la définition d'une nouvelle politique d'ordonnancement . . . 106
4.3 Liste des paramètres régissant le comportement de l'ordonnancement 109

5.1 Pourcentage du nombre de nœuds développés en fonction de l'heuristique et du nombre de processus légers pour cent problèmes générés aléatoirement du TSP de taille 12 126
5.2 Pourcentage du nombre de nœuds développés en fonction de l'heuristique et du nombre de processus légers pour cent problèmes générés aléatoirement du TSP de taille 13 126

Chapitre 1

Introduction

Un constat souvent fait dans la communauté parallèle est le décalage entre les évolutions technologiques des architectures parallèles et celles des modèles de conception d'applications parallèles. Ce constat se justifie plus particulièrement dans le cas des applications dites irrégulières pour lesquelles peu d'approches génériques de conception sont proposées. L'ambition du projet ESPACE est de proposer une telle approche.

1.1 ESPACE

Mes travaux de recherche et de thèse ont été réalisés au sein du projet ESPACE du LIFL. L'objectif de ce projet est de définir un cadre de conception et le support exécutif pour des applications irrégulières. Le cœur du projet s'est dans un premier temps construit autour de l'environnement PM^2 développé par Raymond Namyst[1] durant sa thèse au LIFL. Les travaux récents autour de PM^2 se concentrent sur l'exploitation efficace des résaux haut-débit.

Une de nos volontés au lancement de ce projet a été de promouvoir l'utilisation de PM^2 à l'extérieur de notre laboratoire. C'est ainsi que j'ai, dès le début de ma thèse, participé à l'opération inter-PRC STRATAGÈME et je me suis plus particulièrement intéressé aux applications d'optimisation combinatoire. J'ai eu, dans ce cadre, des contacts réguliers avec l'équipe de Catherine Roucairol à Versailles.

1.2 PM^2

Les applications fortement irrégulières, comme celles d'optimisation combinatoire, ont comme caractéristique majeure d'être potentiellement génératrices d'un parallélisme massif en terme de tâches et d'être extrêmement irrégulières et imprévisibles au niveau de la durée d'exécution des tâches générées. Cela exige donc du support exécutif des capacités évoluées de régulation et des aptitudes à supporter efficacement un nombre de tâches très important. Ces contraintes ont guidé les choix de conception de PM^2. La plateforme PM^2 s'est ainsi construite autour d'un noyau de multiprogrammation très efficace (MARCEL) capable de supporter un grand nombre d'activités. Un mécanisme original de migration est également proposé pour offrir des possibilités de régulation très évoluées.

Le point original de ce travail concerne principalement le concept de virtualisation qu'offre un tel environnement et nous montrerons que la régulation de charge dans un tel contexte permet d'offrir

1. Aujourd'hui maître de conférences à l'ENS-Lyon

une véritable virtualisation de l'architecture. Nous avons appliqué ces concepts aux applications hautement irrégulières afin de déterminer leur viabilité, au niveau structuration comme au niveau exécutif. Afin de détailler les points principaux de notre contribution il est nécessaire de présenter les spécificités des applications visées en les comparant aux autres classes d'applications irrégulières, ce que nous ferons dans une prochaine section de cette introduction.

1.3 Contributions de la thèse et organisation du document

Dans cette thèse nous allons étudier la manière de paralléliser les algorithmes appartenant à la famille des Branch&Bound dans un contexte parallèle faiblement couplé. Notre contribution se compose de différents aspects :

Granularité des applications hautement irrégulières Nous analyserons l'impact du choix de la granularité des applications irrégulières dans les performances. Par cette étude, nous mettrons en évidence la dualité existant entre la fine granularité applicative et la granularité exécutive imposée par les supports d'exécution traditionnels à gros grain. Nous proposerons alors un environnement basé sur les processus légers qui offre une granularité exécutive plus fine et qui est donc en adéquation avec la granularité « naturelle » des applications visées. Cette étude fait l'objet du chapitre 2 et du début du chapitre 3.

Virtualisation et ordonnancement L'utilisation des processus légers dans un cadre distribué est souvent difficile à mettre en œuvre. Il nécessite la définition d'un modèle de programmation qui prend en compte la fine granularité des activités. Notre proposition est basée sur la virtualisation de l'architecture. Elle est composée de deux aspects : des concepts structurants pour découper une application en activités parallèles et une prise en compte efficace de ces activités à l'exécution. Mon travail concerne plus particulièrement le second aspect. J'ai étudié les besoins exécutifs des applications hautement irrégulières, en particulier au niveau ordonnancement. De cette étude, il apparait que les critères à prendre en compte pour une bonne exécution sont étroitement liés à l'application concernée. J'ai donc proposé un ordonnanceur générique qui peut être instancié pour chaque application, ou classe d'applications. L'étude de la virtualisation et de l'ordonnanceur générique fait l'objet des chapitres 3 et 4.

Rôle des priorités Les applications que nous étudions ne sont pas à travail constant, c'est-à-dire que l'ordonnancement de ses activités influent sur la quantité de travail à fournir pour résoudre le problème. Cette particularité est prise en compte dans les environnements à gros grain en ordonnant l'exécution des activités suivant leur importance. Cela provoque une exécution *spéculative* car cet ordre est calculé par des méthodes heuristiques qui ne sont donc pas sûres à 100%. Dans un environnement à grain fin toutes les activités sont exécutées simultanément et cet ordre est donc sans objet. Pour conserver cet aspect spéculatif à l'exécution, nous proposons d'utiliser la notion de priorité associée aux processus légers qui permet de privilégier certaines activités sur d'autres. Le problème est que cette notion n'est définie que dans un cadre local, le respect des priorités dans un cadre distribué nécessite la mise en place d'un mécanisme régulation de charge. Cette étude sur l'utilisation des priorités dans les applications irrégulières et sur leur respect dans un cadre distribué fait l'objet du chapitre 5.

Ce document traitant des problèmes hautement irréguliers nous allons présenter cette classe de problèmes et des propriétés de ses applications.

1.4 Caractérisation de l'irrégularité

L'ensemble des problèmes irréguliers est un domaine vaste dont les frontières semblent parfois bien floues. Il est pourtant essentiel de bien comprendre leurs particularités pour les appréhender de manière satisfaisante et donc proposer des solutions adaptées et performantes pour leur résolution.

Le cœur de cette thèse portant sur la parallélisation de problèmes hautement irréguliers sur des architectures distribuées, nous allons définir dans cette section ce que nous considérons comme des problèmes irréguliers. Nous verrons que ces problèmes peuvent être divisés en trois classes différentes suivant leur *degré d'irrégularité*. Nous préciserons les spécificités de chaque classe en illustrant chacune d'elle par un exemple concret.

1.4.1 Caractérisation de l'irrégularité

Essayons de définir ce qui fait l'irrégularité d'une application parallèle de manière naïve. Intuitivement, on peut dire d'une application qu'elle est irrégulière si son comportement est imprévisible, c'est-à-dire s'il n'est pas possible de le connaître par des méthodes uniquement statiques sans exécuter l'application. Précisons cette définition intuitive en détaillant en particulier la notion de comportement pour une application parallèle.

Le comportement d'une application parallèle correspond à ses caractéristiques à l'exécution en terme par exemple de nombres d'activités, coopérantes ou indépendantes, et de leurs besoins, en puissance de calcul ou en espace mémoire. En partant de cette définition, on peut dire que plus il est difficile d'extraire statiquement, sans exécuter l'application, ces informations plus elle est irrégulière. Au contraire, une application dont le comportement peut être entièrement extrait de son expression algorithmique, donc sans nécessiter d'exécution sur des données précises, sera d'une irrégularité très faible.

En essayant de formaliser le raisonnement précédent, il apparaît que la notion au centre de la définition d'irrégularité est le degré de dynamicité du comportement de l'application. Plus il est élevé, plus le degré d'irrégularité de l'application est important. La prise en compte de ce comportement est importante, on peut même dire centrale, pour déterminer une exécution efficace des activités composant l'application. On peut donc lier le degré d'irrégularité de l'application avec la complexité de prévoir une gestion efficace de ses activités, c'est-à-dire déterminer leur placement et leur instant d'exécution dans l'exécution globale de l'application. On parle alors de l'*ordonnancement* des activités de l'application.

Ce raisonnement intuitif nous amène à la définition que Jean-Louis ROCH et son équipe ont proposée, dans le cadre des travaux du projet STRATAGÈME[Rou95], pour une classification des problèmes irréguliers[GRV95, RVR95]. Le critère qu'ils utilisent pour classer les applications irrégulières est celui de la difficulté d'ordonnancement de ses tâches. Plus précisément : « **L'irrégularité d'un algorithme est directement liée à la complexité d'ordonnancement des tâches qui le composent** ». Cette complexité est définie à partir des opérations faites par l'ordonnanceur durant l'exécution, opération qui, quand c'est possible mais ce n'est pas toujours le cas, construit le graphe de dépendances de l'application. Ce graphe décrit les dépendances existantes entre les tâches qui composent l'application. Ces dépendances peuvent être, par exemple, temporelles ou sur les données.

Les auteurs proposent d'utiliser le degré de dynamicité du graphe de dépendance afin de classer les applications irrégulières en trois catégories :

Faiblement irrégulière contient les applications irrégulières dont le graphe de précédence est prévisible,

Moyennement irrégulière pour les applications dont le graphe est semi-prévisible,

Hautement irrégulière pour lesquelles le graphe est totalement imprévisible.

Nous allons détailler chacune de ces catégories pour en identifier les particularités. Nous illustrerons chacune à l'aide d'un exemple concret d'applications mettant en évidence le comportement irrégulier d'applications de la classe.

1.4.1.1 Irrégularité faible

La première catégorie est celle des problèmes irréguliers dont le graphe de précédence est entièrement connu à partir de la taille des données en entrée, indépendamment de la valeur de ces dernières. On dit que leur graphe est *prévisible* et leur degré d'irrégularité est faible car il est possible de calculer statiquement un ordonnancement sans devoir utiliser de mécanismes supplémentaires d'ordonnancement durant l'exécution.

Dans le cas particulier où le graphe dépend de la valeur de certaines données, une évaluation partielle portant sur ces données peut permettre d'extraire le graphe à partir duquel l'ordonnancement optimal pourra être calculé. Cette méthode ne peut évidemment s'appliquer que dans le cas où le coût de cette évaluation partielle n'est pas prohibitif par rapport au coût total de l'application. L'idée centrale est ici de construire le graphe avant l'exécution pour diminuer le degré d'irrégularité de l'application afin de pouvoir l'ordonnancer avec des méthodes statiques ce qui constitue un domaine largement exploré [ERLA94] et où on sait obtenir de bonnes performances.

Exemple Un exemple concret d'application de cette classe est la factorisation de Cholesky pour les grandes matrices creuses telle que proposée par Jean Roman et Laurent Facq [Fac95, FR95]. Dans cette méthode de résolution, les éléments non nuls sont, dans une première phase, regroupés en blocs puis un traitement symbolique sur les blocs ainsi créés permet d'obtenir le graphe de précédence de l'algorithme de Cholesky sur la matrice obtenue. Cette deuxième étape ne nécessite que de connaître la taille des blocs créés et non de résoudre complètement, ou même partiellement, le système. À partir de ce graphe il est possible de calculer, statiquement, un ordonnancement optimal pour cet algorithme parallèle irrégulier.

Pour résumer, les problèmes de faible irrégularité sont ceux dont le graphe de précédence peut être construit sans devoir exécuter l'application. Leur ordonnancement peut être calculé statiquement en n'utilisant éventuellement que la taille des données en entrée indépendamment de ces dernières.

1.4.1.2 Irrégularité moyenne

Les problèmes ayant un degré d'irrégularité faible sont ceux dont il est possible de calculer le graphe de dépendance sans utiliser les valeurs des données en entrée. Les problèmes d'irrégularité moyenne ont eux un graphe *semi-prévisible*. Ces applications se présentent comme une succession de phases, chaque phase étant composée d'un algorithme parallèle.

Pour ordonnancer les calculs d'une phase deux techniques peuvent être utilisées, selon les caractéristiques de l'application :

- À la fin de la phase n il est possible de déterminer le volume des calculs de la phase $n+1$ et donc de calculer un ordonnancement pour les tâches composant cette phase avant le début de leur exécution. Cela peut être fait par une redistribution des données [FMP95] ou, de manière duale, des activités.

- Si la technique précédente n'est pas utilisable à cause du manque d'informations sur les contraintes d'ordonnancement, seule une partie des tâches est ordonnancée, le reste étant conservé « en réserve » pour équilibrer la charge ultérieurement en utilisant par exemple une stratégie d'allocation à la demande [Hem94].

L'ordonnancement des applications de cette classe est réalisé en utilisant le graphe de précédence qui est construit durant le déroulement de l'application car il ne peut pas être directement extrait de la taille des données contrairement aux problèmes vus précédemment. Il est possible de calculer un ordonnancement optimal pour ces problèmes mais cela doit être fait durant l'exécution, entre chaque phase, et non statiquement avant l'exécution.

Pour les problèmes de cette classe, le degré d'irrégularité d'une application dépend de deux paramètres qui sont :

La complexité d'ordonnancement de chaque phase Le nombre d'activités composant chaque phase est un facteur de la complexité d'ordonnancement. Généralement, plus il est élevé et plus l'ordonnancement sera coûteux. Par contre, s'il est trop faible, le degré de parallélisme intra-phase deviendra trop petit pour exploiter l'architecture de la machine parallèle au maximum de son potentiel.

Le nombre de phases Pour une application donnée, une augmentation du nombre de phases entraîne une diminution du nombre de tâches à ordonnancer durant chaque phase et donc de l'irrégularité. Mais cela va à l'encontre du but recherché en parallélisant car cela revient à « séquentialiser» l'algorithme en faisant ainsi décroître le degré de parallélisme de l'application.

Exemple De nombreux problèmes réels appartiennent à cette classe d'irrégularité. On peut citer dans STRATAGÈME des applications de synthèse d'images [MR], de calcul formel [AFR+94] ou encore la machine dictionnaire [Dub96]. À l'extérieur de STRATAGÈME citons les travaux de Cyrille Fonlupt [Fon95] sur la régulation de charge en contexte data-parallèle ou, en anglais, *data-parallel.*

Nous allons détailler plus précisément la méthode ParList[FMP95] dont un des domaines d'application est la synthèse d'image.

L'idée au centre du projet ParList est d'équilibrer la charge entre les processeurs par le biais de la redistribution des données de cette application. Le but de la structure ParList est de faciliter à la fois la lisibilité et la portabilité des applications qui l'utilisent en rendant transparentes les opérations globales liées à la structure de données. Il consiste en un calcul de la charge idéale de chaque processeur en terme de quantité de données puis effectue une redistribution conformément aux résultats de ce calcul. Chaque redistribution marque le début d'une nouvelle phase dans la terminologie définie auparavant. La particularité de cet algorithme tient en ce que les communications nécessaires pour le placement des données ne concernent que les voisins immédiats de chaque processeur, conservant ainsi une notion de localité importante dans les architectures parallèles traditionnelles.

Les applications visées initialement par cet algorithme concernent la synthèse d'images et plus particulièrement l'imagerie médicale en trois dimensions. Par exemple, l'application de squelettisation 3D nécessite pour le calcul d'un point la connaissance de ses 26 voisins. La notion de localité y est donc très importante. Cet algorithme est typique de la classe d'applications considérée car il consiste en une alternance de phases de calcul et de communications, chaque phase de calcul pouvant être ordonnancée de manière efficace en utilisant les informations de la phase précédente.

Les problèmes de cette classe d'irrégularité ont un graphe de dépendance semi-prévisible et

qui peut donc être construit au fur et à mesure de l'exécution afin d'être utilisé pour calculer l'ordonnancement.

1.4.1.3 Irrégularité forte

Les problèmes de forte irrégularité sont ceux dont le graphe de précédence est complètement imprévisible et ne peut donc pas être utilisé pour réaliser un ordonnancement des tâches de l'application parallèle. Les informations qu'il serait possible d'en extraire seraient très rapidement obsolètes et comme, de plus, il n'est pas possible de les utiliser pour prédire le comportement futur elles sont, de toute façon, inutiles. L'ordonnancement ne peut, par conséquent, être effectué que de manière purement dynamique durant l'application en agissant, *a priori* et/ou *a posteriori*, sur la localisation et la date de lancement des tâches, par exemple en en retardant l'exécution afin de les utiliser pour résoudre un déséquilibre découvert ultérieurement.

Le haut degré d'irrégularité des applications de cette classe provient de leur non-déterminisme. Si on se reporte à la définition de l'irrégularité d'une application, ordonnancer les activités d'une telle application est un problème complexe du fait de la faible quantité d'informations fiables utilisables.

Exemple L'exemple probablement le plus connu de cette classe d'applications est celui du parcours d'arbre de recherche par une méthode telle que la méthode Branch&Bound. Cette méthode est utilisée pour résoudre de façon exacte des problèmes d'optimisation combinatoire, contrairement aux méthodes heuristiques comme la méthode tabou ou les algorithmes génétiques qui obtiennent des solutions approchées dont elles ne prouvent pas l'optimalité. Son principe est d'explorer un arbre de recherche décrivant l'ensemble des solutions possibles du problème. Les nœuds de l'arbre contiennent des solutions partielles, les feuilles contiennent des solutions complètes. Par une heuristique, à chaque nœud de l'arbre est associée une valeur qui dépend de la solution partielle qu'il contient. Cette heuristique a la particularité d'être strictement montone [2] suivant la profondeur dans l'arbre, c'est-à-dire que tous les nœuds contenus dans un sous-arbre auront une valeur supérieure à celle du nœud racine du sous-arbre.

En utilisant cette valeur et cette propriété, il est possible de couper, c'est-à-dire de ne pas explorer, des branches de l'arbre durant l'exploration et donc de réduire la taille totale de l'espace exploré. Des détails sur les différentes stratégies de parcours de l'arbre se trouvent en annexe A.1 page 141. Un des critères principaux pour juger un algorithme de Branch&Bound est sa capacité à trouver tôt dans l'exécution de bonnes solutions, c'est-à-dire proches de l'optimal, permettant ainsi de couper beaucoup de branches. La quantité de nœuds élagables ainsi que leur localisation dans l'arbre ne peuvent être calculées avant leur exploration, ce qui est la cause principale du haut degré d'irrégularité de ce type d'applications. En effet, il est impossible d'évaluer au début de l'exploration d'un sous-arbre quel pourcentage de ce sous-arbre il faudra réellement explorer et donc la charge de calcul qu'il nécessitera.

Un autre type d'applications représentatif de cette classe, mais non issu de l'optimisation combinatoire, est celui de la simulation d'un ensemble de particules.

La simulation du déplacement et des interactions d'un ensemble de particules dans un espace fini est une des applications principales développées par les physiciens utilisateurs de machines parallèles. C'est par exemple l'une des applications étudiées actuellement par l'équipe APACHE de Grenoble [BTC96, BT96]. Elle consiste en la modélisation du mouvement de particules élémentaires, qui peuvent être des atomes, des molécules ou encore des électrons, dans un espace clos en prenant en compte les phénomènes d'attraction/répulsion, de collision, de fusion ... L'évolution des données,

2. Croissante dans le cas d'une minimalisation, décroissante dans le cas d'une maximalisation.

c'est-à-dire des particules, est complètement imprévisible et peut varier considérablement pour des données initiales pourtant très proches ce qui rend impossible toute tentative d'analyse *a priori* du comportement de l'application. L'exploitation d'un graphe de calculs serait difficile, ne serait-ce que par l'incertitude portant sur l'existence des particules à la prochaine étape. Dans ce cas la seule solution viable est un ordonnancement à la volée des tâches, représentant une (ou un ensemble de) particule, sur les processeurs afin de garantir une exécution efficace. Pour cela on utilise généralement un algorithme de placement pour répartir initialement les tâches puis on modifie ce placement pendant l'exécution pour prendre en compte les variations de charge causées par l'irrégularité de l'algorithme, ici par les créations et les destructions de particules.

Dans ce document nous allons principalement nous intéresser à cette classe d'applications et en particulier à leur résolution dans un contexte parallèle entièrement distribué.

Chapitre 2

Influence de la granularité dans la résolution parallèle des problèmes hautement irréguliers

Dans ce chapitre, nous allons aborder le problème de la granularité des problèmes hautement irréguliers et de son influence sur leurs méthodes de résolution parallèle. Tout d'abord, nous discuterons de la « granularité naturelle » des applications visées. Nous insisterons sur son influence lors de leur parallélisation en illustrant notre discours par des exemples. Nous établirons ensuite un parallèle entre les différentes classes de granularité proposées par les supports d'exécution distribués. Cela nous amènera à montrer la dualité existante entre les caractéristiques applicatives et les caractéristiques exécutives de la granularité. Cette dualité constitue le cœur de la problématique étudiée.

Nous proposerons ensuite des solutions permettant de faire correspondre les deux granularités, applicatives et exécutives. Nous verrons que ces solutions peuvent être classées en deux familles principales, centralisée et distribuée, chacune ayant ses particularités. Ces dernières influent sur le degré d'adaptation aux problèmes et aux architectures visés. Nous illustrerons notre discours sur les caractéristiques de chacune de ces familles par un exemple concret et des résultats expérimentaux. Enfin, nous présenterons deux méthodes qui intègrent les approches présentées auparavant afin d'en cumuler les avantages.

2.1 Granularité et parallélisation des problèmes hautement irréguliers

Comme nous l'avons vu en introduction, la parallélisation des problèmes hautement irréguliers pose des problèmes différents de ceux des autres problèmes, principalement à cause de l'impossibilité d'exploiter un graphe de précédence. **L'approche généralement utilisée consiste en une réaction *a posteriori* au comportement de l'application plutôt qu'à anticiper celui-ci.**

Dans cette section nous allons commencer par préciser ce qui fait la spécificité des problèmes hautement irréguliers, nous verrons en particulier que le découpage en tâches d'une telle application pose le délicat et crucial problème de la granularité. L'essentiel de cette section sera consacré à ce problème. Nous étudierons la granularité « intrinsèque » globalement associée à ces problèmes en la comparant à celle de problèmes appartenant à d'autres classes de plus faible irrégularité. Nous mettrons ensuite en parallèle ces constatations sur la *granularité applicative*, celle des applications

étudiées, avec la granularité proposée par les supports d'exécution communément utilisés par la communauté que nous appelons *granularité exécutive.*

2.1.1 Présentation des problèmes hautement irréguliers

La classe des problèmes hautement irréguliers d'optimisation combinatoire étant celle étudiée principalement dans ce document, nous allons revenir un peu plus en détail sur ses particularités par rapport aux autres classes d'applications irrégulières présentées en introduction.

La caractéristique principale qui démarque la classe d'applications que nous étudions, celle des problèmes d'optimisation combinatoire, des autres réside dans le fait qu'il n'y a que très peu de dépendances entre les données, voire souvent aucune. Cela implique que les activités qui effectuent des calculs sur ces données ont peu de dépendances entre elles. Une telle application se présente comme un ensemble de tâches indépendantes pouvant être ordonnancées dans un ordre quelconque, la seule relation pouvant exister entre deux tâches étant celle de parenté, c'est-à-dire qu'une tâche en crée une (ou des) autre(s) comme représenté sur la figure 2.1.

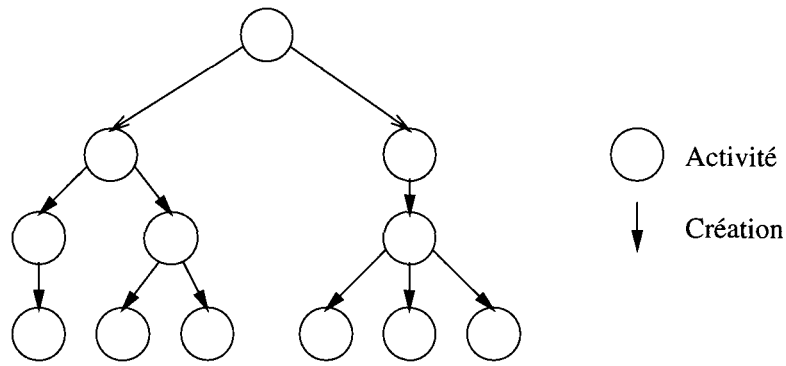


FIG. 2.1 - *Graphe de relation entre les tâches d'une application hautement irrégulière*

La conséquence directe de cette caractéristique est que le graphe de dépendance est en fait un *arbre*, peu exploitable pour l'ordonnancement. Cela est encore plus vrai si les nœuds de cet arbre ne peuvent pas être valués, ce qui est le cas dans la plupart des applications. En effet, l'irrégularité de l'application empêche de prédire la forme de l'arbre et la quantité de travail que représente chaque nœud de l'arbre.

2.1.2 La granularité des activités

Nous allons étudier ici la notion de granularité appliquée aux activités d'une application parallèle. Nous commencerons par définir ce que nous entendons par granularité puis nous illustrerons cette définition sur deux exemples.

Définition La granularité d'une application parallèle décomposée en activités peut être définie comme la quantité de calcul nécessaire à une activité indépendamment des autres.

Stricto sensu, cela signifie que la granularité minimale qu'il est possible d'obtenir, quelle que soit l'application, est l'instruction élémentaire, puisque chaque instruction n'a comme dépendance que la précédente, c'est-à-dire qu'elle doit être exécutée juste après une autre. Une granularité aussi fine n'est pas envisageable ici à cause des performances déplorables qu'elle engendrerait. Nous

allons étudier sur deux exemples, aux extrêmes du spectre de l'irrégularité, la granularité typique utilisable efficacement.

• Prenons tout d'abord le cas d'un produit de matrices carrées $C = A \times B$ avec A et B de dimension n. Pour calculer la valeur d'un élément de C il faut disposer d'une ligne et d'une colonne de A et de B. Intuitivement, la granularité de cette application est d'avoir une activité par élément résultat. Le nombre d'activités sera donc de n^2 et la complexité, donc la granularité, de chaque activité en $O(n)$. La granularité la plus fine[1] consisterait en la création d'une activité pour effectuer la multiplication de deux éléments des matrices A et B mais le coût de gestion de l'activité serait alors élevé par rapport à son coût effectif, et serait donc peu performante[2]. Nous revenons sur ce point dans la suite.

• L'autre exemple, dont les détails algorithmiques sont décrits dans l'annexe A.1, concerne les algorithmes de résolution exacte de problèmes combinatoires par exploration arborescente d'espaces de recherche. Pour étudier leur granularité il convient de détailler la manière dont la parallélisation est effectuée. Trois approches de parallélisation sont possibles pour ce type d'applications :

Parallélisation au niveau du nœud L'évaluation d'un nœud se fait en parallèle donc par plusieurs activités. Cette approche n'est viable que dans le cas où cette opération est très coûteuse en regard du coût de gestion d'une activité, ce qui est rarement le cas.

Parallélisation du parcours Chaque activité explore une partie de l'espace de recherche. C'est l'approche la plus adaptée aux architecture parallèles que nous visons.

Exploration concurrente utilisant différentes méthodes Le parallélisme est utilisé pour permettre l'exploration simultanée de l'espace par différents algorithmes de même type, par exemple une exploration largeur d'abord et une profondeur d'abord, ou de types différents, une méthode Branch&Bound et une exploitant une, ou plusieurs, métaheuristiques. Si ces deux méthodes coopèrent durant l'exécution on parle alors de méthode *hybride*. Des travaux récents s'intéressent à cette technique [BPT96].

Il est à noter que cls trois approches sont classées ci-dessus suivant un ordre croissant de granularité, la première a une granularité inférieure au nœud, la seconde de l'ordre du nœud (ou d'un ensemble de nœuds) et la troisième de la résolution complète du problème.

Dans la première approche, la granularité est très fine : son utilisation n'est envisageable que pour des problèmes très particuliers et les méthodes mises en œuvre dans ce contexte sont totalement spécifiques à l'application étudiée et donc non généralisables.

La troisième méthode est similaire à la seconde dans le cas où les méthodes ne coopèrent pas. Au contraire, une coopération crée de nouvelles dépendances entre les activités. Ces dépendances peuvent être soit directes, création dépendante de résultats non encore obtenus, soit indirectes, comme des accès concurrents à des structures de données partagées, par exemple. Leur étude dépasse le cadre de ce document et nous n'en parlerons donc pas plus.

La seconde approche, celle où le parallélisme intervient dans le parcours de l'espace de recherche, correspond au type d'applications que nous visons. Dans ce cas la granularité élémentaire de calcul, c'est-à-dire la quantité d'opérations élémentaires composant une activité indépendante, peut être très fine, allant jusqu'à quelques dizaines d'instructions machine, car elle représente l'évaluation

1. Si l'on reste au niveau vectoriel, on pourrait aussi envisager de paralléliser le produit scalaire effectué pour calculer la valeur d'un élément.

2. Elle peut cependant être intéressante si les éléments des matrices sont très gros, comme dans le cas particulier de matrices de matrices.

d'un nœud et le choix d'explorer ou non le sous-arbre correspondant. Cependant, ce grain est modifiable à volonté en agglomérant plusieurs activités en une seule comme illustré sur la figure 2.2 qui complète le graphe présenté sur la figure 2.1. **La possibilité de manipuler à volonté la granularité constitue une des particularités de ce type d'applications.**

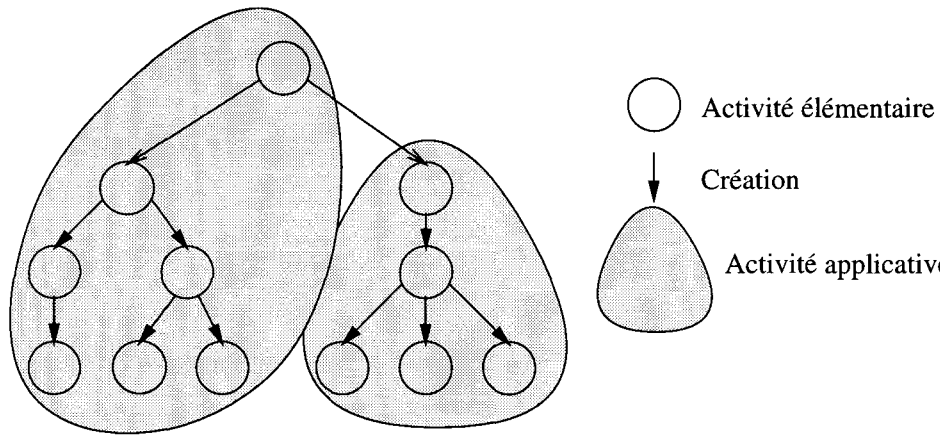


FIG. 2.2 - *Agglomérat d'activités élémentaires en activités applicatives*

En conséqunce se pose le problème délicat du choix de la granularité. Si elle est prise trop fine, le coût de gestion de l'activité peut être trop important par rapport au coût de l'activité elle-même. Par exemple, imaginons qu'une activité représente 100 instructions machine. Quel que soit le système de gestion des activités utilisé, le coût de ce dernier sera plus élevé, l'application passera donc plus de temps à gérer les activités qu'à les exécuter. Par contre, si la granularité est trop importante, la capacité de parallélisme de l'architecture parallèle utilisée peut être sous-exploitée par manque d'activités. De plus, si le nombre d'activités est peu important il devient plus difficile d'équilibrer la charge[3] car on dispose de moins d'éléments à placer ou à déplacer pour réaliser l'équilibrage. Cela est d'autant plus difficile que, comme on parle ici d'applications irrégulières, plus on manipule des tâches comportant un grand nombre d'activités élémentaires plus il est difficile d'estimer la charge de calcul total qu'elles représentent. En effet, plus les entités manipulées sont de grain fin, plus la marge d'erreur dans l'estimation de leur coût est faible.

Idéalement, c'est-à-dire pour une exécution optimale, il faudrait déterminer la granularité des tâches applicatives dynamiquement en réagissant « à la volée » au comportement de l'application. Malheureusement, c'est rarement le cas car cette valeur est souvent déterminée[4] par la granularité du support d'exécution utilisé pour paralléliser l'application comme nous allons le voir.

Il est important de distinguer la notion de granularité applicative, dont nous avons parlé jusqu'à présent, de la granularité des activités supportées par les supports d'exécution, que nous appelons granularité exécutive. La première dépend des caractéristiques de l'application, tel que son degré d'irrégularité, alors que la seconde dépend du contexte dans lequel l'exécution a lieu. Les deux devraient intuitivement être indépendantes mais nous allons voir que ce n'est pas le cas. Tout d'abord, nous allons détailler les deux familles de supports d'exécution en les classant suivant la granularité des activités qu'ils supportent. Nous verrons ensuite comment faire correspondre des granularités applicatives et exécutives différentes.

3. Le terme charge doit ici être pris au sens large, il ne s'agit pas uniquement de la charge de calcul.
4. C'est-à-dire bornée, en pratique.

2.1.3 Granularité du support d'exécution

On définit la granularité d'un support d'exécution pour applications parallèles au type d'activités qu'il peut gérer. D'une manière générale, ces supports peuvent être classés en deux catégories suivant la finesse de grain de leurs activités :

Gros grain Les activités correspondent à des processus UNIX. Ces supports utilisent typiquement l'envoi de message comme paradigme de distribution. Les exemples les plus répandus de support d'exécution de ce type sont PVM [GBD+95] ou MPI [Mes94][5].

Grain fin Les activités manipulées correspondent à des flots d'exécution, ces flots étant encapsulés dans des processus de gros grain. Beaucoup de travaux ont été menés ces dernières années sur ce type de support d'exécution. Citons par exemple ATHAPASCAN-0 développé dans le cadre du projet APACHE[Pla93], NEXUS[FKT94], CHANT[HCM94], TPVM [FS95] ou encore DTMS [Col95]. Une description plus avancée de ces environnements se trouve dans la section 3.2 page 49. Celui développé dans notre équipe, PM^2 [Nam97b], sera détaillé dans la suite de ce document. Alors que les environnements à gros grain poussent à structurer l'application en tâches communicantes, ceux à grain fin font privilégier la structuration en activités indépendantes en prenant éventuellement en charge les dépendances de données, comme c'est le cas dans Athapascan-1 [DGR97] qui est un exemple d'environnement complet, à la fois méthodologique et exécutif, poussant vers une structuration en activités. Nous reviendrons plus tard sur ce point important. Le lecteur intéressé peut se rapporter à [Chr96, Nam97b, Gin97] pour une étude exhaustive de ce type d'environnement.

Pour être tout à fait complet il faut préciser l'existence d'une troisième catégorie, les processus de poids moyen. Ils sont en fait une spécialisation des processus légers qui ont la particularité d'être gérés directement par le système d'exploitation. Ils sont donc plus sûrs d'utilisation que les processus légers mais cela se paie par un coût de gestion (création, destruction, ...) plus élevé que des processus légers traditionnels. On trouve des activités de ce type dans le système SOLARIS [LB96].

2.1.4 Dualité entre les granularités applicative et exécutive

Nous avons présenté deux visions différentes de la granularité : applicative et exécutive. La question qui se pose est celle de la prise en compte des deux au moment de la parallélisation d'applications hautement irrégulières. Par exemple, peut-on concevoir et exécuter efficacement une application de gros grain sur un environnement proposant une granularité fine? Plus globalement, peut-on faire la correspondance entre différentes granularités exécutives et applicatives? Un élément de réponse se trouve dans le tableau 2.1 que nous allons détailler maintenant.

Granularité Applicative / Exécutive	Gros	Fin
Gros	Correspondance implicite	Mécanisme explicite
Fin	Correspondance implicite	Correspondance implicite

TAB. 2.1 - *Correspondance entre granularité exécutive et applicative*

5. Certaines implantations de MPI supportent un grain plus fin de parallélisme et la nouvelle norme, MPI2 [Mes], impose cette caractéristique.

Le cas de la diagonale est trivial, les correspondances posant problème sont celles entre granularités différentes. Une granularité applicative grosse peut être supportée immédiatement par un support d'exécution à grain fin car les processus lourds ne sont que des processus légers encapsulant des informations supplémentaires[6].

Une grosse granularité applicative correspond à des problèmes dans lesquelles les dépendances sur les données, et donc entre les activités, concernent de grosse quantités de données. Un exemple est un produit de matrices, chaque matrice étant de dimension 10000. Il est envisageable dans ce cas de créer un processus lourd uniquement pour effectuer ce calcul, la granularité applicative est élevée et l'application peut être implantée de manière efficace avec un environnement supportant une grosse granularité exécutive. Comme nous l'avons expliqué dans le paragraphe précédent, cette classe d'applications peut également être implantée efficacement avec un environnement supportant une fine granularité.

Par contre, une granularité applicative fine correspond à une application dont les activités représentent un faible nombre d'instructions élémentaires. Dans l'exemple précédent, cela peut correspondre à des dépendances entre les données au niveau de la ligne et de la colonne. Les environnements qui supportent une granularité fine exécutent ces applications de manière efficace, le problème se pose quand la granularité applicative est fine alors que la granularité exécutive est grosse. Une correspondance directe, qui ferait correspondre à chaque activité applicative une activité exécutive, n'est pas possible car le support d'exécution ne peut pas gérer un tel nombre d'activités. Il faut donc utiliser des mécanismes permettant une prise en charge des activités applicatives. Nous allons étudier différentes manières de les mettre en place dans la section 2.2.

Les applications de Branch&Bound qui sont au cœur de notre étude ont une granularité variable, qui peut être choisie par le programmeur. Intuitivement, ces applications peuvent donc être exécutés efficacement par les deux catégories de support. Nous allons voir dans la suite que le choix de cette granularité a un impact, qui est loin d'être négligeable, sur les performances de l'application.

2.2 Étude de parallélisation à gros grain

Les environnements offrant une grosse granularité étant les plus utilisés, il est logique de les utiliser pour paralléliser la résolution des problèmes hautement irréguliers. Comme nous l'avons vu, le problème principal à résoudre est d'établir une correspondance entre la granularité de l'application et celle du support d'exécution. Dans cette section nous allons présenter nos travaux et nos réflexions sur ce point précis.

Nous commencerons par étudier comment déterminer la granularité applicative pour une exécution efficace lorsque cette granularité peut être quelconque. Nous verrons ensuite le moyen pour passer d'un grain fin de calcul vers un gros grain. Nous insisterons en particulier sur les contraintes supplémentaires que cela crée dans la conception de l'application. Nous étudierons ensuite différents moyens de prendre en compte ces contraintes en illustrant, pour chaque méthode, notre discours par un exemple.

2.2.1 Impact de la granularité applicative

Le choix de la granularité des activités dans une application hautement irrégulière comme le Branch&Bound est un problème délicat qui a un impact direct sur les performances de l'application.

6. Un processus lourd n'est d'ailleurs qu'une coquillle accueillant un (dans le cas traditionnel, typiquement la fonction `main` d'un programme C) ou plusieurs (si la granularité du support est fine) processus légers.

La forme de la courbe de performances en fonction de la granularité est similaire à celle présentée sur la figure 2.3.

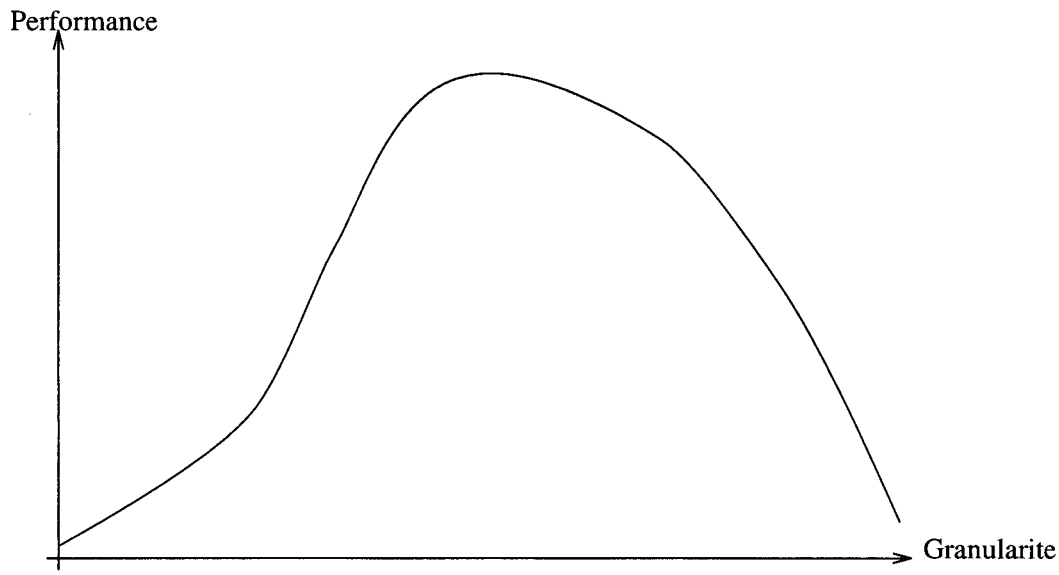


FIG. 2.3 - *Courbe typique de l'influence de la granularité sur les performances d'une application hautement irrégulière d'exploration arborescente*

Les performances sont mauvaises lorsque la granularité est faible car le temps passé à gérer l'activité est important par rapport au temps nécessaire à son exécution. Quand la granularité est très forte, les performances ne sont pas bonnes à cause de la sous-utilisation de l'architecture parallèle causée par le faible nombre d'activités générées. Le choix de la granularité assurant des performances optimales dépent de deux paramètres:

1. le nombre de processeurs,
2. la granularité « minimale » de l'application pour une exécution efficace qui dépend d'elle et de la granularité exécutive.

Nous avons illustré cette caractéristique sur un exemple en mesurant le temps pris par la résolution d'un problème d'affectation quadratique de taille 15 sur dix processeurs en utilisant le schéma centralisé que nous allons présenter dans la suite. La courbe résultante est représentée sur la figure 2.4. Sa forme confirme notre discours: les performances sont faibles pour une petite granularité, bonnes pour une granularité moyenne et mauvaises pour une élevée. Les valeurs de granularité qui définissent ces seuils (petit, moyen, élevé) dépendent directement de la granularité exécutive. Dans cet exemple la granularité exécutive est forte, il s'agit d'un environnement à gros grain.

Le choix de la granularité pour une bonne efficacité dépend de plusieurs facteurs qui sont liés à la fois à l'architecture sous-jacente et aux caractéristiques de l'application. De plus, comme nous le détaillons dans la suite, le choix de cette granularité influe aussi sur la surface explorée de l'arbre donc sur le temps total mis pour la résolution du problème: la granularité « naturelle » de ce type d'applications est, intuitivement, celle du nœud mais elle est, en général, inapplicable.

Pour montrer cette influence, nous allons présenter dans la suite deux méthodes assurant la correspondance entre une granularité applicative fine et exécutive grosse. Nous étudierons sur des

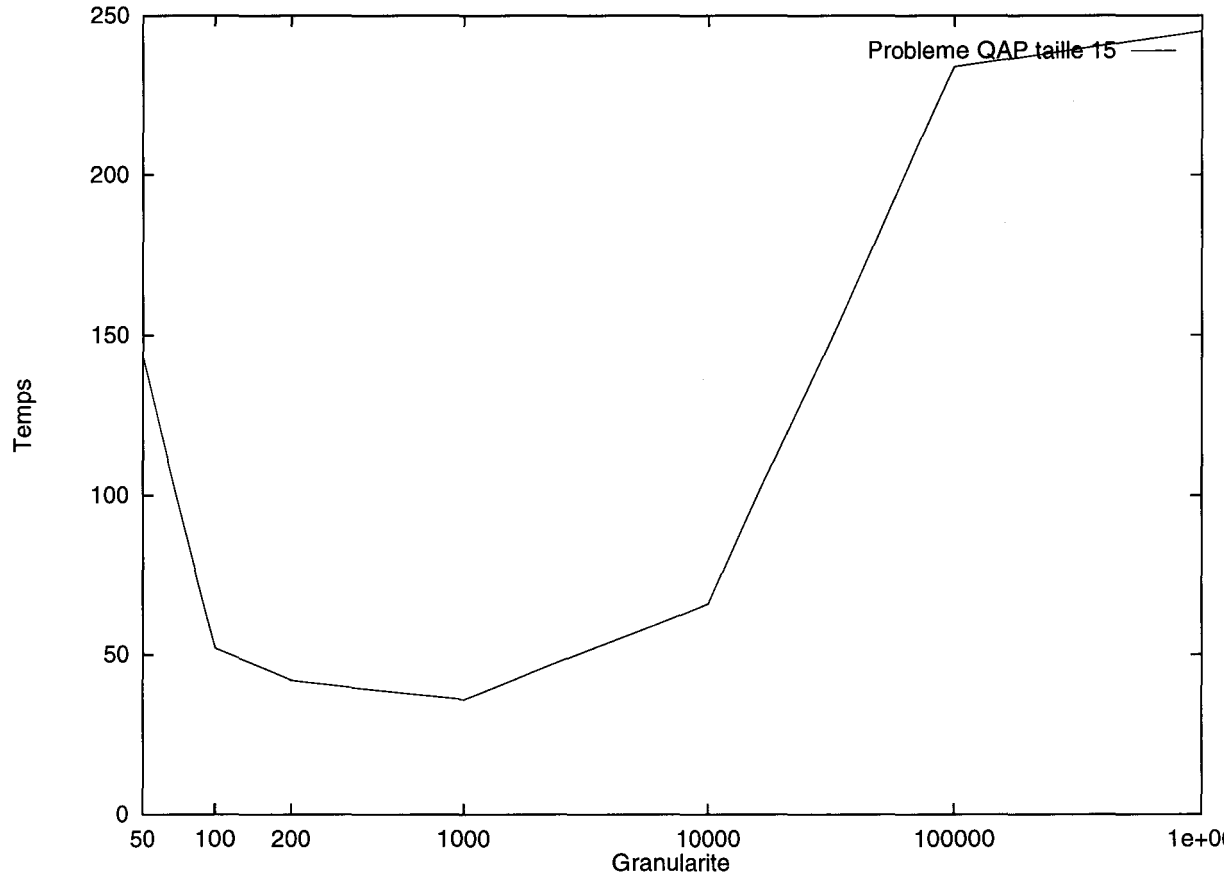


FIG. 2.4 - *Influence de la granularité sur le temps d'exécution de la résolution d'un problème d'affectation quadratique de taille 15*

résultats concrets l'influence du choix de la granularité applicative sur la taille de l'espace total exploré.

2.2.2 Transition du grain fin vers le gros grain

Le passage d'une granularité fine de calcul vers un modèle d'exécution à gros grain signifie que le niveau de parallélisme réel dans l'application sera inférieur au niveau potentiel, c'est-à-dire au nombre d'activités prêtes à être exécutées à un instant donné. Cela implique une gestion à deux niveaux des activités :

Activités en cours d'exécution Celles qui sont réellement exécutées par les processeurs de calcul de l'architecture parallèle.

Activités en attente Celles qui sont prêtes, qui pourraient donc être exécutés, mais qui ne peuvent pas l'être car le degré de parallélisme du support d'exécution ne le permet pas.

La gestion des activités du premier type est généralement prise en compte par le support d'exécution puisqu'elle consiste à exécuter un traitement sur des données puis à stocker (ou renvoyer, ou afficher ...) les résultats trouvés, qui peuvent être de nouvelles activités qui appartiennent alors

à la seconde catégorie. La gestion de ces activités en attente est plus complexe et constitue le point central de toute parallélisation[7] d'applications de cette classe. C'est ce que nous allons détailler dans la suite.

2.2.3 Utilisation d'une structure de données globale

D'une manière générale, les activités du second type sont stockées dans une structure de données d'où elles seront transmises au support d'exécution qui les exécutera. Ce fonctionnement est schématisé sur la figure 2.5.

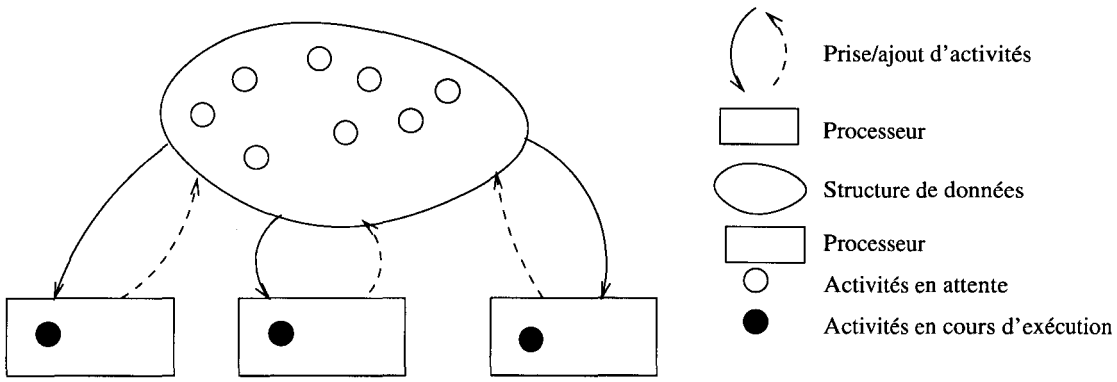


FIG. 2.5 - *Schéma général de parallélisation d'applications hautement irrégulières dans un environnement à gros grain*

Les opérations principales que cette structure doit fournir sont l'insertion et le retrait. Pour certains problèmes il existe une relation d'ordre entre les activités en attente, le respect de cette relation étant important pour les performances de l'application (cf. chapitre 5). Dans ce cas il est souvent, suivant les besoins de l'application, nécessaire de disposer de quelques opérations supplémentaires sur la structure telles que l'extraction d'un élément répondant à un critère donné, comme par exemple une valeur minimale.

L'efficacité de l'application parallèle est liée aux performances de cette structure de données globale qui est le véritable cœur de la parallélisation. Schématiquement la gestion de cette structure peut être de deux types :

Séquentiel : Un seul accès simultané à la structure est possible. Les accès parallèles sont donc sérialisés.

Parallèle : Plusieurs accès peuvent être effectués simultanément sur la structure. Cela nécessite une conception plus évoluée mais la rend plus adaptée aux applications parallèles.

De même le stockage de ces éléments peut se faire de deux façons différentes :

Groupé : Toutes les activités sont regroupées sur un seul processeur de l'architecture ; solution simple mais qui limite le nombre d'activités stockables simultanément à un instant donné.

Éclaté: L'ensemble des processeurs est utilisé pour stocker les activités. Cette méthode exploite mieux les capacités de l'architecture utilisée mais complexifie la gestion.

7. À gros grain...

Dans le cadre de nos travaux nous avons étudié et testé différents schémas possibles d'implantation de cette structure de données, séquentiel et parallèle, centralisé et distribué. Nous allons maintenant présenter ces travaux et les conclusions que nous en avons tirées. Nous commencerons par un schéma entièrement centralisé avant d'en introduire un ayant un degré de distribution plus important.

2.2.4 Schéma centralisé

Le schéma le plus connu et le plus utilisé est le schéma centralisé qu'on appelle communément maître-esclave. Son étude est l'objet de cette section.

2.2.4.1 Principe

Dans ce schéma d'implantation (voir figure 2.6), la structure de données est entièrement gérée par une seule activité et elle est localisée sur un seul processeur de l'architecture parallèle. C'est une structure intrinsèquement séquentielle. Dans la classification introduite ci-dessus, ce schéma a les caractéristiques suivantes : séquentiel et groupé.

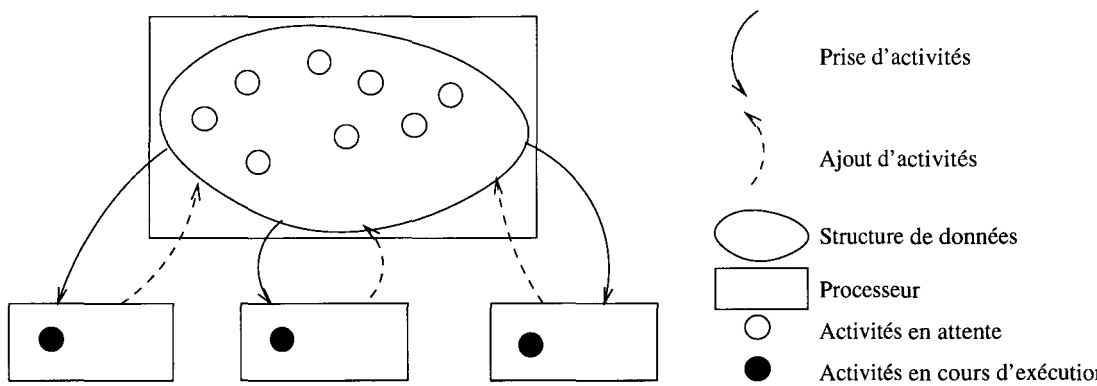


FIG. 2.6 - *Schéma centralisé d'implantation d'une structure de données parallèle globale*

Le nombre de systèmes utilisant ce schéma est trop élevé pour en permettre une énumération exhaustive ou même succinte. De manière arbitraire, citons deux systèmes développés au LIFL MARS[Haf98], ALBA[Mel97] et un, situé dans le cadre de l'optimisation combinatoire, développé en Angleterre[LM96].

2.2.4.2 Avantages

C'est un schéma qui est simple à implanter et dont le fonctionnement est facile à appréhender car il utilise un schéma de communication rudimentaire. C'est implicitement vers ce schéma de communication que poussent des environnements comme PVM, dans lesquel on a typiquement une activité qui se charge de l'initialisation de l'application, du lancement d'autant de tâches que l'architecture parallèle compte de processeurs puis qui surveille et coordonne le déroulement de l'exécution afin d'en déterminer la terminaison.

Cela en fait un schéma intuitif de communication quand le support d'exécution utilisé est à gros grain et utilise le passage de messages comme outil principal de distribution.

L'absence de parallélisme dans la structure de données elle-même évite les problèmes de gestion d'accès exclusif et simplifie donc sa conception. De même, l'implantation d'opérations spéciales, telle que la recherche d'une valeur minimum par exemple, a déjà été largement étudiée dans le cadre séquentiel et, avec ce schéma, tous ces résultats peuvent être utilisés directement. Cela en fait un bon candidat pour un passage « en douceur » d'une application purement séquentielle vers un environnement parallèle.

2.2.4.3 Inconvénients

L'inconvénient principal de ce schéma est justement sa séquentialité qui en fait le goulot d'étranglement de l'application quand le degré de parallélisme réel, c'est-à-dire le nombre de processeurs, de l'architecture est important et que le débit des communications est élevé. C'est rarement le cas sur des réseaux classiques de stations de travail mais cela arrive très souvent dans le cas de l'utilisation d'une machine parallèle dédiée ou de machines performantes reliées à des réseaux haut-débit, tels que Myrinet ou ATM. Dans ce cas, les performances sont pénalisées par le non-parallélisme de la structure elle-même car le débit du réseau fait que le processeur dédié à la gestion de la structure est saturé par les requêtes et devient donc un goulot d'étranglement pour l'application. Notons qu'il est possible de réduire son influence sur les performances en augmentant la granularité, ce qui aura pour effet de diminuer la fréquence des accès à la structure, mais il en résultera un déséquilibre encore plus grand entre la granularité « naturelle » de l'application et celle de son implantation.

Centraliser la gestion de la structure de données sur un seul site de l'architecture parallèle amène aussi des problèmes de contention des communications en un point du réseau. Un autre problème lié à cette caractéristique est la limitation de la taille des problèmes traitables par la taille de la mémoire du site sur lequel est stockée la structure. On peut une fois encore contourner cette limite en modifiant la granularité des activités mais cette solution n'est qu'un pis-aller.

Enfin, centraliser l'opération qui est au cœur d'une application distribuée sur un seul site n'est jamais une bonne chose en regard de la tolérance aux fautes car cela complique singulièrement l'implantation d'un fonctionnement en mode dégradé.

2.2.4.4 Performances

Dans la section suivante nous allons étudier les performances de cette approche en utilisant comme support applicatif un programme de résolution du problème, célèbre en optimisation combinatoire, de l'affectation quadratique[8]. La problématique de cette application est présentée dans l'annexe A.2 ; nous ne la répétons pas ici pour ne pas alourdir la lecture.

Comme nous l'avons mentionné précédemment, le problème principal qui se pose est celui de la granularité. Dans les expériences dont les résultats sont présentés, nous avons décidé de la considérer comme un paramètre de l'exécution. Durant l'exploration d'un sous-arbre deux possibilités existent :

- Si la taille[9], du sous-arbre est inférieure au paramètre de granularité donné alors il sera complètement exploré séquentiellement,

- Sinon le développement sera effectué sur un nombre de nœuds égal au paramètre, les sous-arbres non explorés étant stockés dans la structure de données globale pour être explorés ultérieurement[10].

8. Cette application nous a été transmise par l'équipe PNN de PR*i*SM, Versailles.

9. En nombre de nœuds.

10. En fait, pour des raisons d'efficacité, le fonctionnement est légèrement différent car seuls les sous-arbres ayant une hauteur suffisante sont stockés, les autres sont explorés immédiatement pour éviter de stocker des nœuds cor-

Les tests ont été réalisés en utilisant des instances tirées de la célèbre QAPLIB [BKF91] qui est une collection de problèmes classiques utilisée par la communauté comme *benchmark*. Pour tester l'efficacité de l'algorithme nous l'avons utilisé pour résoudre le problème Nugent de taille 16 en nous focalisant sur l'accélération[11]. L'architecture parallèle utilisée pour ces tests, ainsi que pour ceux qui concernent l'implantation distribuée, est un ordinateur IBM SP2[12], le support d'exécution est PVM version « domaine public ». Les résultats de ces mesures se trouvent sur la figure 2.7 et ils sont assez bons, notamment avec une efficacité proche de 100% pour 16 processeurs.

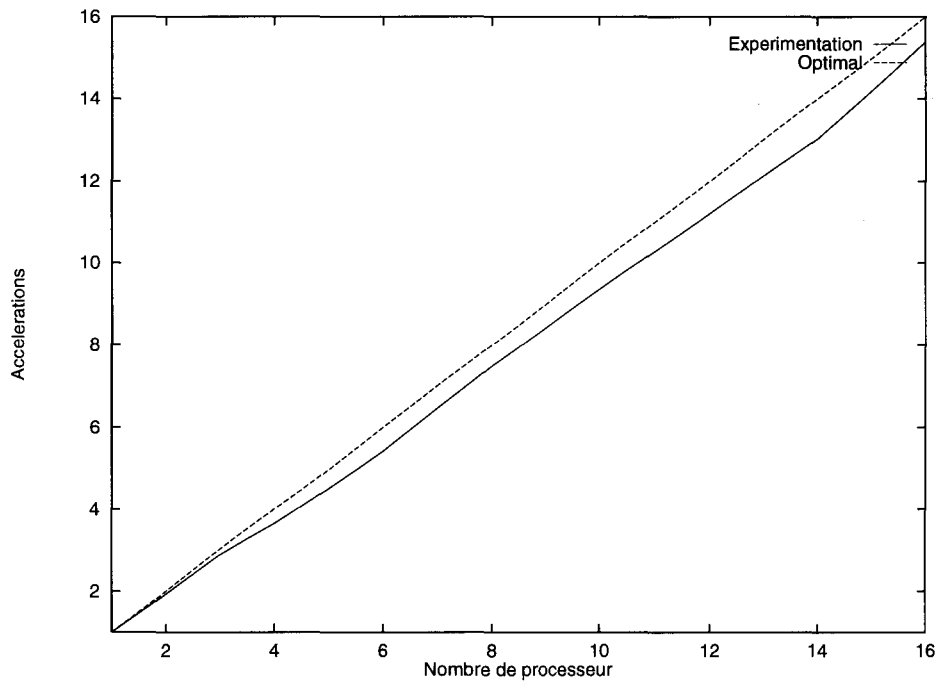


FIG. 2.7 - *Mesures d'accélération pour la résolution du célèbre problème d'affectation quadratique Nugent de taille 16*

Nous commenterons plus en détail les résultats du tableau 2.2 dans la section 2.2.5.4 page 37 où nous les comparerons avec les résultats d'une autre implantation qui est elle non centralisée. Cela nous éclairera plus particulièrement sur le rôle joué par la granularité sur les performances.

Durant la période pendant laquelle nous conduisions ces tests, nous avons eu accès à une machine parallèle disposant d'un potentiel élevé de parallélisme, un Cray T3D avec 256 processeurs. Nous avons pu alors observer le comportement de ce schéma de parallélisation sur un nombre plus important de processeurs. Les résultats obtenus ont été assez surprenants, avec des temps variant du simple au double dans des conditions (en apparence) exactement identiques, c'est-à-dire avec le même programme, le même problème à résoudre et une machine entièrement dédiée, donc sans perturbations extérieures dûes aux autres usagers. En observant attentivement les fichiers de logs de l'exécution, nous avons découvert que ces résultats erratiques s'expliquaient par le goulot d'étranglement que représente la structure de données globale centralisée. Dans certains conditions, liées

respondant à de trop petits sous-arbres et dont le coût de gestion serait alors prohibitif au regard du coût de leur exploration. Nos mesures ont cependant montré que le nombre de nœuds ainsi explorés était très faible.

11. Speed-up en anglais.

12. Nous remercions le LMC IMAG pour avoir mis ce matériel à notre disposition.

Nom du problème	Valeur de la solution	Temps (secondes)	Nombre de nœuds explorés
esc16a	68	59	525.285
esc16b	292	333	10.221.885
esc16c	160	2087	63.649.543
esc16d	16	1793	43.884.805
esc16e	28	38	452.266
esc16g	26	1752	55.195.289
esc16h	996	9619	332.385.306
esc16i	14	3569	130.559.757
esc16j	8	988	30.596.415
nug20	2570	13423	121.169.074
rou20	725522	68342	445.097.368

TAB. 2.2 - *Performances de la résolution des problèmes de taille 16 d'Eschermann et Wunderlich ainsi que d'autres problèmes classiques de taille 20 avec une approche centralisée. Le problème esc16f est si petit que les résultats ne sont pas présentés, car peu intéressants.*

à l'initialisation de l'application et qui dépendent de la topologie des processeurs utilisés[13], les processeurs demandaient tous du travail aux mêmes moments et cela se reproduisait à plusieurs reprises durant une même exécution, ce qui en mettait certains en situation de famine et donc pénalisait les performances. Cette expérience « grandeur nature » met en évidence le défaut principal de ce schéma qui avait été souligné dans la section précédente.

C'est dans le but d'éliminer cette importante restriction que nous avons étudié des solutions proposant un stockage complètement distribué. Cette étude est décrite dans la section suivante.

2.2.5 Schéma distribué

Des solutions existent pour s'affranchir des limitations qui constituent les inconvénients de l'approche précédente. Elles sont basées sur la *décentralisation* de la gestion de la structure de données globale. La réflexion sur cette classe de méthodes est issue de la non-adaptation de la méthode précédente aux architectures distribuées qui ont vu le jour lors de la dernière décennie. Idéalement, une structure partagée stockée dans une mémoire partagée, virtuellement ou non, serait parfaite mais le coût d'une telle solution serait trop excessif. C'est pour répondre à ce problème qu'est apparu le schéma d'implantation distribuée que nous allons présenter maintenant.

2.2.5.1 Principe

Cette méthode revient à gérer un espace mémoire distribué sur l'ensemble des processeurs au niveau applicatif. L'utilisation d'un mécanisme plus général, qui serait élément du support d'exécution par exemple [Dum97], est à proscrire pour des raisons de performance. En effet, les besoins des applications visées sont trop rudimentaires pour nécessiter l'utilisation d'un support générique forcément plus coûteux.

13. Le réseau d'interconnexion du Cray T3D est une grille 3D torique. Une partie des processeurs de cette grille est allouée, de manière exclusive et dynamique, à l'exécution d'une application. La géométrie de la partition de processeurs allouée peut varier entre les exécutions. Par exemple, pour 128 processeurs, une exécution peut se dérouler sur une grille 4*8*4 ou sur une grille de 8*4*4, ce qui modifie les temps de communication.

Le principe général de ce schéma est de ne plus avoir une activité chargée de gérer les fonctionnalités de la structure de données mais un ensemble d'activités coopérantes comme représenté sur la figure 2.8.

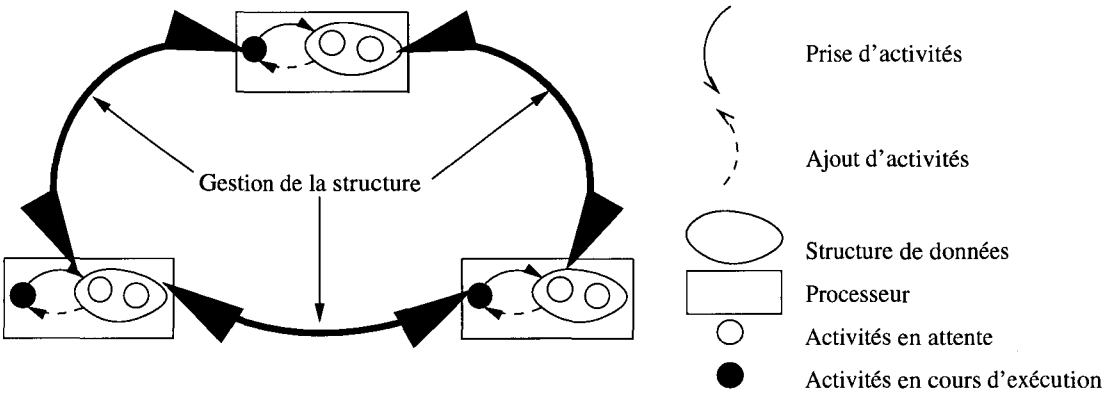


FIG. 2.8 - *Schéma distribué d'implantation d'une structure de données parallèle globale*

Dans ce cas, deux types de tâches d'une même application cohabitent sur les processeurs :

Les tâches applicatives : qui exécutent les activités calculatoires de l'application. Elles sont symbolisées par un point noir sur la figure 2.8.

Les tâches administratives : s'occupent de la gestion de la structure de données. Ce sont elles qui s'occupent de la communication entre les nœuds de l'architecture parallèle.

Le nombre de tâches applicatives par processeur conditionne le degré de parallélisme de l'application. Bien que ce nombre puisse être quelconque en théorie, en pratique il est très souvent égal à 1 car un nombre plus élevé est trop complexe à gérer dans un environnement à gros grain. Nous revenons sur ce point dans la suite. De manière duale, le nombre de tâches administratives par processeur fixe le degré de parallélisme de la structure de données, c'est-à-dire le nombre maximal d'opérations pouvant s'exécuter concurremment sur cette dernière à un instant donné. Il est aussi, généralement, égal à 1.

Contrairement au schéma étudié précédemment, celui-ci permet un réel parallélisme des opérations sur la structure. C'est une conséquence de la distribution de sa gestion qui lui permet d'être accédée, tant par les tâches applicatives que par les tâches administratives, simultanément par tous les processeurs.

Deux familles d'implantation de ce schéma peuvent être distinguées en utilisant comme critère la gestion des données :

Distribution des données : C'est le cas représenté sur la figure 2.8. Chaque processeur gère une certaine quantité de données et communique avec les autres suivant un protocole *ad-hoc* pour « globaliser » la structure, ce protocole étant typiquement chargé de l'équilibrage de charge. C'est le schéma couramment utilisé sur les architectures parallèles à mémoire distribuée telles qu'un réseau de stations de travail. Il rentre dans la catégorie du stockage éclaté dans la terminologie présentée auparavant.

Centralisation des données La structure est entièrement contenue dans la mémoire d'un nœud et les processeurs y accèdent en parallèle selon un protocole, typiquement des verrous, pour

conserver sa cohérence. Ce type d'implantation, appelé aussi groupé, est adapté aux architectures à mémoire partagée[RK88, Man92], éventuellement virtuellement, et a aussi été étudié dans le cadre des machines à mémoire distribuée[Mey, LC96].

Dans les deux cas, chaque élément stocké dans la structure n'y est stocké qu'une fois, il n'y a pas de réplication des données sauf dans le cas, très particulier, de l'utilisation d'un cache.

Le degré de parallélisme maximal de la structure, donc le nombre maximal de tâches administratives s'exécutant simultanément, dépend de la famille à laquelle elle appartient. Dans le premier cas, il est supérieur ou égal au nombre de nœuds de l'architecture car il y a au moins une tâche par nœud[14]. Dans le second cas, ce nombre dépendra du type de structure utilisée. Dans le cas d'un tas, par exemple, ce nombre sera proportionnel à la profondeur de l'arbre[15].

Nous étudions dans la section 2.2.5.4 un exemple d'implantation du premier type.

2.2.5.2 Avantages

L'intérêt d'utiliser une structure parallèle sur une architecture distribuée est de pouvoir disposer d'une réelle concurrence des opérations. La structure étant distribuée sur les nœuds, deux opérations d'insertion, par exemple, peuvent se faire au même instant. Le goulot d'étranglement que représentait dans l'implémentation précédente l'accès à la structure est supprimé ou en tout cas ses effets sont largement atténués.

Le coût supplémentaire induit par la gestion de la distribution est compensé par le gain procuré par la réduction du temps de réponse qui est la conséquence de la concurrence des accès. Il est de plus possible d'effectuer ces opérations de distribution, typiquement de l'équilibrage de charge, sur une partie de la structure, entre deux nœuds par exemple, pendant que des activités sont ajoutées ou soustraites, en effectuant une sorte de recouvrement administration/utilisation semblable au recouvrement calcul/communication classiquement implanté dans les applications distribuées.

De plus, la taille des problèmes traitables[16] est plus importante que dans le schéma précédent car le stockage de la structure se fait sur l'ensemble de la mémoire disponible de l'architecture au lieu d'être localisé sur un site seulement. La distribution de sa gestion permet aussi d'envisager l'implantation de mécanismes de tolérance aux fautes performants car, contrairement au schéma étudié précédemment, la panne d'un processeur ne fait perdre qu'une partie de la structure. Il devient alors possible soit de continuer l'exécution en mode dégradé soit, par la mise en place d'un mécanisme de « mirroring » par exemple, de continuer comme si rien ne s'était passé après une éventuelle phase de redistribution du travail.

2.2.5.3 Inconvénients

L'inconvénient principal de ce schéma d'implantation réside dans sa complexité. En effet, dans le type d'environnement visé, parallélisme à gros grain et communications par envoi de message, le seul moyen de disposer de deux types d'activités s'exécutant concurremment est d'utiliser au moins deux processus à gros grain par nœud comme illustré sur la figure 2.9. Cela augmente la complexité du protocole de communication qui doit alors gérer deux types de requêtes.

Comme nous l'avons mentionné dans la section précédente, le degré de parallélisme de la structure dépend de son type, de la distribution ou de la centralisation du stockage des données. Cependant, dans le cas où ce stockage est distribué, obtenir un degré de parallélisme supérieur au

14. On se place dans le cas où chaque nœud de l'architecture est utilisé pour la gestion de la structure.
15. Plus exactement, il sera inférieur ou égal à $\frac{profondeur}{2}$.
16. À granularité constante...

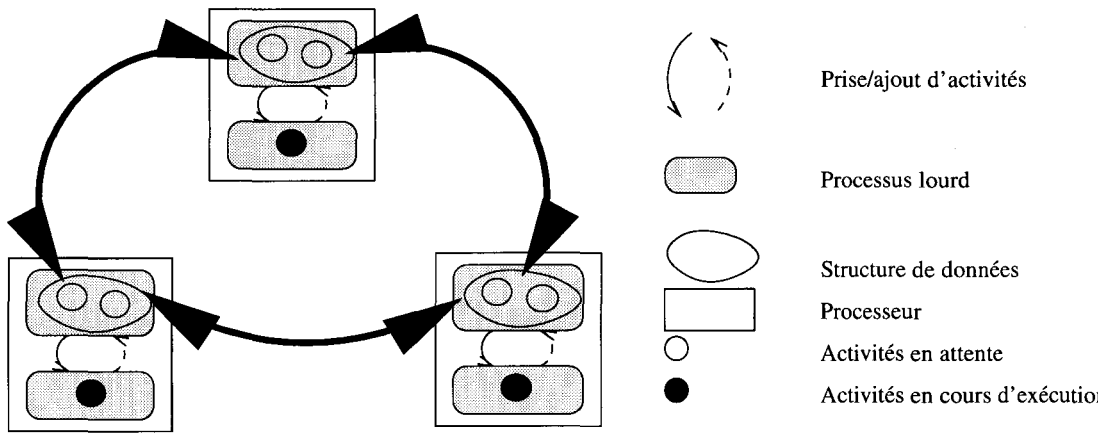


FIG. 2.9 - *Schéma de communication complet pour l'implantation distribuée d'une structure de données globale sur une architecture parallèle sans mémoire partagée*

nombre de nœuds signifie, dans les environnements UNIX courants, disposer d'un grain plus fin de parallélisme. Les problèmes d'accès concurrents à des données partagées surgissent alors, obligeant à utiliser des mécanismes de verrouillage de certaines parties de la structure. Ces problèmes s'ajoutent à ceux posés par la distribution elle-même. De la même façon que les mécanismes de verrouillage limitaient le degré de parallélisme dans le cas d'un stockage centralisé, ils limitent aussi le degré dans le cas distribué. Dans ce cas cette limitation est moins contraignante car l'essentiel du parallélisme est fourni par la distribution et non par les opérations concurrentes sur un processeur.

2.2.5.4 Exemple d'implantation

Dans cette section nous allons présenter une structure de données utilisant ce schéma distribué de communication. Le contexte visé est celui des machines parallèles à mémoire distribuée dédiées, comme l'IBM-SP2, ou partagées entre les utilisateurs, comme un réseau de stations de travail.

Caractéristiques Cette structure a été développée dans le cadre de la parallélisation d'une application d'optimisation combinatoire visant à résoudre le problème de l'affectation quadratique[17]. La résolution exacte de ce problème s'effectue en explorant entièrement un arbre de recherche suivant la méthode Branch&Bound, l'arbre pouvant être découpé en sous-arbres de manière quelconque, sans dépendances autres que la parenté. Cet algorithme se prête donc bien à la parallélisation, il est du type hautement irrégulier (voir section 2.1) avec des dépendances minimales entre les activités.

Le protocole de communication entre les tâches applicatives et administratives est identique à celui utilisé avec le modèle centralisé, nous le detaillerons donc pas. La gestion de la granularité est elle-aussi identique.

À chaque nœud est associée une valeur de priorité dont le respect est important pour favoriser les anomalies d'accélération. Nous revenons sur ce point dans le chapitre 5, pour l'instant nous considérerons, pour simplifier, qu'une relation d'ordre existe entre les nœuds et que ceux ayant la valeur la plus élevée doivent être traités en priorité [LW86, dBKT95].

17. Ce travail a été présenté dans [DLCMM96, DM97].

Conformément au cadre de travail fixé, nous nous plaçons dans le cas où le stockage de la structure est totalement distribué : chaque processeur, et la mémoire qui lui est associée, contenant une partie des données. Le problème qui se pose alors est l'équilibrage de la charge, c'est-à-dire, dans ce cas, le nombre de nœuds stockés sur chaque processeur. Cet équilibrage doit se faire non seulement sur le nombre mais aussi sur la « qualité », la priorité, des nœuds, il doit être *quantitatif* et *qualitatif*. En effet, pour qu'une activité soit exécutée par un processeur il faut qu'elle soit dans la zone de stockage de ce processeur ; donc si tous les nœuds ayant la plus haute priorité sont stockés sur le même processeur d'autres, moins prioritaires mais situés sur un processeur différent, seront explorés avant eux, ce qui est à éviter pour obtenir de bonnes performances.

De plus, l'activité de gestion doit être faite de façon concurrente avec le déroulement des activités de calcul afin de ne pas bloquer un processeur sur une opération de communication, typiquement l'attente d'un message, sans qu'une activité de calcul ne puisse occuper ce temps d'inactivité menant ainsi à une sous-utilisation du processeur.

Pour résumer, les caractéristiques de l'implantation de la structure sont les suivantes :

Distribution : Le stockage et la gestion de la structure sont entièrement distribués.

Équilibrage : La répartition des nœuds stockés est uniforme sur les différents processeurs tant suivant le critère du nombre que celui de la qualité.

Asynchronisme calcul/gestion : Les activités de gestion de la structure fonctionnent de manière totalement asynchrone avec les activités de calcul.

Implantation Dans la liste précédente, la première fonctionnalité à prendre en compte pour choisir la structure en termes d'activités système[18] est celle d'asynchronisme. Le seul moyen, dans un environnement à gros grain comme ceux visés, pour avoir une réelle concurrence[19] entre plusieurs activités est de créer des processus indépendants pour exécuter ces dernières. Nous aurons donc besoin de deux processus par processeur, un en charge de la gestion de la structure des données et l'autre exécutant les activités applicatives en cours d'exécution suivant la terminologie fixée dans la section précédente. Seul le premier communiquera avec « l'extérieur » alors que le second ne s'adressera qu'à ce dernier, les communications entre les tâches de calcul et celles de gestion seront donc uniquement locales.

Les deux autres caractéristiques impliquent la mise en place d'un mécanisme de régulation de charge et donc la nécessité de définir un protocole entre les activités de communication pour effectuer cette régulation. Dans l'optique de maximisation des performances qui est la nôtre, il est important que cet équilibrage ne se fasse que lorsque cela est nécessaire. En particulier, quand la charge est relativement bien équilibrée la régulation ne doit pas se faire. Cela implique l'utilisation d'une stratégie à seuil, les seuils étant choisis de manière à assurer l'équilibrage sans pénaliser les performances. Dans notre protocole, deux seuils permettent de distinguer trois états de charge : sous-chargé, normal et sur-chargé comme illustré sur la figure 2.10.

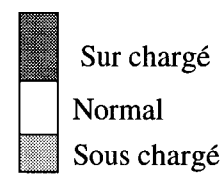


FIG. 2.10 - *Les états*

Le concept clé de notre régulation est la *réaction* au déséquilibre. Quand une modification d'état se produit localement, le nouvel état est diffusé globalement et si un autre processeur peut agir,

18. c'est-à-dire de processus UNIX.

19. Réelle aux yeux du programmeur, il est clair que sur une machine monoprocesseur cette concurrence ne peut se faire qu'en temps partagé.

c'est-à-dire demander ou fournir des nœuds, pour rétablir l'équilibre il le fait. Voilà pour le principe, mais son application directe risque d'être peu efficace pour deux raisons :

L'inefficacité de la diffusion : L'opération de diffusion est rarement efficace sur les environnements couramment utilisés[20]. Il ne s'agit que d'un simple envoi multiple de messages.

Le risque « d'inondation» : Si tous les processeurs réagissent au même moment pour rétablir l'équilibre, le processeur émetteur risque d'être noyé et de se trouver dans un état de charge très élevé nécessitant de nouveau une procédure d'équilibrage.

Pour ces raisons il est nécessaire de ne pas envoyer à tous les processeurs en même temps une information de modification de charge. Nous avons donc opté pour une structure de communication entre les tâches de communication formant un anneau, pour la diffusion des modifications d'état. De cette manière, il y a une séquentialité dans le passage des modifications d'état entre les processeurs ce qui évite le phénomène d'inondation décrit précédemment. Quand un tel message est reçu par un processeur celui-ci y réagit soit en envoyant des nœuds soit en se contentant de le propager. L'action qu'il entreprend dépend de son état comme récapitulé dans le tableau 2.3. Lorsqu'un transfert doit se faire, l'activité qui envoie des nœuds choisira toujours les plus prioritaires de sa file afin que l'ils soient traîtés le plus tôt possible dans l'exécution.

État du receveur	Message reçu	
	Sous chargé	Normal
Sous chargé	Propage le message	Propage le message
Normal	Transmet des nœuds à l'envoyeur	Propage le message
Sur chargé	Transmet des nœuds à l'envoyeur	Transmet des nœuds à l'envoyeur

TAB. 2.3 - *Action entreprise à la réception d'un message de changement d'état*

Au vu du tableau on constate que le schéma de communication en anneau n'est utilisé que pour les messages de modification d'état, la transmission de travail réalisant effectivement l'équilibrage est lui effectué suivant un principe de maillage complet. Le risque d'inondation a été minimisé en sérialisant la transmission des informations d'état par l'utilisation de l'anneau. Il faut cependant noter que ce risque n'est pas complètement éliminé car cela nécessiterait la mise en place de coûteux mécanismes d'élection visant à décider quel serait l'*unique* partenaire d'un processeur durant une phase[21] d'équilibrage. L'indéterminisme du comportement de l'application oblige à refaire ce processus d'élection très souvent, le rendant peu efficace en pratique, c'est pourquoi nous avons opté pour ce mécanisme plus simple.

Nous pouvons résumer les caractéristiques de notre approche ainsi :

Distribution Le stockage et l'accès aux données contenues dans la structure sont distribués sur l'ensemble des processeurs de l'architecture.

Concurrence dans l'accès aux données Le nombre d'accès simultanés à la structure ne dépend plus que du nombre de processeurs.

Équilibrage Un mécanisme d'équilibrage garantit que le stockage des nœuds se fait de façon relativement uniforme sur les processeurs.

20. Bien que les architectures sous-jacentes proposent souvent un mécanisme efficace ...

21. Le terme phase est ici un abus de langage car il n'y a pas réellement de notion de phases dans notre algorithme. On devrait plutôt parler de moment.

Asynchronisme calcul/répartition Les activités de calcul et celles de répartition fonctionnent de manière asynchrone, en particulier quand une communication bloquante est en cours le calcul continue. En d'autres termes notre approche propose un réel recouvrement calcul/communication.

Nous allons maintenant en étudier les performances.

Performances Dans cette section nous allons étudier les performances que nous avons obtenues en implantant l'algorithme décrit dans la section précédente. L'application que nous avons utilisée pour cela est la même que pour l'étude de l'approche centralisée présentée dans la section 2.2.4.4 page 29. Pour confirmer ces bonnes performances, nous avons mesuré le nombre de nœuds développés sur chaque processeur lors de la résolution d'un problème conséquent. L'histogramme résultant est présenté sur la figure 2.11.

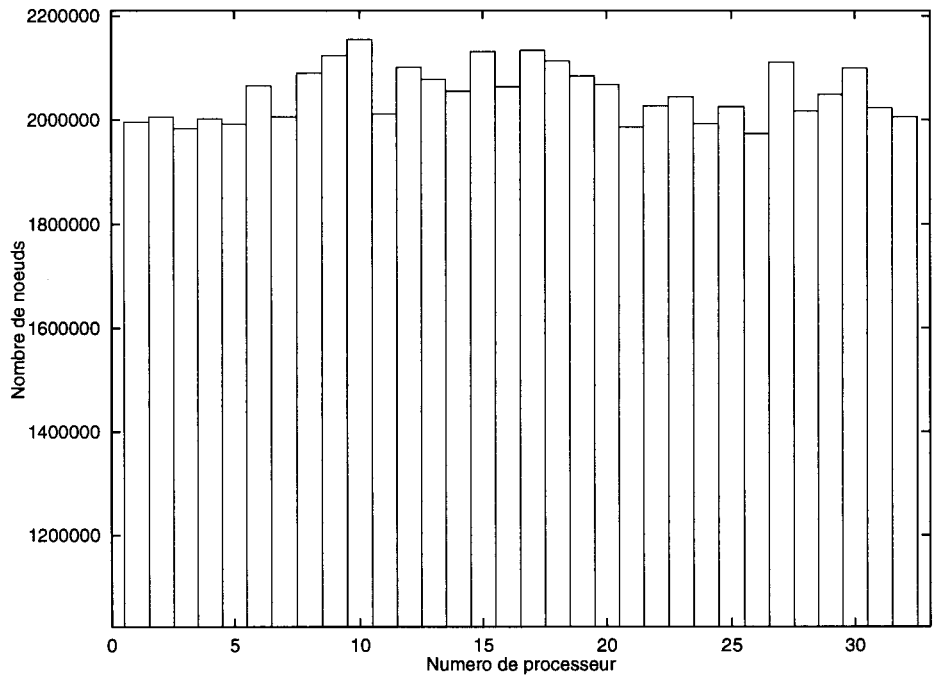


FIG. 2.11 - *Répartition des nœuds pour la résolution d'un problème d'affectation quadratique dans un contexte à grosse granularité de parallélisme avec un schéma de conception entièrement distribué*

Les performances en terme d'accéleration sont similaires à celle du modèle centralisé, c'est-à-dire avec une efficacité comprise entre 95% et 100%, ceci montre que l'algorithme d'équilibrage est efficace en terme quantitatif, c'est-à-dire qu'il maintient les processeurs actifs pendant un maximum de temps. Les seuils sont fixés à un pour définir un processeur normalement chargé et au-dessus de trois (non compris) il devient surchargé.

Dans cette section, nous allons plutôt nous focaliser sur l'aspect qualitatif, nous entendons par là sur la pertinence du travail effectué. En effet dans ce type d'algorithmes il est important de « faire du travail », explorer des sous-arbres, qui soit le plus utile possible donc qui ait le plus de chances[22] de mener à un bon résultat. L'algorithme d'équilibrage et la finesse de découpage

22. L'usage d'une fontion heuristique pour déterminer ce degré d'utilité le rend spéculatif.

(la granularité), nous y reviendrons dans la suite, jouent un rôle important dans cette optique: chaque tâche applicative s'adressant toujours à la tâche administrative qui se trouve sur le même processeur, il faut que les sous-arbres les plus prioritaires, ceux dont le nœud racine a la meilleure valeur d'évaluation partielle, soient répartis de façon homogène sur les processeurs pour que chacun effectue le travail le plus intéressant. La granularité rentre aussi en jeu pour une raison différente: en diminuant la granularité, on augmente le nombre de sous-arbres en attente d'exploration et donc la chance qu'un très intéressant soit choisi. L'originalité de l'approche choisie pour la résolution de ce problème tient d'ailleurs dans cette caractéristique: alors que les explorations de sous-arbre se font suivant une stratégie « profondeur d'abord »[23], le choix d'un sous-arbre se fait sur la valeur de son évaluation partielle.

La stratégie générale est donc une combinaison de ces deux stratégies, la profondeur d'abord étant la plus adaptée, pour des raisons de coût, au problème traité mais l'utilisation de la stratégie meilleure d'abord, avec une granularité beaucoup plus grosse cependant, est un plus pour trouver un bon résultat rapidement. En diminuant la granularité de parcours, on augmente son influence et les résultats expérimentaux du tableau 2.4 montrent que le gain obtenu, en terme de nombre de nœuds développpés, est réel. Pour faciliter la lecture, nous rappelons les résultats avec la méthode centralisée en regard de ceux de la méthode distribuée.

Problème	Approche centralisée			Approche distribuée			Gain (%)
	Temps	Nb, de nœuds	Granularité	Temps	Nb. de nœuds	Granularité	
esc16a	59	525.285	50000	39	451.631	1000	14
esc16b	333	10.221.885	50000	292	10.197.329	1000	<1
esc16c	2087	63.649.543	50000	1792	62.571.257	5000	2
esc16d	1793	43.884.805	50000	1147	39.463.020	50000	10
esc16e	38	452.266	50000	48	434.733	1000	4
esc16g	1752	55.195.289	50000	1605	51.531.127	5000	7
esc16h	9619	332.385.306	50000	9576	331.908.440	10000	<1
esc16i	3569	130.559.757	50000	2687	121.855.947	5000	7
esc16j	988	30.596.415	50000	948	30.624.955	10000	Négatif
nug20	13423	121.169.074	50000	14949	120.278.528	10000	<1
rou20	68342	445.097.368	3000000	67591	444.847.900	300000	<1

TAB. 2.4 - *Performances de la résolution des problèmes de taille 16 d'Eschermann et Wunderlich et d'autres problèmes de taille 20 avec une approche centralisée et une approche distribuée*

Les gains apportés par l'utilisation d'une méthode distribuée, en partie liés à la plus faible valeur de granularité qu'elle permet, sont donc significatifs mais le coût, en terme de développement, est réel comme l'a montré la description de la partie implantation de cette section. La solution peut résider dans un compromis entre ces deux approches, compromis réalisé par des approches nommées hybrides qui font l'objet de la section suivante.

2.2.6 Approches hybrides

Dans les paragraphes précédents nous avons présenté les deux grandes approches de parallélisation pour cette classe d'applications irrégulières. Pour résumer notre discours, on peut dire que

23. L'annexe A.1 page 141 détaille les différentes stratégies de parcours de l'arbre de recherche et leurs caractéristiques.

le prix, tant au moment du développement qu'à celui de l'exécution, de la plus grande complexité du modèle distribué est justifié par les deux bénéfices suivants :

La taille des problèmes traitables qui est augmentée car l'ensemble de la mémoire de l'architecture parallèle est utilisé au lieu de celle d'un seul processeur.

Le degré de concurrence de la structure de données globale.

Pour conserver ces bénéfices tout en diminuant la complexité de gestion, différentes approches « hybrides » ont été proposées dans la littérature. Nous allons en détailler deux, une approche hiérarchique et une utilisant un stockage par indirection.

2.2.6.1 Approche hiérarchique

Le but de cette approche est d'augmenter le degré de parallélisme de la structure de données sans pour autant augmenter la complexité de sa gestion et donc en conservant à la base un schéma centralisé maître/esclave. Pour cela l'idée est d'utiliser une hiérarchie, typiquement un arbre, de maîtres, chaque maître étant lui-même dirigé par celui de niveau supérieur, jusqu'au niveau le plus élevé qui contrôle l'ensemble. Ce schéma est représenté sur la figure 2.12. Il est utilisé, par exemple, dans [SV, AGF94].

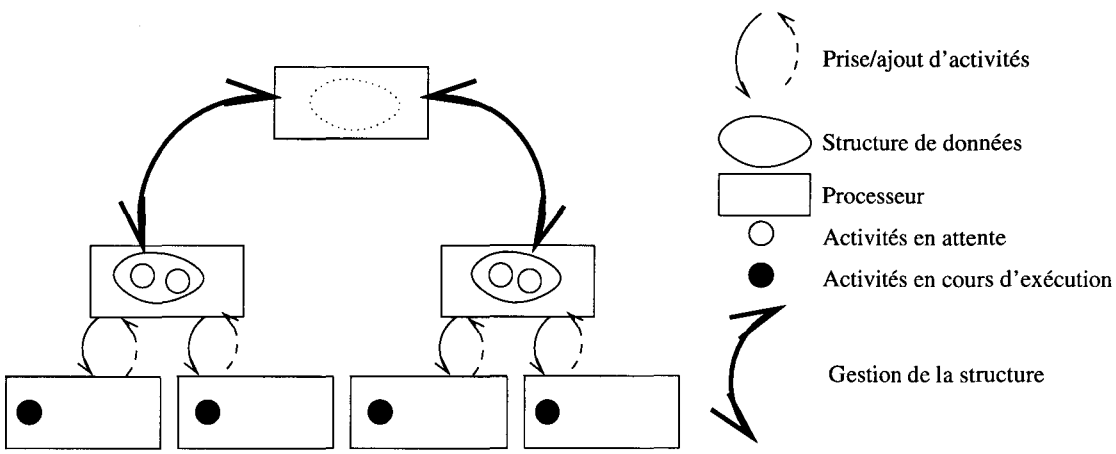


FIG. 2.12 - *Schéma hiérarchisé d'implantation d'une structure de données parallèle globale*

Son principe est d'éclater la gestion de la structure tout en y conservant une notion hiérarchique. Cela permet de conserver une certaine facilité de conception tout en distribuant le stockage. Différentes façons de gérer la structure sont envisageables, par exemple :

Distribution du stockage au dernier niveau Les activités ne sont stockées qu'au dernier niveau de l'arbre des maîtres. Les niveaux supérieurs servent à assurer l'équilibre du stockage entre les processeurs. C'est le cas présenté en figure 2.12.

Stokage totalement distribué Chaque nœud de l'arbre des activités contient une partie de la structure de données. L'équilibrage est donc réalisé globalement par tous les processeurs.

De la même manière, différents protocoles d'équilibrage peuvent être envisagés, leurs caractéristiques principales étant :

Synchrone ou asynchrone L'équilibrage se déroule de façon concurrente avec les opérations de calcul. La séparation des activités de calcul et de gestion en différentes activités, qui seront typiquement des processus UNIX, permet cette fonctionnalité.

À la demande ou continu On appelle équilibrage à la demande celui qui ne se fait que lorsque l'un des esclaves demande du travail au maître auquel il est associé et que celui-ci ne peut lui en fournir. Par opposition, un équilibrage continu se déroule en permanence et vise à anticiper les déséquilibres pour éviter que la situation évoquée précédemment n'apparaisse.

Les avantages liés à cette approche ont déjà été évoqués : relative facilité de conception, plus grande utilisation des ressources disponibles, possibilité d'une régulation asynchrone avec le calcul.

Les inconvénients sont ceux liés à l'utilisation d'un schéma maître/esclave : peu de tolérances aux fautes et, dans certains cas, risque d'engorgement du maître situé au sommet de la hiérarchie amenant une perte de performances.

Cette approche représente cependant un bon compromis entre la facilité de développement du schéma centralisé et les caractéristiques du schéma distribué. Pour atteindre le même but, une autre approche a été récemment proposée par Gengler et Diderich. Son étude fait l'objet de la section suivante.

2.2.6.2 Nommage centralisé et stockage réparti

Ce schéma est issu d'une réflexion basée sur un double constat :

1. L'apparition des machines parallèles utilisant le paradigme de la mémoire virtuellement partagée. L'apparition de telles machines, rendue possible par l'augmentation des performances des réseaux de communication, permet des accès mémoires distant dont le coût n'est que deux ou trois fois plus élevé qu'un accès mémoire local.

2. Le point clé dans le schéma distribué réside dans la possibilité de distribuer le stockage plus que dans l'augmentation du degré de parallélisme de la structure.

Pour mieux comprendre ces observations, nous allons détailler ces deux points. Par mémoire virtuellement partagée on entend que n'importe quel processeur peut accéder directement à la mémoire d'un autre processeur soit pour y lire des données, soit pour en écrire. Des machines comme celles de la famille MPP de CRAY, à laquelle appartiennent les modèles T3D et T3E, proposent ce paradigme comme mécanisme de base pour la communication entre processeurs en fournissant une bibliothèque nommée `shmem` [BK94]. Avec une efficacité suffisante, c'est-à-dire d'un facteur 2 à 3 par rapport à un accès mémoire local, cela permet d'envisager de nouvelles méthodes algorithmiques pour les problèmes qui nous intéressent. En effet, il est clair que la méthode la plus simple et la moins coûteuse pour le stockage des activités en attente est la centralisation. C'est uniquement pour des raisons d'implantation liées aux architectures visées, distribuées et faiblement couplées, que les méthodes de stockage totalement distribuée sont étudiées.

En partant de ces observations, Gengler et Diderich proposent dans [DG96] une méthode permettant de distribuer le stockage de données tout en conservant une gestion centralisée. Ils utilisent pour cela un principe d'indirection illustré sur la figure 2.13 que nous allons maintenant détailler.

Le principe est de conserver un unique maître auquel les autres processeurs s'adressent pour stocker des activités en attente ou pour en obtenir, mais ce maître ne conserve que des références

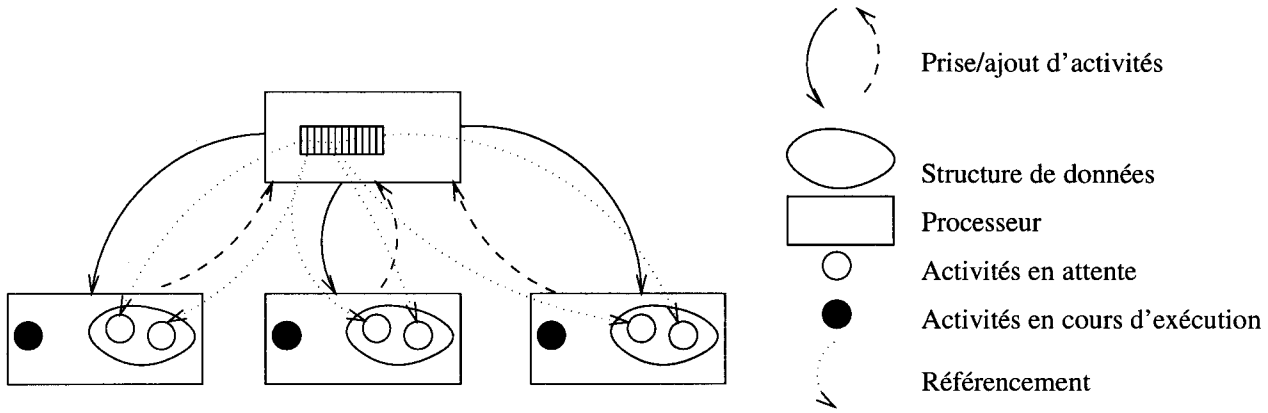


FIG. 2.13 - *Schéma d'implantation d'une structure de données parallèle globale utilisant le principe d'indirection*

sur les données décrivant les activités au lieu de conserver les données elles-mêmes. Cela permet donc d'utiliser l'ensemble de la mémoire des processeurs pour le stockage en augmentant ainsi la taille des problèmes traitables.

C'est là que réside l'avantage principal de cette méthode : elle permet de marier la simplicité du modèle centralisé tout en s'affranchissant de la limite qu'il impose en terme de mémoire utilisée. Elle a cependant quelques inconvénients :

Surcoût de communication Pour obtenir une activité en attente, ou en ajouter une nouvelle dans la structure de données globale deux communications sont nécessaires alors que dans les schémas précédents une seule suffisait. Ce surcoût, qui pourrait être prohibitif sur une architecture faiblement couplée telle qu'un réseau de stations, est, dans le cas des machines parallèles visées qui fournissent un réseau performant, tout à fait raisonnable.

Surcoût mémoire De la même façon que deux communications sont nécessaires pour accéder à une donnée, son stockage occupe deux emplacements en mémoire, un pour la stocker réellement et l'autre pour stocker sa référence. Ce dernier emplacement doit être petit par rapport au premier pour éviter de rendre son coût prohibitif. On peut cependant noter que les problèmes résolus à l'aide de ce type de méthodes sont de grande taille, le stockage d'une activité nécessitant donc une place mémoire importante. Cela rend alors le surcoût mémoire induit par le référencement quantité négligeable.

Degré de parallélisme peu élevé La structure centralisée étant conservée pour les accès aux données, ce problème, détaillé dans la section 2.2.4 se pose toujours. Il risque même d'être exacerbé par les architectures visées. En effet, le stockage distribué ainsi que le faible coût des communications incite à utiliser une faible valeur de granularité. Les processeurs étant puissants cette faible granularité va impliquer une forte activité de communication qui risque donc d'amener à une saturation du maître, particulièrement si le nombre de processeurs est élevé ce qui est souvent le cas, car c'est l'une des forces de ce type de machines. Il est donc possible de perdre en temps d'attente les gains obtenus par la distribution de la structure. À cet égard, il est intéressant de faire le parallèle avec les résultats expérimentaux présentés dans la section 2.2.4.4 page 29 qui ont été obtenus dans des conditions expérimentales semblables, en particulier sur la même architecture parallèle.

Cette méthode est cependant intéressante car elle illustre bien l'évolution des méthodes de stockage que nous avons présentées. La première solution vers laquelle est poussé le concepteur d'une application parallèle hautement irrégulière est la centralisation. Après avoir constaté ses limites, en particulier en terme de capacité de stockage, il est naturel de se tourner vers une solution plus distribuée au prix d'une complexité accrue. Il est ensuite naturel de tenter de marier les deux approches en tenant compte des évolutions techniques récentes, soit en partant de l'approche distribuée pour se rapprocher du centralisé, ce qui donne l'organisation hiérarchique présentée en 2.2.6.1, soit en faisant le chemin en sens inverse dont le résultat est cette approche par indirection.

2.2.7 Conclusion sur l'approche à gros grain

Dans cette section, nous avons présenté plus en détail les spécificités des applications hautement irrégulières, ce qui nous a mené à introduire la notion de granularité qui est l'élément clé de toute parallélisation de ce type d'applications. Cette granularité permet de mettre en évidence la nécessité d'avoir, dans un environnement ne proposant pas d'activités de fine granularité, deux types d'activités : celles qui sont en cours d'exécution et celles dont l'exécution est retardée. Nous avons montré comment cela amène à mettre au cœur de ces applications une structure de données dont le but est de gérer ces activités en attente. Nous avons alors présenté les deux principales méthodes permettant d'implanter cette structure de données dans un contexte parallèle totalement distribué, en illustrant notre discours avec des mesures de performances. Nous avons terminé en présentant deux méthodes combinant les caractéristiques des précédentes.

Les caractéristiques d'une telle structure peuvent être résumées en deux points :

Son contrôle La manière dont les éléments sont gérés. Cette gestion peut dépendre d'une autorité unique et donc être centralisée, ou être répartie sur l'ensemble des processeurs et donc être distribuée.

Son mode de stockage des données La manière dont les éléments sont stockés, c'est-à-dire les endroits où ils seront rangés. Cela peut être dans l'espace mémoire d'un seul processeur, donc être centralisé, ou bien sûr l'ensemble des processeurs, donc être distribué.

Nous récapitulons les caractéristiques des structures que nous avons présentées dans ce chapitre dans le tableau 2.5.

Contrôle \ Stockage	Centralisé	Distribué
Centralisé	Schéma maître/esclave	Stockage par indirection
Distribué	Non utilisé	Schéma distribué présenté

TAB. 2.5 - *Classification des différents schémas de parallélisation d'une structure de données globale distribuée*

Dans ce tableau il est clair que la méthode la mieux adaptée aux architectures parallèles distribuées et faiblement couplées est celle offrant à la fois un stockage et un contrôle distribués. Ce tableau ne met cependant pas en évidence la difficulté de conception que l'utilisation de cette méthode implique.

Il est à noter que la nécessité d'utiliser ces méthodes vient de l'impossibilité d'exécuter des activités dès qu'elles sont générées et ceci à cause de la granularité des activités supportées par la famille de supports d'exécution choisie. Cela nous impose donc de séparer artificiellement les

activités en cours d'exécution des activités en attente avec le risque de continuer à exécuter une activité alors qu'une autre beaucoup plus intéressante, c'est-à-dire pouvant amener un très bon résultat, reste en attente de la libération d'un processeur.

Dans la mesure où les problèmes que nous visons ont la particularité d'avoir une granularité applicative manipulable à volonté, une solution naïve pourrait être de penser « Créons autant d'activités qu'il y a de processeurs puis maintenons cet état en créant une nouvelle activité dès qu'un processeur se libère ». Cette solution, séduisante *a priori*, a pour conséquence de créer des activités qui auront une granularité élevée, typiquement un processeur explorera la même partie de l'arbre de recherche pendant 95% du temps d'exécution de l'application. Si ce sous-arbre mène à des solutions peu intéressantes alors une portion importante de temps processeur aura été perdue alors qu'elle aurait pu être employée à explorer une partie plus intéressante de l'arbre. Un autre inconvénient, plus lié aux contraintes techniques d'implantation, sera le temps perdu en attente d'un nouveau sous-arbre à explorer, c'est-à-dire la création d'une nouvelle activité. Cela peut sembler être un problème simple mais il n'en est rien car il faut interrompre le traitement d'une activité pour lui demander de se scinder en deux, de donner une partie de son espace de recherche à une autre activité, ce qui est difficilement réalisable de façon efficace dans un environnement à gros grain.

2.3 Conclusion

Dans ce chapitre nous avons mis en évidence le rôle crucial joué par la granularité dans la parallélisation des problèmes hautement irréguliers. Nous avons vu que le problème principal qui se pose est celui de la dualité entre les granularités exécutive et applicative qui oblige à une gestion à deux niveaux des activités de l'application parallèle.

Nous avons ensuite montré que cette gestion implique l'utilisation d'une structure de données globale qui est le point principal des parallélisations de cette classe d'applications dans un contexte à gros grain. Nous avons alors présenté les approches principales permettant de paralléliser cette structure de données, l'approche centralisée, l'approche distribuée et enfin deux approches hybrides des précédentes. Ces approches sont utilisées pour marier des granularités applicative et exécutive différentes. Elles obligent à mettre en place une description des activités intermédaires entre l'activité et le support d'exécution.

Nous allons présenter dans le chapitre suivant des environnements proposant une granularité exécutive adaptée aux applications visées.

Chapitre 3

Virtualisation d'architecture et ordonnancement d'applications parallèles hautement irrégulières

Nous avons vu dans le chapitre précédent que les problèmes hautement irréguliers pouvaient être d'une granularité fine et que leur prise en compte par des environnements classiques à gros grain mettaient en œuvre des mécanismes complexes et peu généralisables. Cela est dû à la différence entre les granularités applicative et exécutive. Cependant, il existe des environnements qui proposent des modèles de programmation basés sur une granularité exécutive fine qui semblent donc adaptés aux applications visées. L'étude de leur utilisation dans un cadre distribué fait l'objet de ce chapitre.

Dans un premier temps nous détaillerons pourquoi le parallélisme à grain fin est ici adapté en proposant une approche de parallélisation basée sur cette notion. Nous en détaillerons les avantages en nous plaçant dans un cadre local. Nous étendrons ensuite notre étude à un cadre distribué en présentant les différentes solutions existantes pour marier le parallélisme à grain fin et les architectures faiblement couplées. Cela nous amènera à exposer notre approche qui est basée sur la virtualisation d'architecture. L'idée au centre de la virtualisation est que le programmeur doit s'affranchir au moment de la conception des contraintes techniques posées par l'architecture parallèle sur laquelle son application s'exécutera. Le but de cette approche est de faciliter la programmation mais aussi de fournir, par l'utilisation de mécanismes éléments du support d'exécution, une bonne efficacité.

Cette approche est composée de deux aspects : l'un concerne la méthode de structuration de l'application en terme d'activités, l'autre s'intéresse à la gestion des activités par le support d'exécution pour une bonne efficacité. Nous étudierons ces deux aspects en isolant les concepts clé nécessaires à la virtualisation. Nous verrons qu'au cœur de la gestion des activités on trouve la notion de mobilité qui repose sur un mécanisme de migration. Cette notion étant particulièrement importante dans notre modèle, nous l'étudierons plus en profondeur en regardant aussi son utilisation dans d'autres systèmes basés eux aussi sur le parallélisme à grain fin. Nous verrons en particulier que l'utilisation de la migration implique de calculer une répartition, un placement, des activités de l'application sur les processeurs, c'est-à-dire *d'ordonnancer* les activités.

L'ordonnancement dynamique d'activités de faible granularité dans un environnement distribué est un problème récent et il y a trop peu de travaux pour l'instant dans ce domaine. Nous étudierons les principaux en isolant leurs caractéristiques marquantes. Nous mettrons cela en perspective avec les contraintes posées par un haut niveau d'irrégularité et regarderons l'adéquation des systèmes étudiés avec ces contraintes. Nous conclurons sur l'adéquation des approches proposées à la

parallélisation des applications qui nous intéressent.

3.1 Environnements à granularité fine et applications hautement irrégulières

Nous avons présenté dans le chapitre précédent plusieurs schémas de parallélisation d'applications hautement irrégulières reposant sur des environnements à gros grain. Nous allons maintenant étudier une approche radicalement différente car utilisant un grain de parallélisme beaucoup plus fin.

Nous qualifions cette approche de grain fin car elle repose sur l'utilisation de supports d'exécution proposant des processus légers, c'est-à-dire des activités d'un grain plus fin que les processus UNIX des approches précédentes. Nous commençons cette partie par une présentation des particularités de ces environnements puis comment elles permettent de paralléliser des applications hautement irrégulières. Nous détaillerons ensuite les caractéristiques d'une telle implantation.

3.1.1 Le principe des supports à grain fin

Un processus lourd[1] est composé d'un ensemble de données, telles que la table des fichiers ouverts, la table de gestion des signaux, ... et d'*un* flux d'exécution, c'est-à-dire d'un compteur ordinal et d'une pile d'exécution. Par contre, un processus léger n'est qu'un flux d'exécution.Un processus léger ne peut exister que dans un processus lourd dans lequel il est encapsulé.

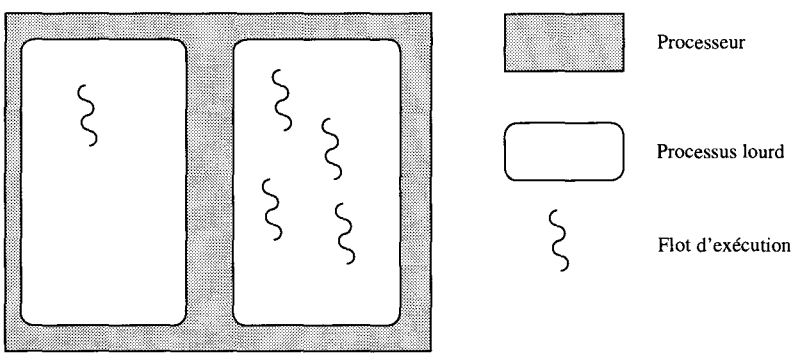


FIG. 3.1 - *Processus légers encapsulés dans un processus lourd*

Dans la figure 3.1, deux processus lourds sont représentés : celui de gauche n'est pas multithreadé (on dit aussi multiprogrammé), c'est-à-dire qu'il ne contient qu'un seul flot d'exécution, qui est le programme principal dans la terminologie traditionnelle. Celui de droite, par contre, contient plusieurs flots, l'un d'eux étant une fonction du programme principal alors que les autres sont des processus légers[2]. Ce processus lourd est dit *multiprogrammé* car il exécute plusieurs activités de manière concurrente[3].

1. Typiquement un processus UNIX mais d'autres systèmes, comme Windows-NT, proposent des mécanismes similaires.

2. Notons que le programme principal n'est rien de plus qu'un flot d'exécution donc est lui aussi un processus léger. La distinction artificielle faite ici n'a qu'un but didactique.

3. Réellement concurrente si la machine hôte de ce processus dispose de plusieurs processeurs et si le système offre cette fonctionnalité, concurrente en temps partagé dans le cas contraire.

L'intérêt principal des processus légers réside dans le fait que leur gestion est beaucoup moins coûteuse que celle des processus lourds, d'un facteur allant de 10 à 100. Cela est dû à la faible quantité d'informations de gestion qui les caractérisent. Une autre raison, assez technique, est que l'ensemble de ces informations réside dans l'espace mémoire du processus, donc dans l'espace utilisateur, et qu'y accéder ne nécessite pas d'invoquer de coûteux appels au noyau. Le prix de cette efficacité se paie en terme de sécurité, car se passer du noyau implique se passer aussi des mécanismes de protection mémoire qu'il fournit[4].

Dans le cadre des applications que nous visons, disposer d'activités ayant un faible coût de gestion, donc qualifiées de « légère », permet d'envisager d'en créer un grand nombre.

Pour illustrer cela, prenons une application appartenant à la classe que nous avons principalement étudiée dans le chapitre précédent: celle des problèmes hautement irréguliers. Dans les approches à gros grain que nous avons présentées, la différence entre les granularités applicative et exécutive rendait nécessaire la distinction entre deux types d'activités: celles en cours d'exécution et celles attendant la disponibilité d'un processeur pour s'exécuter. Le haut degré de parallélisme que permet l'utilisation des processus légers va permettre de supprimer cette dernière catégorie.

Avec un support à grain fin, il n'existe plus d'activités en attente d'exécution car toutes les activités sont exécutées simultanément, chacune étant prise en charge par un processus léger. Cela est illustré sur la figure 3.2.

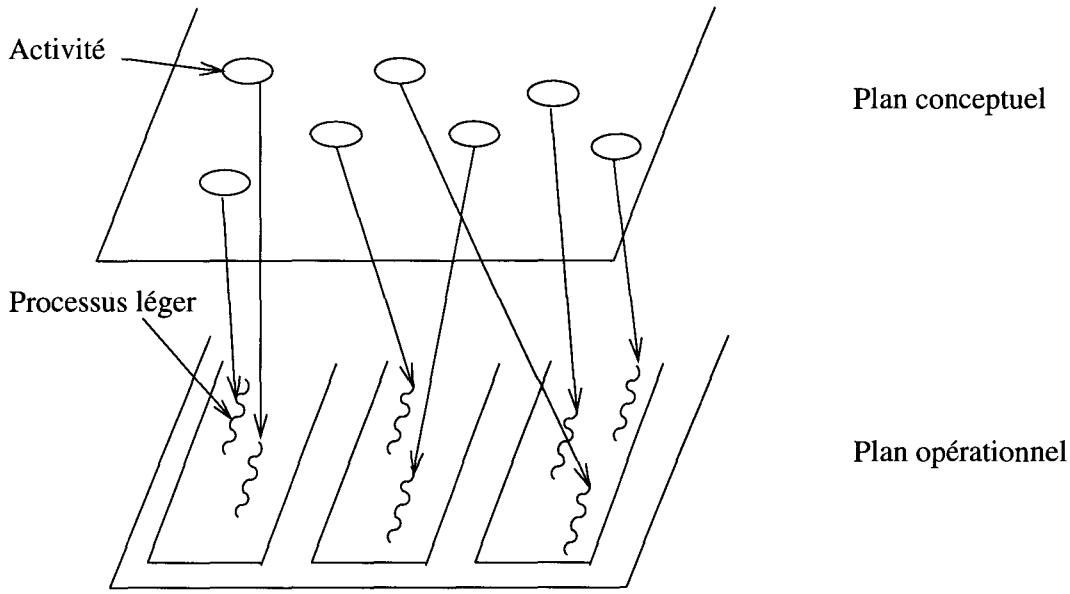


FIG. 3.2 - *Schéma d'implantation d'une application hautement irrégulière sur un support d'exécution proposant des activités de granularité fine*

Cette figure ne contient que l'idée de base de cette proposition. Les détails, notamment ceux liés à la distribution des activités sur les processeurs, seront explorés dans la suite du document.

De manière concrète, dans les exemples donnés en section 1.4.1.3 page 16, ce changement de granularité exécutive se concrétise de la manière suivante:

Branch&Bound Au lieu d'utiliser une structure de données pour stocker les sous-arbres en attente

4. Des bibliothèques de processus légers comme MARCEL[Nam97b] offrent cependant des mécanismes rudimentaires de protection.

d'exploration, cette exploration démarre effectivement dès que l'activité qui en a la charge est créée.

Simulation de particules D'un modèle basé sur le stockage des particules dans un tableau local à chaque processeur avec des communications entre les processeurs pour transmettre les interactions entre particules, on passe à un modèle où chaque particule, ou groupe de particules suivant la granularité applicative choisie, est prise en charge par des activités communicantes, de manière directe ou indirecte.

3.1.2 Avantages

Le premier de ces avantages, la **flexibilité**, réside dans le choix complet dont dispose le programmeur pour choisir la granularité des activités composant son application. Dans les approches précédentes, ce choix était soit imposé par le support d'exécution, soit il obligeait à le contourner par des mécanismes compliqués. Ici, le programmeur choisit la granularité de ces activités comme il l'entend. Qu'elle soit fine, moyenne ou grosse, le support d'exécution la supportera, on le verra, efficacement[5]. Une conséquence directe est que le degré de parallélisme à l'intérieur d'une application est virtuellement infini, il ne dépend plus ni du support d'exécution ni de l'architecture sous-jacente mais au contraire du souhait du concepteur de l'application ; nous verrons dans la suite que cette propriété est très importante pour certains problèmes.

En corollaire, cela implique la suppression de la dualité entre la granularité définie au moment de la conception de l'application par le programmeur et celle imposée, de manière directe ou indirecte, par le support d'exécution utilisé. Les granularités applicative et exécutive sont identiques.

Un avantage supplémentaire, la **transparence**, réside dans le fait que les activités applicatives sont des entités qui sont maintenant visibles directement par le support d'exécution, ce qui n'était pas le cas auparavant. Cela permet d'envisager la conception de mécanismes inclus dans le support d'exécution, ou travaillant de concert avec lui sous la forme d'une sur-couche indépendante, prenant en charge de manière transparente la gestion ou le placement par exemple, des activités applicatives.

L'idée principale que nous défendons est celle d'une gestion des activités, transparente pour l'application, dont les avantages sont :

- **Facilité de développement** Le programmeur de l'application n'a plus à prendre en compte les problèmes techniques liés à la parallélisation, à part celui de la décomposition de son application en activités. Notons que la granularité de ces activités peut être quelconque, c'est-à-dire qu'elle est conditionnée par les caractéristiques de l'application et par elles seules. Nous étudierons dans la suite les paradigmes proposés dans ce but.

- **Transparence** Aussi bien au moment de la conception qu'à celui de l'exécution, le fonctionnement du support d'exécution est complètement transparent pour l'application.

- **Efficacité** Le support d'exécution peut, en utilisant les caractéristiques des activités applicatives dont il dispose, les gérer, et en particulier décider de leur placement, afin d'assurer une bonne efficacité. Cela n'est possible que grâce au fait que le support est conscient de l'existence des activités applicatives.

Pour atteindre ces buts il est essentiel d'avoir un élément commun entre le support d'exécution d'une part et l'application d'autre part. En fournissant un type d'activités dont la granularité peut être quelconque (dans le sens qu'elle sera gérée efficacement quelle que soit sa classe), on

5. Jusqu'à un certain degré de finesse, cependant.

donne au programmeur la possibilité de créer des activités représentant une granularité applicative quelconque sans pour autant que ce choix de granularité pénalise les performances.

3.1.3 Conclusion

Dans cette section, nous avons présenté le principe que nous proposons pour la parallélisation des applications hautement irrégulières utilisant un support d'exécution à grain fin. Au cœur de notre proposition on trouve la volonté d'affranchir au maximum le concepteur d'une telle application des contraintes posées par la parallélisation pour le laisser se concentrer sur les aspects algorithmiques. Nous proposons pour cela l'utilisation d'un environnement basé sur les processus légers qui permettent de manipuler, de manière transparente pour l'application, des activités de granularité quelconque sans sacrifier les performances.

Dans le cas des applications que nous avons étudiées précédemment, l'utilisation des processus légers permet une adéquation entre la granularité applicative, à savoir celle des activités formant l'application parallèle, et la granularité exécutive celle du support d'exécution.

Le problème de l'utilisation des processus légers dans un cadre distribué se pose maintenant. La présentation technique que nous en avons fait dans cette section ne concerne que le cadre local. Au contraire, dans la figure 3.2, nous les utilisons dans un cadre distribué. Le passage de l'utilisation des processus légers d'un cadre purement local vers celui que nous visons, qui nécessite l'usage de primitives de communication, fait l'objet de la suite de ce chapitre.

3.2 Processus légers et distribution

Le défi posé aux concepteurs de support d'exécution distribué consiste à intégrer les processus légers dans un modèle de programmation qui répond aux critères suivants :

Cohérent Les processus légers sont un mécanisme de conception permettant de s'affranchir de la lourdeur de la gestion des activités dans les environnements traditionnels. À ce titre, ils constituent un élément supplémentaire à la disposition du programmeur et posent de nouvelles questions sur la façon de structurer une application. Il est donc nécessaire, à la fois pour des raisons de structuration, de facilité de programmation et de réutilisation de codes existants, de les *intégrer* dans un nouveau modèle de programmation qui permettra d'exploiter leurs spécificités.

Efficace Le but premier de l'utilisation des processus légers étant d'améliorer l'efficacité des applications, le modèle doit être conçu en préservant cet objectif.

La question la plus importante dans cette optique est de savoir comment intégrer les processus légers, qui sont une notion typiquement liée à un environnement local, avec des capacités de distribution. Deux approches principales se dégagent :

Juxtaposition Le modèle de programmation est conçu en juxtaposant des capacités de multiprogrammation légère avec une librairie de communication par messages comme PVM ou MPI. Chaque activité de l'application peut directement communiquer avec toute autre activité.

Intégration Les processus légers se trouvent au centre du modèle de programmation. Le reste du modèle est conçu autour et en fonction d'eux.

La première approche, séduisante par sa simplicité de mise en œuvre, montre vite ses limites tant en terme sémantique (comment utiliser des primitives de communication conçues pour les processus lourds avec des processus légers?) que pratique (l'intégration de composants traditionnels avec des processus légers pose de nombreux problèmes techniques), nous allons donc nous concentrer sur la deuxième.

On peut séparer les propositions faites pour intégrer processus légers et distribution en deux familles qui se distinguent par le degré d'interaction entre processus légers :

Communication directe Les processus légers peuvent communiquer entre eux de la même manière que les processus lourds dans les environnements traditionnels. Le modèle proposé est cependant différent de la juxtaposition car il prend en compte les propriétés paticulières des processus légers comme nous le verrons dans les environnements étudiés.

Invocation distante de services Les processus légers prennent en charge des appels de services en provenance d'autres processus légers. On peut rapprocher cette idée de conception du célèbre modèle client/serveur, un appel de procédure correspondant à une requête de service qui sera rendue par l'exécution d'une procédure sur un des processeurs.

Nous allons détailler ces deux approches en illustrant notre discours par la description d'environnements existants. Nous allons essayer en particulier de mettre en évidence les éléments structurants propres à chaque modèle. Nous étudierons aussi l'adaptation de chacun aux applications que nous visons.

3.2.1 Communication directe

Les environnements de développement d'applications parallèles à gros grain étant, dans leur immense majorité, basés sur l'envoi de message ce paradigme est bien connu. La communication directe entre processus légers en est la transcription directe. Au lieu d'avoir des échanges de message entre processus lourds, ces échanges ont lieu entre processus légers.

Le choix de ce mécanisme a les conséquences suivantes :

Réutilisation facilitée d'applications Le paradigme d'envoi de messages est le plus répandu dans la communauté du parallélisme actuellement. Le nombre d'applications existantes programmées avec lui est donc élevé, *étendre* son utilisation au cadre des processus légers permet d'adapter ces applications à ce nouvel environnement relativement facilement au lieu de nécessiter une refonte totale.

Nécessité d'un nommage global La communication se faisant de processus léger à processus léger il est nécessaire de disposer d'un nommage global pour eux. Cela implique aussi des échanges d'identifiants entre les activités pour déterminer à qui elles doivent s'adresser. Quand ces activités ont une courte durée de vie courte, il existe un risque d'envoyer un message vers une qui s'est déjà terminée. Pour éviter cela il est parfois nécessaire de mettre en place de, coûteux, mécanismes de synchronisation.

Structuration de l'application Le modèle de programmation à base d'échange de messages est apdapté pour certaines catégories d'applications, telles que les applications simulant le monde réel. Cette inadaptation est encore plus mise en évidence quand ce paradigme est utilisé avec les processus légers où la nécessité de disposer d'une adresse double (celle du processus lourd encapsulant le processur léger destinataire) amène une certaine lourdeur dans la programmation.

Différents supports d'exécution proposent cette approche, nous allons en détailler deux : TPVM et CHANT. Nous allons commencer par présenter les raisons ayant présidé à la création de ces deux environnements puis nous décrirons succinctement leurs fonctionnalités. Enfin, nous discuterons sur la façon de programmer les applications hautement irrégulières présentées auparavant avec ces environnements proposant l'envoi de message comme brique de base de la distribution. Nous n'abordons volontairement pas l'implantation de ces environnements sauf quand leur influence sur les performances est si forte qu'elle conditionne le choix du modèle de programmation. Le lecteur intéressé par ces aspects consultera avec profit [Nam97b, Gin97].

3.2.1.1 TPVM

TPVM a été conçu par l'équipe qui a également travaillé sur PVM [GBD+95]. L'objectif était de pallier à l'inefficacité du système PVM concernant deux points : une gestion des tâches trop coûteuse et un mauvais recouvrement calcul/communication. Initialement, l'idée était de remplacer l'utilisation des processus lourds par des processus légers tout en préservant la compatibilité avec PVM.

L'utilisation du mécanisme d'envoi de message se fait de façon identique à PVM, la création de nouvelles tâches, qui sont des processus légers, aussi. Notons que le site de création d'une nouvelle tâche peut être choisi soit par l'application soit par le support d'exécution en fonction de la charge du système, à l'instant de la création. Un mécanisme similaire existe d'ailleurs dans PVM pour la création des processus lourds.

Il est apparu aux concepteurs de TPVM que l'introduction des processus légers menait à l'écriture d'applications comportant un grand nombre, typiquement plusieurs milliers, de tâches fonctionnant simultanément et que le modèle de programmation de PVM, ne comportant que l'envoi de messages, n'était plus adapté, en particulier pour les aspects synchronisation et régulation de charge. C'est pourquoi ils ont introduit un mécanisme d'exécution dirigée par les données qui permet de conditionner l'activation, et donc la création et en particulier le choix du site de création, d'une activité à la disponibilité des données dont elle a besoin. Cette idée, semblable au mécanisme bien connu du data-flow qui est au cœur de l'implantation parallèle de langages fonctionnels, est aussi au centre des environnements CILK et ATHAPASCAN-1 que nous détaillons dans la section 3.5.2 page 72.

Notons que les deux modèles proposés, le data-flow et la communication par passage de messages, sont très éloignés l'un de l'autre et qu'il est difficile de les marier dans une application. Au contraire, une autre fonctionnalité proposée par TPVM, le mécanisme de mémoire partagée, peut être utilisée en conjonction avec les deux précédents sans problème. La cohérence proposée est forte et implique donc un coût en communication assez élevé. Il semble donc difficile de s'en servir comme outil unique de distribution du fait de ses faibles performances sur une architecture distribuée, elle sera donc utilisée en complément d'un des deux autres mécanismes.

3.2.1.2 Chant

Le but poursuivi par CHANT est différent de celui de TPVM car ses concepteurs visent à fournir un support de processus légers communicants au-dessus duquel seraient ajoutés des services plus évolués pour les programmeurs d'application. Les deux objectifs visés par ses concepteurs sont la portabilité, pour pouvoir être implanté facilement sur un grand nombre d'architectures, et l'efficacité pour des raisons évidentes.

Plus précisément un des domaines applicatifs visé en priorité par les concepteurs est l'exécution du code issu de compilateurs de langages data-parallèles [HHM+95].

Le mécanisme principal de distribution sur lequel est basé tout l'environnement CHANT est l'envoi de messages. Les autres facilités offertes au programmeur sont toutes construites au dessus de ce mécanisme dont les performances ont été particulièrement optimisées. Pratiquement, la communication directe peut se faire entre deux processus légers s'ils sont *chanteurs*, c'est-à-dire s'ils ont été créés comme tels auprès du support d'exécution. La plupart des opérations disponibles sur les processus légers, comme la création par exemple, ont leur correspondance pour les processus légers *chanteurs*.

Cependant, l'environnement CHANT étant particulièrement destiné au support de langages data-parallèles, il fournit des primitives permettant de regrouper plusieurs activités en une entité manipulable directement offrant donc la possibilité d'opérations collectives sur un ensemble de processus légers. De telles entités sont nommées cordes[6] dans la terminologie CHANT car elles sont formées de plusieurs fils (threads) d'exécution. Les opérations offertes sur les cordes concernent la communication, envoi d'un message à tous les membres par exemple, et la synchronisation avec un mécanisme de barrière.

De plus, les concepteurs de CHANT sont conscients de la lourdeur du mécanisme d'envoi de message pour certaines opérations comme la mise à jour d'une variable partagée par tous les modules par exemple, car alors le destinataire du message de modification doit être dédié à cette tâche uniquement ce qui est peu intuitif. Nous revenons plus en détail sur ce point dans la section suivante. Ils ont donc ajouté à leur plate-forme un mécanisme d'invocation distante de services pour rendre son utilisation plus souple. Le passage de message étant cependant au cœur du système CHANT, il nous semble naturel de le classer dans cette catégorie et non dans celle étudiée par la suite.

Pour être exhaustif, signalons que l'équipe de conception de CHANT a évoqué dans [CHM97] son intention d'intégrer un mécanisme de migration dans sa plate-forme, essentiellement pour des besoins de régulation de charge.

3.2.1.3 Discussion sur les environnements à passage de message

Dans cette section, nous allons analyser les deux environnements présentés sous l'angle du programmeur d'applications hautement irrégulières.

Pour cela nous alons analyser la construction d'une application telle que celle déjà présentée avec le mécanisme d'envoi de messages. Rappelons pour mémoire les grandes lignes de cette application : elle est constituée d'un potentiellement grand nombre d'activités n'ayant pas d'autre relation que celle de parenté, c'est-à-dire que chaque activité a été créée par une autre. De plus, nous allons supposer qu'il existe une (ou des) valeur partagée entre toutes les activités dont la valeur change peu souvent mais qui est, par contre, lue souvent. Comment implanter cela avec le modèle proposé?

La création d'une activité se fera par la création d'un thread. Des paramètres peuvent être transmis par l'envoi d'un message à l'activité nouvellement créée. Cette activité effectuera ensuite son travail en en créant éventuellement d'autres puis se terminera. Aucune communication explicite n'étant nécessaire dans l'application les seules qu'elle effectuera seront pour transmettre des paramètres aux activités qu'elle aura créées. L'absence de dépendances réelles entre les activités rend peu utile l'utilisation du mécanisme de dépendances sur les données proposées par TPVM.

La gestion de la valeur partagée est plus délicate. N'oublions pas que les processus légers sont encapsulés dans des processus lourds, une valeur partagée par toutes les activités doit donc être visible de tous les processus lourds. Il s'agit donc d'une variable située dans une mémoire partagée. TPVM nous proposant un mécanisme de mémoire partagée, il semble évident de l'utiliser. Ce n'est cependant pas une bonne idée pour des raisons d'efficacité. En effet, au cœur du mécanisme proposé

6. *Rope* en anglais

se trouve la notion de zones mémoire exportées : une zone mémoire appartenant à un processus lourd peut être utilisée par tous les autres qui y accéderont grâce à des opérations de lecture et d'écriture distante. Une écriture distante se fera par un envoi de message vers le processus propriétaire de la zone alors qu'une lecture en utilisera deux, un pour la demande de lecture et un autre pour le renvoi du résultat. Si la valeur doit être lue souvent, disons toutes les millisecondes ce qui est peu pour ce type d'applications, il est clair qu'une implantation aussi inefficace mènera à de mauvaises performances. La question est alors de savoir comment gérer cette valeur.

Il est clair que doit résider en permanence dans chaque processus lourd une copie de la valeur, cette copie est partagée par les processus légers résidant sur ce processus et est mise à jour globalement quand une nouvelle valeur y est affectée. Cette opération de mise à jour doit être effectuée par un envoi de message puisque c'est le moyen de communication principal des environnements étudiés. Chaque message devant être adressé à un processus léger précis, deux solutions sont possibles :

- utiliser un des processus légers de calcul en lui faisant scruter périodiquement les messages qui lui sont envoyés. Quand il recevra un message de mise à jour ce processus modifiera la valeur à partir du contenu du message,

- laisser en permanence un processus léger sur chaque processus qui ne fera qu'attendre un message contenant la nouvelle valeur. Ce processus s'apparente pour l'application aux démons communément utilisés pour le transfert de données dans Internet.

La première solution a l'inconvénient de mélanger deux types de travaux, calcul et implantation de communication, dans la même activité et en plus risque d'être inefficace car il oblige le calcul à s'interrompre périodiquement pour scruter la présence de messages en attente et leurs caractéristiques. Cette solution est donc à proscrire. La deuxième est meilleure car elle évite les deux inconvénients mentionnés ci-dessus mais elle oblige à disposer en permanence d'une activité sur le processus. En dehors du coût que cela engendre, qui semble dérisoire par rapport au million d'activités que l'application peut générer, se pose le problème de sa dénomination. Elle implique en effet une phase d'initialisation, spécifique à l'application et qui ne peut donc pas être faite par le support, durant laquelle tous les processeurs vont se communiquer les identifiants de chaque processus léger chargé de gérer la valeur. Elle implique aussi de gérer une donnée supplémentaire, l'identifiant, pour chaque processeur alors que l'adresse de ce dernier (plus précisément du processus lourd s'exécutant sur ce processeur) devrait être suffisant pour l'identifier. Cela complique la programmation de l'application en introduisant une phase d'initialisation et une double dénomination qui ne sont justifiées que par des contraintes techniques.

Dans la section suivante, nous allons étudier deux autres environnements proposant l'intégration des processus légers et de la distribution. L'approche qu'ils ont choisie est différente car elle est basée sur l'invocation distante de services.

3.2.2 Invocation distante de services

Le modèle client/serveur est un des modèles les plus utilisés en informatique distribuée par sa simplicité et sa robustesse. Pour preuve, tous les services fournis par le réseau Internet, comme le World Wide Web ou les nouvelles[7], sont bâtis sur ce modèle. Le principe d'appel de procédure à distance, *Remote Procedure Call* ou RPC en anglais[BN83], est une évolution de ce modèle visant à simplifier son utilisation en éliminant la désignation explicite du serveur et la gestion des messages

7. *News* en anglais

du côté client. Il consiste à rendre transparente pour le client l'exécution d'une procédure sur un site différent du sien. Tout se passe pour ce dernier comme s'il appelait une procédure locale en utilisant simplement une syntaxe spéciale.

Dans un contexte de parallélisme à grain fin, cette invocation créera sur le site distant, le serveur dans la terminologie client/serveur, un processus léger qui se chargera de l'exécution du service demandé. Ce fonctionnement est schématisé sur la figure 3.3 que nous allons maintenant détailler.

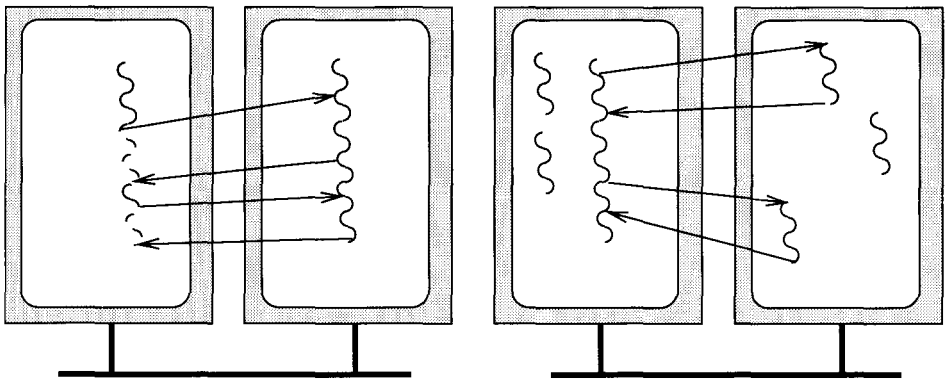


FIG. 3.3 - *Intégration des processus légers et de la distribution par l'invocation de procédure distantes. Sur la gauche RPC dans le cas sans processus légers donc avec un seul flot d'exécution. Sur la droite avec utilisation de processus légers.*

Le fonctionnement sera le suivant : un processus léger demande l'exécution d'un service avec certains paramètres sur un autre processus, les références du service (son identifiant) et les paramètres sont transmis au processus concerné qui crée un processus léger pour exécuter le service demandé. À la fin de l'exécution du service, les résultats sont renvoyés au processus ayant initialement demandé le service. Certains systèmes proposent aussi une invocation de service sans retour de résultat, permettant, par exemple, le lancement distant de processus démons. Nous verrons l'utilité d'un tel mécanisme dans la section 4.5.

On voit donc que l'on est très proche d'un fonctionnement en mode client/serveur. Cependant, sur la figure 3.3, on saisit immédiatement un des avantages des processus légers : ils permettent à un serveur de répondre à différentes requêtes simultanément alors, que dans le cas sans processus légers, le traitement de ces requêtes doit être sérialisé[8]. C'est, historiquement, une des raisons pour lesquelles les processus légers ont été introduits dans les systèmes d'exploitation.

Les objectifs de ce type de mécanisme sont cependant différents des nôtres et ne doivent pas être confondus. Il s'agit d'un côté d'ajouter un mécanisme de bas niveau pour augmenter l'efficacité d'un type précis de serveurs alors que, de l'autre côté, il s'agit d'introduire un mécanisme proche du RPC dans une application parallèle comme élément *structurant.*

Les caractéristiques de cette approche sont les suivantes :

Structuration de l'application Comme mentionné précédemment, le mécanisme utilisé pour introduire la distribution est structurant. Alors que l'introduction du passage de message dans une application séquentielle est rarement naturelle et nécessite souvent une réécriture

8. En fait la plupart des serveurs s'affranchissent de cette limitation en créant un processus lourd pour chaque requête en utilisant l'héritage de contexte du `fork` UNIX mais cela sort du cadre de notre discours.

en profondeur. La structuration en procédures est naturelle et nécessite peu, voire aucune, transcription.

Absence de nommage explicite Le seul paramètre supplémentaire par rapport à un appel procédural traditionnel est le processeur sur lequel le service sera exécuté, contrairement à l'envoi traditionnel du message qui nécessite l'adresse du *processus léger* destinataire.

Indépendance de la localisation La plupart des services ne nécessitent pas l'utilisation de ressources[9] localisées sur un processeur donné. Ces services peuvent donc être exécutés sur n'importe quel processeur et peuvent être éventuellement déplacés durant leur exécution. Cela est difficile à réaliser avec des processus communicants et oblige à utiliser des mécanismes coûteux comme les proxys.

Différents systèmes proposent cette approche, ils sont trop nombreux pour espérer en faire une étude exhaustive dans ce document. Nous allons nous focaliser sur l'étude de deux d'entre eux: NEXUS et ATHAPASCAN-0{a,b}.

3.2.2.1 Nexus

Le système NEXUS[FKOT94, FKT94] est issu des travaux de l'équipe de Ian Foster qui est partie de la constation que la bonne adéquation des processus légers avec les besoins du calcul haute performance rendait leur utilisation intéressante comme support d'exécution pour les compilateurs de langages data-parallèles. NEXUS sert maintenant de support d'exécution pour le projet américain Globus[10] de calcul haute performance.

Les trois mécanismes de base de NEXUS sont:

Les processus légers Les opérations disponibles sur les processus légers dans NEXUS sont un sous-ensemble de la norme POSIX [IEE94] ce qui permet une réutilisation directe des bibliothèques conformes à cette norme disponibles sur de nombreuses architectures.

Les pointeurs globaux permettent un accès direct à une adresse mémoire située sur un processeur distant. Ce mécanisme est une extension à la distribution des pointeurs bien connus de tout programmeur.

Les demandes de service distant qui constituent le mécanisme décrit ci-dessus avec une différence importante qui est qu'il n'y a pas de retour de résultat. Ce mécanisme est appelé Remote Service Request (RSR) dans la terminologie NEXUS.

L'asynchronisme total des RSR s'explique par la présence des pointeurs globaux. Ce sont en effet ces derniers qui sont utilisés pour un stockage du résultat éventuel, ce qui permet d'avoir un site de retour éventuellement différent du site initiateur du service. Cette flexibilité a bien entendu un coût. Notons que NEXUS ne propose aucun mécanisme pour échanger des messages entre processus légers, c'est un environnement entièrement tourné vers l'invocation distante de services.

Un des soucis majeurs de l'équipe NEXUS a toujours été la portabilité de l'environnement, conditition *sine qua non* d'une grande distribution. Ils veulent pouvoir l'adapter à une architecture très simplement et ont donc fait des choix dans ce sens dès la conception. Leur souci majeur a été la réutilisation aisée des fonctionnalités déjà présentes sur les architectures parallèles en terme de processus légers, d'où le choix d'un sous-ensemble de la norme Posix, et en terme de communication, par la prise en compte des environnements les plus utilisés (PVM, MPI, ...).

9. Qui peuvent être des données, bien entendu.
10. `http://www.globus.org`

3.2.2.2 Athapascan-0b

Le cas d'ATHAPASCAN-0b est un peu particulier car il est le successeur du support d'exécution du projet APACHE à ses débuts : ATHAPASCAN-0a[Chr96]. Son but est de fournir à la fois un cadre conceptuel et un environnement d'exécution pour les applications irrégulières, quel que soit leur degré d'irrégularité. Il fournit comme mécanisme de base de décomposition l'appel procédural distant, c'est pour cette raison que nous avons classé ATHAPASCAN-0b dans cette catégorie même s'il s'en éloigne maintenant et cela pour des raisons liées à la restructuration du projet APACHE. En effet, ce dernier avait pour base et comme *élément structurant* les deux notions d'appel de service distant et de poly-algorithmes. Ces derniers consistent en différentes descriptions algorithmiques de la même application, l'environnement choisissant l'expression la plus adaptée aux conditions d'exécution. On retouve dans ce concept l'idée, pour le concepteur d'applications parallèles, de s'affranchir du système sous-jacent. Cette idée est également au cœur du projet ESPACE, bien qu'elle s'y situe à un niveau d'abstraction différent.

Récemment, l'équipe du projet APACHE a décidé de faire remonter les concepts structurants à un niveau supérieur d'abstraction, ATHAPASCAN-1, que nous allons détailler dans la suite, est un des modèles proposés dans ce but. Dans ce nouveau cadre, ATHAPASCAN-0b devient un support d'exécution qui fournit une interface standard et facilement portable pour le développement d'ATHAPASCAN-1. Les aspects structurants n'en sont donc plus l'élément clé, au bénéfice des fonctionnalités et de la modularité. Le point clé sur lequel se sont concentrés les efforts de l'équipe d'ATHAPASCAN-0b concerne la réutilisation d'éléments standards, donc largement diffusés, pour assurer un portage facile vers de nouvelles architectures et cela sans nuire aux performances de l'ensemble. Il est cependant à noter que des programmeurs utilisent directement ATHAPASCAN-0b pour concevoir des applications parallèles.

Dans cette section, nous allons présenter ATHAPASCAN-0b en suivant le même schéma que pour les autres environnements présentés précédemment. Cet environnement présente quelques particularités très intéressantes comme nous allons le voir.

On peut dégager deux fonctionnalités clés dans ATHAPASCAN-0b :

Exécution distante de services Cette fonctionnalité ayant déjà été présentée précédemment, nous n'allons pas la détailler plus longuement.

Communication directe par ports Nous avons vu dans la section précédente que certains systèmes proposent la communication directe entre processus légers et pourquoi ce mécanisme n'était pas très adapté à des entités aussi fines que les processus de grain fin. C'est en partant de ce constat que les concepteurs d'ATHAPASCAN-0b proposent un mécanisme légèrement différent et qui s'inspire de la conception des serveurs sur le réseau Internet. Chaque processus léger peut se mettre en attente sur un port de communication, chaque port est propre à un processus lourd et les messages doivent toujours être envoyés vers un port. Les processus légers peuvent filtrer les messages qui les intéressent par le mécanisme des « étiquettes»[11] bien connu des utilisateurs de librairie de passage de messages. Ce mécanismes des ports, illustré sur la figure 3.4, permet de supprimer l'un des inconvénients de l'envoi de messages vers les processus légers car il est fort probable qu'un au moins sera toujours à l'écoute du port destinataire. L'étape suivante peut être la création automatique d'un processus léger pour répondre à l'arrivée d'un message sur un port que personne n'écoute mais les concepteurs d'ATHAPASCAN-0b n'ont pas implanté un tel mécanisme.

11. Appelé tags en anglais

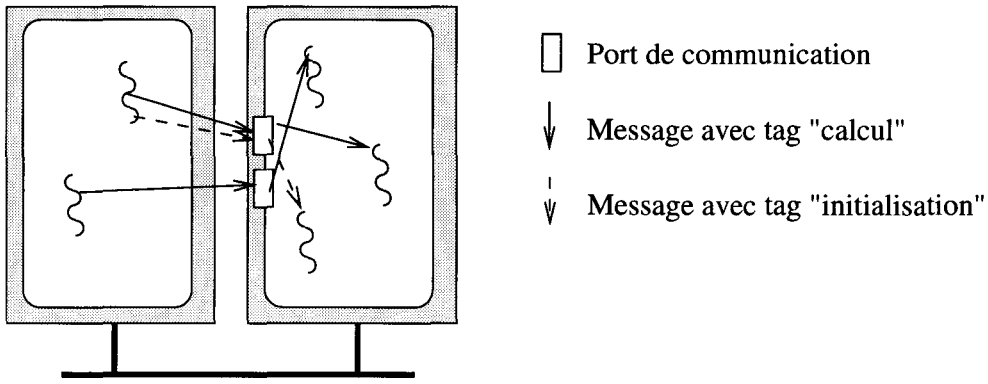


FIG. 3.4 - *Illustration de l'utilisation du mécanisme des ports et des étiquettes dans l'environnement* ATHAPASCAN-0*b*

La création d'activités en ATHAPASCAN-0b se fait par l'invocation distante de services, les processus légers pouvant par la suite communiquer par le biais de l'envoi de messages. Cela permet d'envisager trois schémas de conception d'une application :

Communication par envoi de messages Une fois que les activités ont été créées par l'appel de services, elles communiquent directement, typiquement par l'affectation d'un port, ou d'un tag et d'un port, par activités.

Appel distant de services Ce schéma consiste à utiliser uniquement la fonctionnalité d'appel de services. Nous le décrivons en détail dans la suite.

Regroupement d'activités en pool L'application est décrite comme un pool d'activités, créées par invocation distante, qui se mettent en attente de travail, typiquement de paramètres car le service a déjà été spécifié dans le type d'activités créées, sur un port. L'envoi d'un message à une seule des activités en attente répartit les travaux entre elles. Ce schéma est une extension aux processus légers du schéma maître/esclave auquel il ajoute la possibilité d'utiliser un grain plus fin de parallélisme.

3.2.2.3 Discussion sur les environnements à invocation distante de services

Dans cette section nous allons étudier l'adéquation des environnements proposant les RPC comme mécanisme principal de distribution à la parallélisation des applications hautement irrégulières. La caractéristique principale de ces applications, comme nous l'avons vu dans le chapitre 2, est de contenir un potentiellement grand nombre d'activités indépendantes et qui peuvent être gérées séparément, la seule relation existant entre elles étant celle de parenté. La fonctionnalité principale dont nous avons besoin est donc celle de création d'activités qui doit être particulièrement peu coûteuse car les activités étant nombreuses, leur gestion doit être efficace. Dans les environnements basés sur l'invocation distante de services, cette fonctionnalité a été particulièrement optimisée pour être la plus efficace possible puisque c'est elle qui est au centre du support. Ce point est crucial pour les applications visées.

Indépendamment des considérations de performance, sur lesquelles nous revenons dans la partie conclusion de cette section, l'utilisation exclusive de l'invocation distante de services induit une structuration implicite de l'application. Or cette structuration est justement bien adaptée aux

applications visées. La méthodologie de conception imposée[12] est donc en adéquation avec les caractéristiques algorithmiques de l'application. Cette propriété est importante car la majeure partie du travail d'un concepteur de programmes parallèles consiste traditionnellement à prendre un algorithme performant dans le contexte séquentiel et à l'adapter à un environnement parallèle. En fournissant un mécanisme de *décomposition* de l'application en activités parallèles aussi répandues que l'est l'appel de procédure, ces environnements facilitent la conception de nouvelles applications ou le passage d'anciennes vers un environnement distribué.

Il reste cependant le problème déjà évoqué dans la section 3.2.1.3 page 52 des variables globales. Nous avons mentionné précédemment que leur prise en compte dans les environnements à passage de message nécessitait la mise en place d'une activité permanente dédiée à leur mise à jour. Avec l'invocation distante de services, le cas est différent car toute communication entre deux processus lourds, encapsulant les processus légers, se fait par la création d'une activité nouvelle sur le processus récepteur. La mise à jour de la valeur de cette façon est donc tout à fait naturelle car elle consiste tout simplement en l'appel d'un service spécifique comme représenté sur le schéma gauche de la figure 3.5.

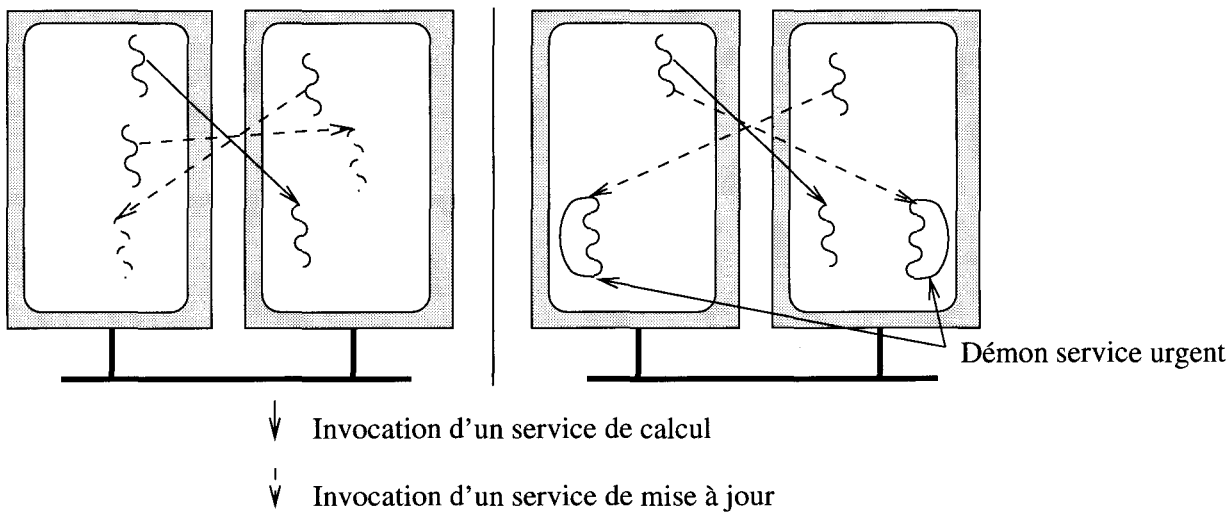


FIG. 3.5 - *Exemple d'implantation d'une application hautement irrégulière avec un environnement proposant l'appel de procédure distant comme paradigme de distribution. Sur le schéma de gauche, implantation traditionnelle. Sur la figure de droite, implantation utilisant les démons services urgents de* ATHAPASCAN-0*b pour éviter la création d'activités supplémentaires ayant une courte durée de vie.*

Une objection possible tient en la nécessité de créer une activité, opération qui reste coûteuse, simplement pour stocker une nouvelle valeur dans une variable. Le ratio $\frac{Nombre\ d'instructions\ de\ l'activité}{Nombre\ d'instructions\ pour\ la\ créer}$ est trop petit, ce qui est une source d'inefficacité. Conscients de ce problème, la plupart des concepteurs d'environnements basés sur l'invocation distante de services ont implanté un mécanisme permettant d'éviter cette création. Dans ATHAPASCAN-0b, il consiste en l'utilisation de processus légers spéciaux, appelés **`Démon services urgents`**, qui exécuteront directement les services déclarés urgents par le programmeur sans créer pour cela d'activités supplémentaires. Ce fonctionnement est représenté sur le schéma droit de la figure 3.5 dans lequel les services de mise à jour sont déclarés

12. On devrait plutôt dire suggérée car il est toujours possible d'en utiliser une autre.

urgents au support d'exécution ce qui implique une prise en compte différente.

Pour résumer, les environnements présentés dans cette section sont particulièrement adaptés aux applications visées. L'aspect structurant est donné par la possibilité de découper l'application en activités indépendantes de manière *intuitive* pour le programmeur et l'aspect efficacité étant assuré par l'optimisation particulière de l'invocation distante de services, qui est la la fonctionnalité exclusivement utilisée.

3.2.3 Conclusion

Dans cette section, nous avons étudié quatre supports d'exécution alliant la programmation distribuée et le parallélisme à grain fin. Nous avons vu les deux principaux schémas de conception proposés : l'envoi de message et l'invocation distante de services, sous les deux angles de la facilité de conception et de l'efficacité. Rappelons que notre étude de ces environnements concerne leur adéquation au développement d'applications hautement irrégulières. Dans cette optique, les fonctionnalités importantes demandées sont :

1. La capacité à gérer efficacement un grand nombre d'activités ;
2. L'adéquation entre le modèle de conception et les caractéristiques de l'application ;
3. Des possibilités avancées d'adaptation du support, donc de paramétrage de l'environnement, pour optimiser l'utilisation des ressources.

Nous allons détailler ces points séparément en essayant de déduire de chacun les contraintes qu'ils supposent sur le support d'exécution dans l'optique de concevoir le support d'exécution « idéal » pour cette classe d'applications.

Gestion efficace d'un grand nombre d'activités Le degré de parallélisme des applications visées étant très important il est crucial, pour une exécution efficace, que les opérations sur les activités soient très efficaces et donc que les mécanismes internes du support soient pensés dans cet esprit. Cela exclut *a priori* l'utilisation de composants génériques. Comme cela est souligné dans [Gin97], cette contrainte va à l'encontre de celle de portabilité et un compromis est donc nécessaire.

Modèle de conception Nous avons vu dans la partie précédente que le modèle de conception basé sur l'invocation distante de services était tout à fait adapté aux caractéristiques des applications visées. Notons que ce modèle peut cependant être complété par d'autres mécanismes pour pallier à sa lourdeur ou son manque d'efficacité dans certains cas précis. Les « démons services urgents » d'ATHAPASCAN-0b sont un exemple d'une telle adaptation.

Possibilités d'adaptation Ce terme générique regroupe la prise en compte de tous les évènements, intérieurs ou extérieurs, qui perturbent l'environnement dans lequel l'application s'exécute. Citons, classiquement, la charge de la machine cible mais aussi la brusque surcharge d'un processeur due à un phénomène de localité de l'application. Ce point est délicat car il impose une interaction entre l'application et son environnement, or il est difficile d'introduire une telle connaissance dans un programme sans le rendre alors extrêmement dépendant du contexte dans lequel il s'exécute et donc de l'architecture sous-jacente. En introduisant de manière *explicite* une telle connaissance dans l'application on lui fait perdre une grande part de portabilité ce qui, au vu de la vitesse d'évolution des architectures parallèles, est une limitation importante.

Les deux premiers points sont plutôt bien pris en compte par les environnements actuels mais le troisième est quant à lui plus compliqué car il suppose de fournir à l'application des éléments qui doivent normalement, pour des raisons de portabilité, lui être cachés. Pour atteindre ce but, la solution est de fournir une abstraction de la machine physique, une machine virtuelle, qui est la vue qu'a l'application de la structure sous-jacente. Le support d'exécution est chargé d'établir la correspondance entre cette machine virtuelle et la machine physique réelle. Ce concept de machine virtuelle est au cœur du système PVM [GBD+95] mais avec un niveau d'abstraction moins avancé que celui que nous visons car il est adapté à un modèle de programmation, l'envoi de messages, moins sophistiqué que celui que proposent les environnements à base de processus légers. Nous allons maintenant voir comment ce concept de virtualisation de l'architecture peut être appliqué à un environnement capable de gérer un grand nombre d'activités donc proposant un grain très fin de parallélisme.

3.3 Présentation de la virtualisation

Dans cette partie nous allons présenter le concept au cœur du projet ESPACE pour un support d'exécution distribué utilisant les processus légers : la virtualisation de l'architecture. Nous allons commencer par décrire le but à atteindre, puis nous détaillerons les moyens nécessaires pour atteindre ce but. Nous nous focaliserons en particulier sur le concept qui permet la mise en place d'une virtualisation totale, la mobilité.

3.3.1 Objectif

Le but de la virtualisation est de fournir au programmeur une abstraction de l'architecture physique sur laquelle son application s'exécutera afin que la conception soit la plus générique (et donc portable) possible. Cela ne doit bien évidemment pas être fourni au détriment de l'efficacité qui reste le souci principal de tout programmeur parallèle.

La virtualisation est composée de deux éléments, un lié à l'expression de la décomposition de l'application en activités et le second concerne la gestion des processus légers prenant en charge ces activités. Ces dernières doivent être prises en compte dans l'optique d'une bonne efficacité. La figure 3.6 illustre le dernier point, le but de cette étape de la virtualisation étant d'assigner un site[13] d'exécution pour les activités de l'application.

La virtualisation est au cœur de la conception de PM^2[Nam97b]. Son modèle de conception et son support d'exécution ont donc été pensés dans ce but. Nous allons détailler dans la suite les concepts et les mécanismes qui ont été mis en place pour atteindre ce but au sein de PM^2.

3.3.1.1 La décomposition en activités

Les deux éléments structurants que nous proposons sont l'invocation distante de services et le clonage léger.

L'invocation distante de services Le premier d'entre eux est le mécanisme déjà décrit précédemment de l'invocation distante de services, nous n'allons donc pas le détailler de nouveau ici.

13. Ou un ensemble de sites comme nous allons le voir dans la suite ...

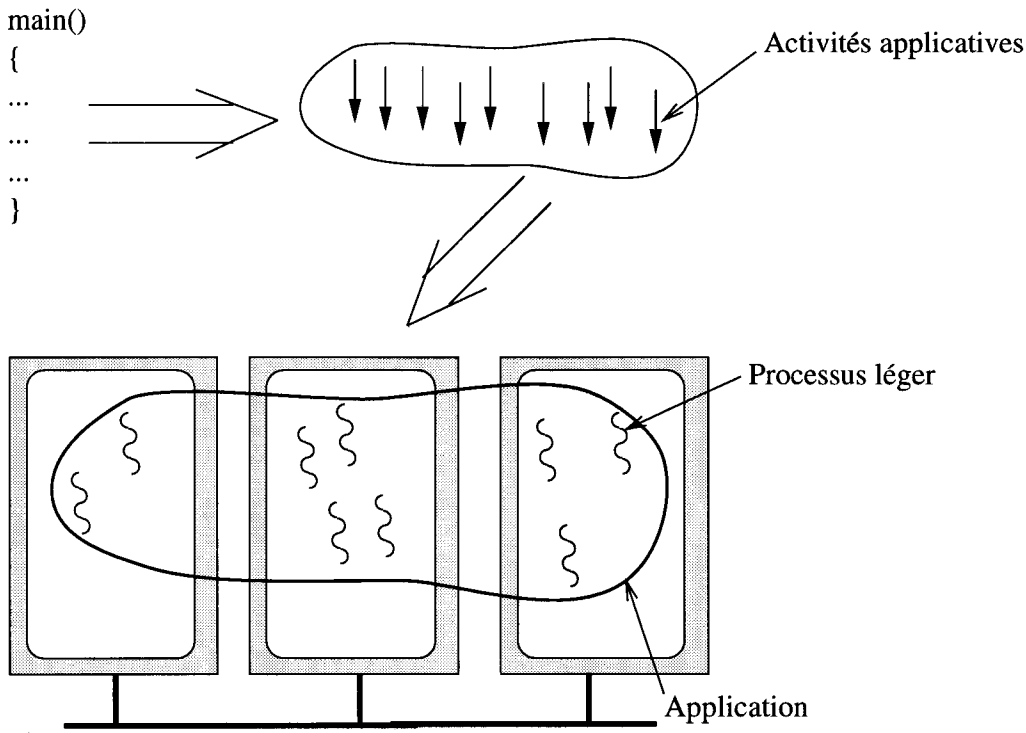


FIG. 3.6 - *La virtualisation consiste à fournir au programmeur un modèle de conception lui permettant d'exprimer de façon intuitive son application en terme d'activités parallèles puis un support pour exécuter efficacement l'application ainsi décomposée.*

Dans PM^2, trois possibilités d'invocation existent :

Synchrone Durant l'invocation de la méthode, l'activité cliente du service est bloquée, elle reprendra son exécution à la fin du service quand les résultats lui auront été renvoyés.

Synchrone à attente différée Pendant l'exécution du service l'activité cliente continue son exécution jusqu'à ce qu'elle ait besoin des résultats. Elle se bloquera alors en les attendant s'ils ne sont pas disponibles.

Asynchrone L'activité créée pour exécuter le service est complètement indépendante de celle qui l'a créée, à aucun moment de l'exécution il n'y a aura de résultat envoyé[14].

La figure 3.7 récapitule ces différents modes d'invocation.

On trouve aussi dans PM^2 un type d'invocation de services, proche des démons services urgents d'ATHAPASCAN-0b qui s'appelle les Quick LRPC. L'exécution d'un service avec ce type d'invocation n'entrainera pas la création d'une activité mais la fonction à exécuter sera prise en charge par une activité système déjà existante.

Le clonage léger Le clonage léger [Nam97a] est un mécanisme original qui a été proposé par Raymond Namyst dans [Nam97b]. Il part de la constatation que l'invocation de services est un

14. Cette méthode d'appel est typiquement utilisée pour créer des activités « démons» de surveillance ou de contrôle. Nous y reviendrons dans le chapitre 4.

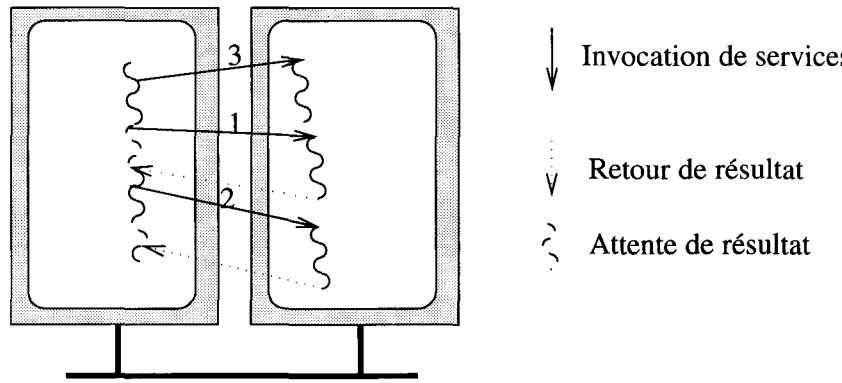


FIG. 3.7 - *Les trois modes possibles d'invocation de services à distance : 1) Synchrone, 2) Synchrone à attente différée, 3) Asynchrone.*

mécanisme assez lourd pour exprimer la décomposition d'une activité en *plusieurs* activités et cela de manière régulière[15], car il oblige à écrire des boucles créant les services puis en attendent les résultats[16] ce qui nuit à la lisibilité du code en obligeant à créer un type de services pour chaque traitement particulier sur les données.

Raymond Namyst a donc proposé un nouveau mécanisme de décomposition qu'il a baptisé clonage léger et qui consiste en deux phases : une de séparation et une de fusion. La séparation consiste à créer un certain nombre, donné par le programmeur, d'activités qui exécuteront exactement le même bloc de code. La fusion se déroule à la fin de ce bloc, elle consistera à réunir toutes les activités créées au moment de la séparation en une seule qui sera, typiquement, chargée de traiter les résultats fournis. Ce fonctionnement est représenté sur la figure 3.8.

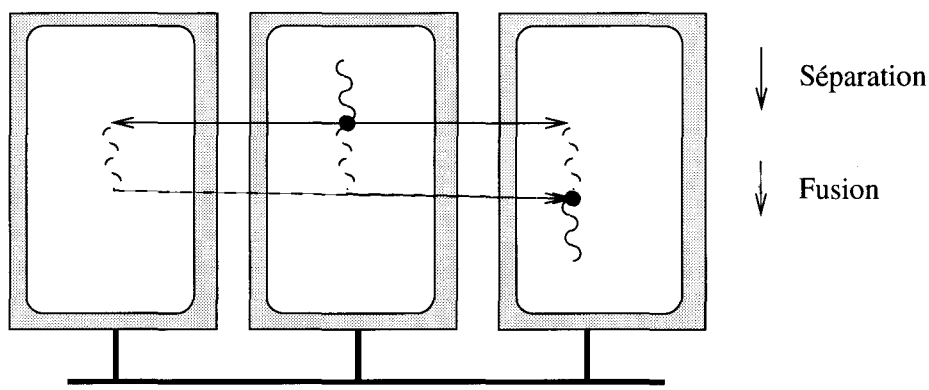


FIG. 3.8 - *Le concept de clonage léger.*

Ce mécanisme peut être assimilé à celui formé des deux opérations fork/join dans la norme

15. Une application hautement irrégulière peut être régulière quand on descend à une granularité très fine, cf les calculs sur les matrices dans l'affectation quadratique (voir annexeA.2), il est donc logique de proposer un tel mécanisme dans un environnement dédié à ce type d'applications.

16. La communauté parallélisme de données a baptisé de telles boucles « boucles de virtualisation».

pthread POSIX avec trois différences qui en font sa particularité :

Distribution Les activités résultant de la séparation peuvent être distribuées sur des processeurs différents, comme cela est le cas sur la figure 3.8.

Rang Chaque activité a un identifiant unique, son rang, qui permet de l'identifier par rapport aux autres. Suivant ce rang des traitements différents ou, plus souvent, des traitements identiques sur des données différentes sont exécutés par les processus légers.

Survivant variable Le join ne se fera pas forcément effectué sur le processus appelant.

Nous allons illustrer l'utilité d'un tel mécanisme sur un exemple concret, celui de la multiplication de matrices. Le grain de décomposition que nous choisissons[17] est l'élément, pour des matrices carrés d'entier de dimension n, n^2 activités seront donc créées. Avec une approche « 100% pure » invocation distante de services il nous faudra définir un service `calcul_cellule` qui prendra en paramètres une ligne et une colonne et qui renverra un entier, il faudra ensuite dérouler une boucle qui lancera tous les services, probablement en utilisant le mode synchrone à attente différée, puis une autre boucle qui attendra le résultat de chaque invocation afin de remplir la matrice résultat.

```
Pour i de 1 à n faire
  Pour j de 1 à n faire
    Invocation_service(calcul_cellule,
        ligne(a,i), colonne(b,j),c(i,j))
  Fait
Fait
Pour i de 1 à n faire
  Pour j de 1 à n faire
    Prend_resultat(c(i,j))
  Fait
Fait
```

```
Séparation(n*n,résultat)
  n_ligne=rang/n;
  n_colonne=rang-(rang/n)*n;
  calcul_cellule(ligne(a,n_ligne),
    colonne(b,n_colonne),
    resultat(n_ligne,n_colonne));
Fusion(n*n,résultat)
  // Remplir c avec le contenu de résultat
```

TAB. 3.1 - *Expression du produit de deux matrices, à gauche avec l'invocation distante de service, à droite avec le clonage léger.*

Les deux façons d'exprimer cette opération, avec et sans le clonage léger, sont présentées dans le tableau 3.1 en utilisant un pseudo-code. On y constate immédiatement la lourdeur de l'écriture avec l'invocation distante seule, qui oblige à définir un service supplémentaire, ce qui nuit à la lisibilité.

Indépendamment de ces considérations qui appartiennent au domaine du génie logiciel, ce mécanisme permet d'introduire des informations *sémantiques* supplémentaires dans la description de l'application en exprimant un lien entre toutes les activités créées lors de la séparation. Cette relation peut être utilisée par d'autres éléments du support d'exécution pour optimiser l'exécution de ces activités en les distribuant cycliquement ou, au contraire, en les regroupant pour éviter des recopies de données. Ce point est une anticipation sur la façon dont les activités formant l'application parallèle sont traitées durant leur exécution, que nous allons détailler maintenant.

3.3.1.2 La gestion des activités

Nous avons vu que pour une complète virtualisation de l'application deux composantes sont nécessaires : des opérations de décomposition en activités et des mécanismes de gestion des activités

17. Voir à ce sujet la discussion de la section 2.1.2 page 20.

pendant leur exécution. Ce dernier point couvre un domaine extrêmement vaste comme nous allons le voir durant une large partie de ce document. Il nécessite en particulier de nombreuses extensions du support d'exécution de base que nous avons présenté depuis le début de ce chapitre. Ces extensions doivent s'appuyer sur des concepts, de plus bas niveau, éléments du support d'exécution. Cette section est dédiée uniquement aux concepts de plus bas niveau nécessaires à la conception de ces éléments supplémentaires, elle contient donc des points assez techniques dont nous tenterons de justifier la nécessité dans le cadre de la virtualisation. Ces concepts sont au nombre de deux : la concurrence et la mobilité.

Concurrence La notion de concurrence signifie que toutes les activités composant l'application s'exécutent concurremment et de façon indépendante et cela quelque soit l'architecture sous-jacente et, en particulier, le nombre de processeurs qu'elle contient. Il serait en effet anormal, et totalement contraire au principe de virtualisation, que le résultat d'une application dépende des conditions dans lesquelles elle a été exécutée, c'est cette éventualité que supprime le concept de concurrence[18].

Une conséquence indirecte du concept de concurrence réside aussi dans la possibilité de faire coexister dans le même espace mémoire, c'est-à-dire dans le même processus lourd, des activités de différents types tant applicatives qu'éléments du support d'exécution. Nous verrons dans la suite que cette possibilité est largement exploitée dans notre proposition de support d'exécution.

Mobilité Le second concept nécessaire à une complète virtualisation est celui de mobilité. Il signifie que le site d'exécution d'une activité, en pratique le processeur, peut changer au cours du temps. Ce changement est typiquement utilisé pour s'adapter aux contraintes, internes ou externes, fluctuantes qui s'exercent sur l'application.

L'idée au centre de ce concept est de s'affranchir de la contrainte de localité traditionnellement présente dans les systèmes de ce type afin de laisser toute latitude au support d'exécution pour le placement des activités tant au moment de leur création qu'à chaque instant de leur exécution. L'opération qui permet cela est appelée la *migration* et consiste à déplacer d'un processeur à l'autre une activité en cours d'exécution comme cela est représenté sur la figure 3.9. Pour que la virtualisation soit complète il est nécessaire que cette opération soit totalement transparente pour les activités, c'est-à-dire qu'elle ne nécessite aucune précaution particulière tant au moment de la conception qu'à celui de l'exécution[19].

Le concept de mobilité est primordial pour une virtualisation complète car il est la clé de voûte de la répartition dynamique des activités sur les processeurs. À ce titre, il est le point essentiel de la prise en compte effective des activités de la virtualisation. Nous étudierons dans la suite sur des supports existants les modèles de conception que la présence de ce mécanisme permet d'envisager.

De nombreux supports d'exécution distribué, à gros grain comme à grain fin, proposent eux aussi un mécanisme similaire à la migration de PM^2. Dans la section suivante, nous allons détailler la façon dont cette fonctionnalité est proposée ainsi que le but visé par son introduction dans les systèmes à grain fin. Cela nous permettra de situer notre proposition dans le cadre global des systèmes distribués au sens large.

18. La concurrence globale peut être rapprochée de la notion d'ordonnancement local en temps partagé que nous étudions dans la suite de ce chapitre.

19. Nous verrons dans la suite que ce but n'est pas encore complètement atteint dans PM^2.

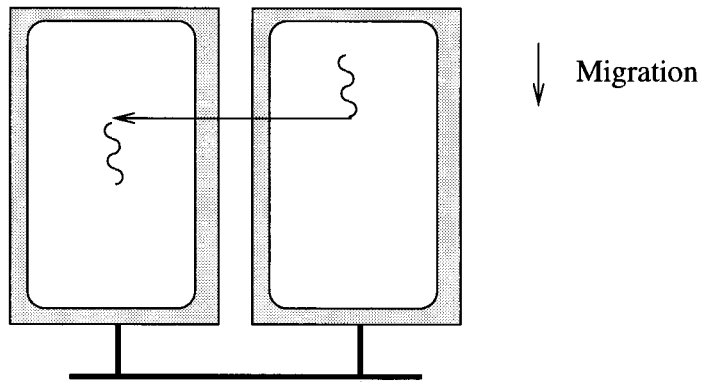


FIG. 3.9 - *Le concept de migration.*

3.4 Utilisation de la mobilité dans les systèmes distribués

Dans cette section, nous allons étudier comment le concept de mobilité, au sens où nous l'avons décrit dans la section précédente, est vu et utilisé dans d'autres supports distribués utilisant aussi une fine granularité pour les activités. Nous ne nous intéresserons pas ici à la mobilité dans le cadre des environnements manipulant des activités de gros grain comme les processus lourds [Zay87] car ils répondent à une problématique différente, comme la tolérance aux fautes par exemple, et sortent donc du cadre de cette étude.

Notre but est ici de confronter les approches afin de mettre le concept de mobilité comme élément de la virtualisation en perspective avec les autres utilisations possibles de la mobilité tant d'un point de vue modèle que technique. Dans ce but nous allons étudier trois systèmes fournissant un mécanisme comparable : ARIADNE, MILLIPEDE et EMERALD.

3.4.1 Ariadne

Le support d'exécution ARIADNE[20] [MR95, MV] a été conçu à l'origine pour des applications de simulation dans un cadre distribué. Du fait du grand nombre potentiel d'activités contenue dans ce type d'applications l'utilisation des processus légers s'est tout de suite imposée comme une évidence aux yeux des concepteurs de ce support.

Modèle de programmation Les concepteurs de ce système voient une applications distribuée comme des activités et des données réparties sur les processeurs, le but du support étant de gérer ces deux types d'entités efficacement, en particulier en tentant de minimiser les communications. C'est dans ce but qu'ils introduisent le concept de mobilité des activités. Leur constat est que la réduction du volume des communications peut se faire de deux manières :

1. en rapprochant les données des activités,

2. en rapprochant les activités des données.

La première approche implique un mouvement des données et donc un système de référencement global pour permettre à une activité d'accéder à une donnée quelconque qui peut en particulier

20. Site Web du projet : `http://www.cs.purdue.edu/research/PaCS/ariadne.html`

être située sur un site différent. Une fois cela disponible, le déplacement des données ne pose pas de problèmes techniques particuliers.

Cependant, dans les applications visées par ce système, les données sont de taille importante et les déplacer est donc coûteux. L'approche duale de la précédente est donc plus adaptée car les activités sont de faible granularité et peuvent donc être déplacées à moindre coût. La mobilité est introduite dans ce cas pour permettre de rapprocher les activités des données afin de minimiser les coûts de communication. Cependant, il reste possible de déplacer aussi les données dans le même but. Il faut alors prendre soin de ne pas utiliser deux politiques, une pour les données et une pour les activités, qui se «parasiteraient » en prenant des décisions opposées.

Pour éviter cela, les auteurs préconisent une régulation faite sur les données, les activités utilisant leur capacité de mobilité pour les suivre. Dans ce but ils fournissent un mécanisme nommé « object localisator » dont le but est de conserver la localisation, le processeur sur lequel il réside, de chaque objet global. Les activités utilisent ce mécanisme pour trouver le processus vers lequel elles se déplaceront car il contient l'objet ou les objets qui les intéressent. Notons toutefois que le mécanisme de migration utilisé dans ARIADNE impose que ce soit l'activité qui demande *explicitement* son déplacement vers un autre processeur. Ce n'est pas gênant dans l'optique des concepteurs d'ARIADNE puisque les activités suivent les données et savent donc à quel moment elles doivent se déplacer, avant d'accéder à une donnée distante.

Les objectifs des concepteurs d'ARIADNE sont différents des nôtres car ils n'ont pas l'intention de fournir un niveau d'abstraction aussi élevé que celui visé par la virtualisation. En particulier, ils laissent la gestion des activités et des données à la charge du programmeur au lieu de lui fournir un modèle complet d'exécution. Dans cette optique, leur vision de la mobilité est plus restrictive que celle que nous proposons car elle impose que les activités migrent d'elles-mêmes ce qui signifie qu'elles ne peuvent pas être déplacées de manière transparente. Conséquemment ils excluent la possibilité de concevoir des mécanismes génériques, c'est-à-dire indépendants de l'application, de régulation de charge sur les activités, seules les données peuvent être gérées de manière transparente par un sur-ensemble du support d'exécution.

3.4.2 Millipede

MILLIPEDE[21] [SS97, ISS96] a des objectifs similaires à ceux de NEXUS et se positionne donc au même niveau que lui. Son but est d'être un support pour la compilation et l'exécution d'applications écrites en différents langages tels que HPF ou Java. L'interaction entre l'application et MILLIPEDE peut se faire à différents niveaux grâce à l'utilisation de différentes API qui permettent de contrôler très finement les conditions d'exécution. Il est aussi à noter que les fonctionnalités offertes par MILLIPEDE se superposent avec celles du système sous-jacent sans les recouvrir. Par exemple des threads système, chargés des entrées/sorties par exemple, cohabiteront dans une application avec ceux gérés par MILLIPEDE qui auront en plus la capacité de mobilité.

Modèle de programmation Le modèle de programmation proposé par MILLIPEDE est bâti autour de deux concepts: la mémoire virtuellement partagée et les processus communicants. La mémoire partagée proposée est classiquement structurée en pages et différents modèles de cohérence sont proposés allant du plus fort (accès strictement sérialisé) au plus relâché (mise à jour sur demande explicite). L'envoi de messages entre les activités est avant tout prévu pour les opérations de synchronisation, le paradigme principal de distribution est la mémoire partagée.

21. Site web du projet MILLIPEDE : `http://www.cs.technion.il/Labs/Millipede`.

Les processus légers peuvent être déplacés entre les processeurs sans limitation. Les problèmes de translation d'adresses locales traditionnellement associés à la migration sont ici résolus en réservant sur tous les processeurs l'espace mémoire correspondant à une activité. Ainsi à la création d'une activité i la place mémoire est réservée pour cette activité exactement aux mêmes adresses sur tous les processeurs. Ainsi, la migration se fait sans nécessiter de translations des adresses de pile puisque l'espace mémoire utilisée par l'activité après sa migration sera le même que celui qu'elle a quittée, il sera simplement situé sur un autre processeur. La limitation principale de cette technique est bien sûr qu'elle limite le nombre d'activités exécutables simultanément, le degré de parallélisme maximal de l'application étant borné par celui d'une machine. Il s'agit d'une limitation importante dans le contexte de la virtualisation où le but est justement de s'affranchir de ce type de contraintes pour laisser le programmeur utiliser un grain de parallélisme, et donc un nombre d'activités, aussi fin qu'il le désire.

Le mélange de la mobilité et de l'envoi de messages est souvent source d'inefficacité à cause de la nécessité de laisser des proxys chargés de traiter les messages qui arrivent après qu'une migration ait été effectuée. Cela peut amener à faire suivre par le message une chaine de proxys qui rallonge inutilement le temps de communication. MILLIPEDE contourne ce problème en fournissant une abstraction de communication nommée « Job event». À chaque processus léger est associée une chaîne d'évènements qui est le moyen unique de communication avec lui, avec la mémoire partagée bien entendu. Cette chaîne d'évènements, qui est entièrement gérée par le système, permet au programmeur de s'affranchir de la contrainte de localité traditionnellement associée aux communications point à point.

Les mécanismes de synchronisation, tels que les sémaphores, offerts par MILLIPEDE sont conçus avec des processus légers ce qui leur permet de bénéficier de la mobilité et les rend gérables[22] par le support d'exécution. Ainsi un sémaphore peut être déplacé pour se rapprocher des processus légers qui font le plus appel à lui afin de minimiser le coût des communications. Cette opération est du ressort du système et étant donc totalement transparente pour l'application.

Utilisation de la mobilité La mobilité est utilisée dans MILLIPEDE pour atteindre plusieurs objectifs :

1. **Minimiser les communications** Il est acquis qu'une application utilisant une mémoire partagée a un coût plus élevé en communications que la même application utilisant l'envoi de messages, ce coût étant le prix à payer pour la facilité de programmation dans ce modèle. La mobilité est utilisée dans MILLIPEDE pour minimiser ce coût en rapprochant les activités des pages qu'elles utilisent[23].

2. **Répartir la charge** Pour minimiser le temps d'exécution, il est important de maintenir actifs la totalité des processeurs disponibles et la mobilité des activités est utilisée dans ce but. Il est intéressant de noter que ce but et le précédent imposent des contraintes duales, atteindre l'un revient à négliger l'autre. Nous verrons dans la suite à quel compromis sont parvenus les concepteurs de MILLIPEDE.

3. **Optimiser le temps d'utilisation des processeurs** Un des objectifs de MILLIPEDE est de fonctionner dans un réseau de stations de travail partagées entre les utilisateurs, en récupérant les périodes d'inactivité des machines individuelles pour y exécuter les applications des autres

22. Dans le sens « qui peut être géré».

23. Ou, dans le cas d'une machine SMP, du même nœud de calcul regroupant des processeurs et de la mémoire physiquement partagée.

utilisateurs. La mobilité est utilisée pour déployer des activités sur des machines inactives et les en retirer quand l'utilisateur habituel de la station l'utilise de nouveau. On trouve aussi cette idée dans les systèmes MARS[Haf96, Haf98] et GATOSTAR[FFS95].

La façon dont MILLIPEDE gère la localisation des activités est complexe car elle prend en compte différents critères destinés à atteindre les objectifs énoncés ci-dessus. Son étude déborde du cadre strict de l'utilisation de la migration pour rejoindre celui plus large de l'ordonnancement, c'est pourquoi nous y revenons en détail dans la section 3.5.2 page 72 consacrée à l'ordonnancement dynamique des applications de granularité fine.

3.4.3 Emerald

EMERALD[SJ90] est un système entièrement basé sur la notion d'objet, chaque objet résidant sur un processeur et pouvant être déplacé de manière transparente. Il se compose d'un compilateur pour un langage spécifique et d'un environnement d'exécution pour les applications issues du compilateur.

Toutes les données en EMERALD sont des objets. Les objets peuvent référencer des objets locaux ou situés sur d'autres processeurs de manière identique (mais avec des performances différentes). Les objets sont référencés par des identificateurs d'objet qui sont une abstraction permettant de les désigner indépendamment de leur localisation sans utiliser de mécanismes génériques de mémoire partagée. Le seul moyen qu'ont deux activités pour partager un objet consiste à posséder la même référence, comme les références sont indépendantes de la localisation des objets les activités peuvent se déplacer indépendamment les unes des autres.

Les objets peuvent se déplacer librement entre les processeurs formant l'architecture parallèle. Le but de ces déplacements est de rendre EMERALD robuste, en particulier d'augmenter sa tolérance aux fautes. Quand une activité exécute une méthode d'un objet, elle invoque un service sur cet objet et crée donc un flot d'exécution « à l'intérieur » de ce dernier. Si, durant ce temps, l'objet est déplacé alors les opérations s'exécutant sur cet objet sont elles aussi déplacées. C'est à cela que sert la mobilité des activités. Le déplacement d'un objet pouvant survenir à n'importe quel moment de l'exécution, la migration doit être préemptive, c'est-à-dire qu'une activité doit être migrable de manière transparente indépendamment de l'état dans lequel elle se trouve. Au-delà de l'aspect modèle, ce qui fait l'originalité de la migration dans EMERALD concerne ses développements les plus récents qui visent à fournir la mobilité dans un cadre hétérogène.

Les environnements que nous avons étudiés jusqu'à présent ne proposaientt d'utiliser la migration que dans un cadre homogène, c'est-à-dire entre des architectures rigoureusement identiques tant du point de vue matériel que logiciel. Dans [SJ95] les concepteurs d'EMERALD proposent d'étendre la mobilité aux architectures hétérogènes. Cette approche est originale et met en évidence la complexité de mise en place d'un tel mécanisme, nous allons détailler leurs travaux sur ce point.

La migration hétérogène est relativement aisée à mettre en place par l'utilisation d'un langage interprété qui permet de s'affranchir des environnements liés aux processeurs. Le problème de cette approche est son inefficacité. Les concepteurs d'EMERALD ont eux décidé d'exécuter du code natif, donc sans perte d'efficacité, mais en mettant en place un mécanisme qui leur permet à tout instant de geler une activité dans un format générique. Pour migrer une activité il en est donc pris une photo qui est ensuite traduite dans ce format. Le résultat est alors transféré vers le processeur destinataire où un mécanisme de traduction inverse lui est appliqué pour rendre l'activité exécutable sur son nouvel hôte. Le coût de ces opérations de traduction n'est pas prohibitif, les auteurs estiment le surcoût par rapport à la migration en contexte homogène à 60%.

Ces opérations de traduction sont possibles parce que le support fonctionne de pair avec le compilateur. EMERALD étant un langage à part entière, il est possible d'extraire au moment de la compilation une grande quantité d'informations permettant d'aider le support à garder et restaurer l'environnement d'une activité. Dans un cadre plus général, tel que celui dans lequel s'inscrivent MILLIPEDE ou PM^2, cela n'est pas possible.

3.4.4 PM^2

La migration que nous proposons dans PM^2 est l'un des points clé permettant la virtualisation. Son but est de s'affranchir de la contrainte de localité habituellement associée aux processus légers pour pouvoir déléguer la prise en charge dynamique des activités de l'application à un élément du support d'exécution. Dans cette optique elle a les caractéristiques suivantes :

Transparente Il n'est pas besoin de modifier le code de l'application pour ajouter des appels explicites pour migrer (comme c'est le cas dans ARIADNE).

Préemptive La migration se fait à l'insu de l'activité, si elle n'essaie pas de le savoir l'activité ne découvrira jamais qu'elle a été migrée. Il est cependant possible pour une activité de préciser qu'elle ne veut pas être migrée pendant un certain temps, par exemple pendant des opérations d'accès sur un fichier.

Générale Contrairement à MILLIPEDE qui, pour proposer la migration ajoute une contrainte assez forte à son support d'exécution (à savoir l'idendité des adresses de pile sur tous les processeurs), elle est intégrée dans PM^2 sans coût supplémentaire ni au stade conceptuel ni au stade exécutif.

Ces deux dernières caractéristique sont très importantes dans l'optique de la virtualisation car elles permettent de concevoir des politiques de placement des activités complètement indépendantes des applications visées.

Un « effet de bord » de la migration transparente et préemptive est qu'elle permet la modification de granularité de l'application. En effet déplacer une activité d'un site vers un autre peut être vu comme couper une activité en deux, la première partie s'étant exécutée sur le processeur d'où elle vient, la seconde sur celui où elle arrive. Cette notion est très intéressante car elle permet à une application existante dont le programmeur ne veut pas, pour des raisons de coût de développement, changer la granularité d'être gérée par le support d'exécution comme étant à grain fin quand même. Elle permet en fait d'*adapter* la granularité de l'application à la granularité du support et cela de manière transparente. Le but poursuivi par cette adaptation est bien entendu de meilleures performances, cette fonctionnalité est désactivable à tout moment pour éviter de réduire le grain d'une application qui est plus performante avec une granularité élevée.

La migration dans PM^2 souffre quand même de deux limitations :

Homogénéité La migration ne peut se faire que dans un contexte homogène. Contrairement à EMERALD il n'est pas possible de migrer une activité entre une station Sun et un PC. Cette limitation résulte du choix fait au départ du projet ESPACE d'un environnement qui soit facilement[24] portable, or la migration hétérogène nécessite la mise en place d'outils spécifiques, à commencer par un compilateur, qui limitent cette portabilité.

24. La plupart des portages de PM^2 sur une nouvelle architecture se font en un ou deux jours. Seules quelques architectures assez spécifiques (Cray T3D, Capitan, ...) ont posé problème.

Prise en compte des pointeurs Un pointeur qui pointe vers une donnée locale est un casse-tête pour la migration car il oblige, pour être traité de manière complètement transparente, à mettre en place des mécanismes lourds. L'approche choisie dans PM^2 fait une distinction entre les pointeurs sur les données locales à l'activité et les globaux, pointant sur une zone de données hors de l'espace mémoire de l'activité. Dans le premier cas les pointeurs doivent être enregistrés explicitement par l'application auprès du support d'exécution et ils seront translatés au moment de la migration de manière transparente, sans nécessiter d'interventions supplémentaires de l'application. Dans le second cas, il est possible d'associer aux activités des fonctions qui se chargeront du transfert des données globales utilisées par l'activité en migration.

Notons cependant que l'utilisation de données globales est contraire au modèle de conception proposé par la virtualisation où les activités sont indépendantes.

La migration dans PM^2 répond aux critères fixés pour une virtualisation efficace en étant préemptive et en n'ajoutant pas de contraintes fortes supplémentaires comme c'est le cas dans MILLIPEDE. Sa seule limitation significative concerne le traitement des pointeurs mais leur utilisation est peu fréquente dans la classe d'applications qui nous intéressent en priorité. La prise en compte de l'hétérogénéité pour des applications quelconques est trop coûteuse tant du point de vue conception qu'efficacité, nous avons donc décidé de ne pas prendre en compte cet aspect dans PM^2. L'évolution de la plate-forme est plutôt concentrée sur les aspects liés au multi-clustering. Une architecture parallèle est alors vue comme un ensemble hétérogène de clusters de processeurs qui sont eux homogènes. Les activités sont placées à leur création à l'intérieur d'un cluster dans lequel elles peuvent se déplacer entre les processeurs mais dont elles ne peuvent pas sortir.

3.4.5 Conclusion

La migration est essentiellement utilisée dans les systèmes distribués à grain fin pour atteindre deux objectifs :

Limiter les communications en rapprochant les activités des données. C'est le cas dans ARIADNE et, en partie, dans MILLIPEDE.

Équilibrer la charge en terme d'activités afin d'utiliser au maximum les ressources de calcul de l'architecture parallèle en utilisant par exemple les machines inactives d'un réseau.

La caractéristique principale de la migration légère réside dans son faible coût par rapport à la migration de processus lourds. Concernant PM^2, les travaux les plus récents [NM97] ont permis d'obtenir des temps de migration de 80 μs[25] pour la migration d'une activité de taille moyenne. Des temps aussi bas permettent d'envisager le déplacement fréquent des activités. Il est alors possible de tester les performances de plusieurs localisations d'une activité pour déterminer la meilleure ce qui revient à utiliser une stratégie incrémentale, qui apprend de ses erreurs.

La migration n'est que l'outil sur lequel s'appuie les supports d'exécution pour placer et déplacer les activités dans le but d'optimiser les performances. Nous avons étudié dans cette partie le moyen technique et le but que les environnements visent. Nous n'avons pas étudié l'élément principal qu'est la politique de placement des activités et la manière dont elle fonctionne au niveau conceptuel (quelles informations utilise-t-elle ?) et exécutif (comment est-elle mise en place ?). Cette problématique est celle de l'ordonnancement et son étude fait l'objet de la section suivante. Nous

25. Ces performances ont été obtenues sur un réseau de PC reliés par Myrinet.

allons y détailler quelques systèmes typiques des différentes catégories d'ordonnanceurs pour des activités de granularité fine, certains utilisant la migration et d'autres non.

3.5 Ordonnancement distribué d'activités à grain fin

La mobilité permet de remettre en cause à tout moment le placement des activités de manière transparente. L'utiliser signifie donc choisir les sites d'exécution des activités, ce choix visant toujours à maximixer (ou minimiser) un ou plusieurs critères qui conditionnent l'efficacité de l'exécution de l'application. Cette dernière peut, de plus, avoir des contraintes de localité supplémentaires, typiquement liées à l'accès aux données. Virtualiser efficacement nécessite donc de calculer les sites d'exécution des activités afin d'atteindre un but de performances en respectant des contraintes propres à l'application.

C'est le but de l'ordonnancement qui va être étudié cette section. Nous allons tout d'abord définir la notion d'ordonnancement et préciser pourquoi virtualiser implique ordonnancer. Nous présenterons ensuite les approches typiques d'ordonnancement en étudiant en détail celles que proposent MILLIPEDE, CILK et ATHAPASCAN-1. Nous verrons ensuite en quoi le concept de mobilité influe sur les mécanismes traditionnels d'ordonnancement et nécessite l'introduction d'une nouvelle approche.

3.5.1 Virtualiser $\Longrightarrow$ Ordonnancer

L'ordonnancement est un sujet très ancien, largement antérieur à la naissance de l'informatique. Il consiste à planifier, pour un ensemble de tâches, le moment et éventuellement l'endroit où elles seront effectuées. La construction d'une maison est un problème classique d'ordonnancement car certaines tâches doivent être effectuées avant d'autres, il vaut mieux poser la toiture après que les murs ont été batis par exemple. D'une manière générale le problème de l'ordonnancement consiste à fournir dans un certain ordre des clients à des ressources (voir figure 3.10).

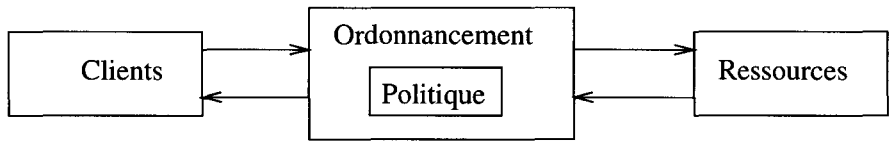


FIG. 3.10 - *Un système d'ordonnancement*

Dans le cadre qui nous intéresse, les clients sont les activités de l'application parallèle et les ressources sont les éléments de calcul c'est-à-dire les processeurs. Calculer un ordonnancement consiste donc, au moins, à assigner à chaque activité un processeur, une ressource, sur lequel elle s'exécutera. Cela n'est cependant pas suffisant car des dépendances temporelles existent aussi entre les activités : en plus d'un endroit où s'exécuter un algorithme d'ordonnancement doit aussi préciser un moment, une date, d'exécution. Ordonnancer une application parallèle consiste donc à fixer pour toutes les activités qui la composent une date et un lieu d'exécution dans le but de minimiser (ou maximiser) un (ou des) critères. Ce choix est très souvent dépendant des caractéristiques de l'application, il utilise donc des informations la concernant, ces informations pouvant être données explicitement ou être extraites de façon transparente de l'application comme nous allons le voir dans la suite.

Suivant les caractéristiques de l'application [26] le calcul de l'ordonnancement peut être entièrement réalisé avant l'exécution de l'application, on parle alors d'ordonnancement statique ou encore déterministe, ou bien en partie ou en totalité durant l'exécution et on parle alors d'ordonnancement indéterministe. Dans les deux cas, la caractéristique principale utilisée pour calculer cet ordonnancement est le graphe de dépendances de l'application qui exprime les relations existant entre les activités qui la composent. Dans le cas où le comportement est non déterministe l'ordonnancement peut être dynamique mais guidé par des informations extraites avant l'exécution ou, au contraire, entièrement dynamique. Comme nous l'avons présenté dans l'introduction de ce document, les problèmes qui nous intéressent ne peuvent pas être ordonnancés de façon entièrement statique, nous ne détaillerons donc pas cette classe d'ordonnanceur. Le lecteur intéressé peut se reporter à [ERLA94].

L'aspect purement dynamique de la virtualisation, qui consiste à placer les activités afin de minimiser certains critères, peut être assimilé à de l'ordonnancement. En effet, il consiste à placer et déplacer les activités constituant l'application sur les processeurs de l'architecture cible et cela de manière transparente pour l'application, alors que l'ordonnancement tel que nous l'avons défini précédemment consiste à affecter à chaque activité une date et un lieu d'exécution. Dans le cas qui nous intéresse, on parle d'*ordonnancement préemptif*. De plus, les applications que nous visons ont comme caractéristiques supplémentaires leur haut degré d'irrégularité et un nombre élevé d'activités, deux particularités qui rendent complexe, voire impossible, le calcul d'un ordonnancement optimal.

Nous allons étudier dans la suite trois exemples de systèmes proposant un ordonnancement pour les applications irrégulières, pour lesquelles il n'est donc pas possible de calculer statiquement un ordonnancement, utilisant de plus une granularité fine d'activités. Le premier, MILLIPEDE, bénéficie du mécanisme de migration présenté dans la section précédente. Les deux autres, CILK et ATHAPASCAN-1, proposent un ordonnancement utilisant le placement au moment de la création. Cette étude se focalisera plus particulièrement sur les critères utilisés pour l'ordonnancement et sur la manière dont sont choisi les sites et les dates d'exécution des activités à partir de ces critères.

3.5.2 Ordonnancement dynamique pour activités de granularité fine

L'ordonnancement dynamique d'activités parallèles est un domaine extrêmement vaste pour lequel il est impossible de construire des outils génériques performants, même si ces outils s'appuient tous sur une thématique et des raisonnements similaires. Par exemple, il n'est pas envisageable de construire un outil qui effectuerait de manière efficace aussi bien l'ordonnancement d'une application formée exclusivement de processus lourds communicants non migrables que celui d'une application composée de processus légers utilisant l'invocation distante de services. À chaque modèle de programmation correspond un type précis d'ordonnanceur conçu pour prendre ses particularités en compte.

Les problèmes que nous étudions sont irréguliers et utilisent un modèle de programmation basé sur l'invocation distante de services et offrant une faible granularité de calcul, les ordonnanceurs que nous allons présenter visent cette classe d'applications. Ils se partagent en deux grandes catégories, l'une utilisant le graphe de dépendances de l'application, à laquelle appartiennent CILK et ATHAPASCAN-1, et l'autre effectuant un ordonnancement réactif, MILLIPEDE.

Les particularités des environnements visés sont liées à la faible granularité de calcul proposée, qui a deux conséquences principales :

Prise en compte de dépendances fines Les activités manipulées étant fines, elles agissent sur

26. En particulier son degré d'irrégularité comme nous l'avons vu dans l'introduction.

de petites quantités de données ce qui permet un ordonnancement qui tient compte de dépendances très fines sur les données. Par exemple, là ou un ordonnanceur d'un environnement à gros grain utilisera des dépendances entre les matrices, un à grain fin travaillera sur les vecteurs voire les éléments de la matrice.

Migration peu coûteuse Le reproche principalement fait à la migration est son coût qui est important quand l'activité à déplacer est à gros grain (plusieurs secondes avec MPVM [CKO+94, CCK+95] par exemple). En conséquence, l'ordonnanceur n'a pas droit à l'erreur quand il prend une telle décision. Le coût d'une telle opération étant bien moindre pour une activité de faible granularité (quelques dizaines de microsecondes pour PM^2 sur un réseau haut débit) il est parfaitement envisageable de migrer plusieurs fois une activité durant son exécution.

Nous allons voir que ces deux caractéristiques jouent un rôle important dans les choix des concepteurs des ordonnanceurs que nous allons étudier.

3.5.2.1 Millipede

Nous avons étudié MILLIPEDE dans la partie consacréee à la migration. Rappelons brièvement l'objectif de la migration dans ce système. La migration est utilisée pour ordonnancer les activités de l'application afin d'atteindre des performances optimales. Pour atteindre ce but, il faut trouver le compromis idéal entre charge minimale, communication minimale et coût de l'ordonnancement.

Méthode d'ordonnancement Le but de l'ordonnanceur de MILLIPEDE est de répartir la charge, en terme de nombre d'activités sur chaque processeur, tout en minimisant les coûts de communication. Le fonctionnement de l'ordonnanceur est entièrement dynamique, il n'utilise aucune information extraite au moment de la compilation. De plus, et contrairement aux deux ordonnanceurs présentés dans la suite, il ne construit pas de graphe de dépendances sur les activités mais fonctionne de manière entièrement *réactive* en déplaçant les activités grâce à la fonctionnalité de migration du support.

Une des particularités de l'ordonnanceur MILLIPEDE réside dans la séparation entre la décision de déclenchement de l'équilibrage et la manière dont il est effectué, c'est-à-dire les activités qui sont effectivement déplacées. La politique de répartition des activités peut être laissée à la charge du programmeur qui peut donc utiliser le critère d'équilibrage qu'il veut (classiquement le nombre d'activités mais cela peut aussi être la quantité de mémoire, locale et non pas partagée, utilisée). Quand il constate un déséquilibre il demande au support le déplacement de n activités du processeur P_i vers le processeur P_j afin d'équilibrer la charge. Le choix des activités qui seront déplacées ainsi que le transport sont faits *par le support.* On a donc ici une séparation entre la décision d'équilibrer, prise par une politique définie par le programmeur, et le choix des activités effectivement déplacées, qui sera lui pris par le support. Cette séparation est nécessaire car le déplacement tend à diminuer les coûts de communication en rapprochant les activités des pages de mémoire qu'elles utilisent. Pour cela il faut utiliser des informations sur les pages accédées par les activités, or ces informations ne sont disponibles que dans le support d'exécution et il serait donc dangereux pour l'efficacité que le programmeur, qui ne dispose pas des informations adéquates, effectue ce choix.

Le problème délicat que doit résoudre le support d'exécution réside dans le choix des activités à déplacer, leur nombre et le site destinataire ayant été déterminés par la méthode d'équilibrage. Ce choix est fait en utilisant une méthode heuristique qui exploite les informations d'accès aux pages, collectées par le support d'exécution. Pour minimiser le coût, tant en temps qu'en espace mémoire,

de la collecte de ces informations, seuls les accès à des pages mémoire distantes sont mémorisés, et non les accès locaux. Cela implique que l'optimalité des décisions prises ne peut pas être garantie. Les auteurs soulignent cependant que même une mauvaise décision est bénéfique car elle fournit des informations supplémentaires permettant de prendre de meilleures décisions ultérieurement. Cela est dûe au fait que seuls les accès distants sont utilisés dans le processus de décision ; si l'ensemble des accès l'étaient le déplacement des pages n'apporterait pas d'informations supplémentaires.

Les critères utilisés pour choisir les activités à déplacer d'un site S vers un site D sont les suivants :

1. **Fréquence maximale des accès vers des pages situées sur le site D** afin de réduire les communications distantes.

2. **Fréquence maximale d'accès distants** Il faut appliquer à ce critère un poids moindre qu'au précédent car il ne réduit pas le volume des communications. Il sert principalement à éviter que des activités effectuant beaucoup d'accès locaux soient déplacées.

3. **Fréquence d'accès minimale des activités restant sur S aux pages utilisées par les activités déplacées** Le but de ce critère est de minimiser les accès à ces pages par les activités restant sur S. Au mieux, le nombre de ces accès sera 0 et les pages pourront être déplacées sur D.

4. **Fréquence minimale d'accès aux pages locales** Comme nous l'avons mentionné, cette information n'est pas conservée par le système par contre de plus anciennes peuvent exister datant du moment où soit les activités candidates, soit les pages, étaient sur un site différent de S.

Ces critères étant souvent en conflit, un compromis est nécessaire. La décision est prise en évaluant pour chaque activité l'amélioration induite par son déplacement, ce qui permet de les classer et donc de choisir les plus intéressantes. Notons qu'au début de l'exécution très peu d'informations sont disponibles pour prendre les décisions, une heuristique supplémentaire est donc utilisée qui dit que deux activités créées successivement ont de fortes chances de partager des données et doivent donc être localisées sur le même site.

Le comportement typique de l'ordonnanceur commencera par une période instable pendant laquelle de nombreux mouvements d'activités prendront place, puis au fur et à mesure que la quantité d'informations de communication tracées par le support d'exécution augmentera l'application atteindra un niveau d'équilibre et les mouvements seront alors plus rares. Le seul problème, que nous avons déjà évoqué, réside dans le phénomène de ping-pong d'activités entre deux hôtes qui peut apparaître si un équilibrage parfait n'est pas possible (dans le cas d'un nombre impair d'activités sur un nombre pair d'hôtes par exemple). Ce cas particulier est traité par un module spécifique dont le rôle est de détecter de telles anomalies et d'y remédier. Nous ne le détaillons pas ici et invitons le lecteur intéressé à se reporter à [SS97].

Pour résumer, le système MILLIPEDE utilise des informations sur le comportement de l'application glanées durant l'exécution de celle-ci. Il ordonnance en réagissant aux déséquilibres qu'il constate, il n'essaie pas d'anticiper l'apparition d'un tel déséquilibre sauf au tout début de l'exécution quand il ne dispose pas encore d'informatons suffisantes pour faire un bon choix. Nous allons maintenant étudier un système qui va, lui, utiliser des informations extraites de la structure de l'application, des dépendances entre les activités qui la composent, pour calculer l'ordonnancement.

3.5.2.2 Cilk

CILK [27][cil97] est un langage algorithmique multithreadé. La philosophie qui a prévalu à sa conception est que le programmeur devait se concentrer sur la structuration de son application en terme d'activités parallèles et de contraintes de localité, en laissant au support d'exécution la responsabilité de l'ordonnancer efficacement sur une architecture donnée. La particularité de CILK est que son support d'exécution garantit des performances efficaces et prévisibles. Dans cette section nous allons étudier le modèle de programmation proposé par CILK ainsi que l'approche utilisée pour l'ordonnancement des applications ainsi créées. Nous regarderons en particulier le modèle théorique développé par les concepteurs de CILK pour prouver l'optimalité de leur ordonnancement. La version que nous présentons est la 5.1 qui est disponible depuis septembre 97.

Modèle de programmation CILK propose un langage de programmation original [cil97] qui est une extension du C auquel il ajoute des opérations parallèles. Elles sont au nombre de trois :

1. `cilk` Le mot clé `cilk` sert à préfixer une procédure C pour en faire une procédure CILK c'est-à-dire qu'elle peut devenir une activité.

2. `spawn` Le mot clé `spawn` permet de créer une nouvelle activité afin d'exécuter une fonction qui aura été définie auparavant avec le mot clé `cilk`. À part l'utilisation de ce mot clé l'appel se fait exactement de la même manière qu'un appel classique de fonction en C. La seule différence, mais elle est de taille, est que l'activité qui a effectué le `spawn` continue son exécution concurremment avec l'activité nouvellement créée.

3. `sync` Le troisième mot clé introduit est utilisé pour la synchronisation entre activités. Quand une activité veut utiliser les résultats que ses fils (qu'elle a créés par un `spawn`) calculent, elle doit être sûre qu'ils ont tous terminé leur exécution. C'est le rôle de l'instruction `sync` qui bloque l'activité qui l'a invoquée jusqu'à ce que tous ses fils aient terminé leur exécution. C'est donc une opération de synchronisation locale car elle ne concerne que les fils de l'activité qui l'a appelée et elle-même bien entendu.

La figure 3.11 illustre le rôle de ces mécanismes dans un programme de résolution du célèbre problème de Fibonacci. L'algorithme utilisé est classiquement récursif, chaque appel correspondant à la création d'une activité nouvelle. Avant de retourner le résultat chaque activité attend le résultat de ses fils en effectuant une opération `sync` pour s'assurer que les valeurs contenues dans les variables `x` et `y` sont exactes. La fonction `main` fait de même pour attendre le résultat avant de l'afficher [28].

Ordonnancement des activités L'ordonnancement d'activités dans CILK est fait de manière entièrement dynamique. Le support utilise le graphe de dépendances des activités qu'il construit dynamiquement à chaque appel des trois mots clé CILK. Pour bien comprendre ce graphe il est nécessaire d'assimiler la terminologie CILK. Un programme CILK est composé d'un ensemble de *procédures* qui sont elles-même divisées en *threads*, un thread pour CILK étant une séquence d'instructions se terminant par une des opérations `spawn`, `sync` ou `return`. Un graphe correspondant à Fibonacci pour n=4 est donné sur la figure 3.12.

Les flèches de cette figure représentent les dépendances et sont utilisées par l'ordonnanceur dynamique. En particulier l'exécution d'un thread ne peut commencer que lorsque tous les threads

27. Le site Web du projet CILK se trouve à l'adresse `http://theory.lcs.mit.edu/~cilk/`.

28. Pour aider les programmeurs étourdis, un `sync` est toujours fait avant l'instruction `return` de toute fonction CILK. Cela implique qu'un fils ne peut pas survivre à son père.

```
#include <cilk.h>
#include <cilk-lib.h>
#include <stdlib.h>
#include <stdio.h>

cilk int fib(int n)
{
     if (n < 2)
          return (n);
     else {
          int x, y;
          x = spawn fib(n - 1);
          y = spawn fib(n - 2);
          sync;
          return (x + y);
     }
}

cilk int main(int argc, char *argv[])
{
     int n, answer;

     n = atoi(argv[1]);
     answer = spawn fib(n);
     sync;
     printf("Resultat : %d\n",answer);
     return 0;
}
```

FIG. 3.11 - *Le programme parallèle* CILK *calculant le* n^{ime} *élément de la suite de Fibonacci*

dont il dépend sont eux-même terminés. Sur le graphe donné en exemple, le `return` du `fibo(3)` (représenté par un point grisé) ne peut s'effectuer que lorsque les deux procédures qui exécutent `fibo(2)` et `fibo(1)` ont retourné leurs résultats. Le graphe ainsi obtenu est un graphe orienté sans circuit (dag en anglais), son respect n'est cependant pas suffisant pour imposer un ordonnancement fixe des activités car il ne fournit qu'un ordre partiel sur les activités. On prouve qu'il n'est pas possible de construire une bonne politique générale de scheduling pour un dag car il est toujours possible de construire un dag dont l'ordonnancement avec une accélération linéaire aurait une complexité en espace supra-linéaire. Par contre les graphes de dépendance de CILK ont une particularité qui leur permet d'être toujours ordonnancés efficacement, ils sont *stricts*. Cela signifie que les seules dépendances qui quittent un sous-graphe vont vers un parent, ce qui en fait un arbre. En CILK il est sûr que cette condition est toujours respectée car la seule flèche qui peut quitter une procédure correspond à l'instruction `return` et est donc dirigée vers la procédure qui l'a créée, dans le graphe à l'entité de niveau supérieur.

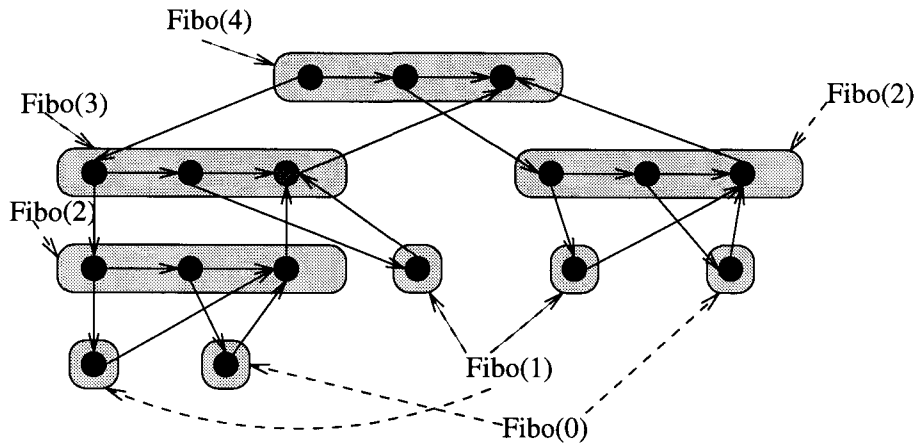


FIG. 3.12 - *Graphe de dépendances de l'application* CILK *calculant Fibo(4). Un rectangle grisé représente une procédure et un point noir un thread. Les flèches descendantes représentent une création d'activités (une opération* **spawn***), les flèches horizontales une continuation d'un bloc d'instructions après une opération structurante et les flèches vers le haut un retour de résultat.*

Le support d'exécution de CILK propose une politique d'ordonnancement, dont l'efficacité a été prouvée, basée sur le concept du vol aléatoire de travail[29]. Le principe est que quand un processeur devient inactif, il va voler des tâches en attente sur un autre processeur[30] choisi au hasard. Chaque processeur exécute séquentiellement les threads qu'il crée, exactement comme dans un programme C traditionnel. Par exemple il exécutera directement un thread créé par un **spawn** en conservant le contexte du thread qui a exécuté l'appel du **spawn** dans une structure de données qui est une pile. De cette manière, quand le fils se termine par un **return**, le contexte de son père est dépilé et son exécution continue. Quand un autre processeur est à cours de travail, il vole ce qui est sur le sommet *opposé* de la pile, c'est-à-dire celui qui aurait été exécuté le plus tard par le processeur cela afin de préserver au maximum la localité des activités pour diminuer les communications.

Cette stratégie d'ordonnancement par vol de travail exécute n'importe quelle application CILK en un temps proche de l'optimal et nous allons justifier cette propriété.

Pour cela il est nécessaire de définir plusieurs valeurs représentant les limites de vitesse que l'exécution d'une application ne peut pas dépasser. Appelons T_p le temps d'exécution d'une application sur p processeurs, le travail correspondant est le temps nécessaire pour exécuter tous les threads du dag. Appelons ce travail T_1 car il correspond au temps de calcul sur un processeur et représente la limite supérieure du temps d'exécution. La première limite sur T_p est que $T_p \geq \frac{T_1}{p}$ ce qui signifie que l'accélération ne peut pas être théoriquement super-linéaire[31]. La seconde limite est basée sur la notion de *taille du chemin miminal* du graphe de dépendances. Elle utilise la valeur T_∞ qui représente le temps d'exécution sur une nombre infini de processeurs. Elle correspond en fait au nombre de niveau du graphe de dépendances, c'est-à-dire à sa profondeur. En effet, il n'est pas possible, même avec un nombre infini de processeurs, d'exécuter un tel ensemble d'activités en moins d'éléments de temps qu'il n'y a de niveaux dans le graphe. On en déduit que la deuxième borne sur T_P est $T_p \geq T_\infty$.

29. *Randomized work stealing* en anglais.
30. On peut rapprocher cette technique des méthodes de régulation de charge dites *receiver initiated.*
31. En pratique cela peut arriver par des phénomènes de cache mais il s'agit d'anomalies.

Les auteurs démontrent que l'ordonnanceur exécute une application CILK en un temps $T_p \leq \frac{T1}{p} + O(T_\infty)$, ce qui est aymptotiquement optimal. Les tests expérimentaux menés par les concepteurs de CILK ont montré que $T_p \approx \frac{T_1}{p} + T_\infty$ était une bonne approximation du temps d'exécution. La complexité en espace de la même application est bornée par la valeur PS_1, S_1 étant l'espace utilisé par la même application sur un processeur. Les preuves de ces bornes sont données dans [BL94], nous ne les détaillerons pas ici. Mentionnons simplement qu'elles utilisent un ordonnancement centralisant la liste des activités en attente pour ensuite étendre les résultats prouvés dans ce contexte à un contexte distribué. Ces résultats sont importants car, en bornant la complexité en temps et en espace de l'exécution, ils permettent de borner le coût de l'exécution.

Le modèle proposé par CILK impose l'utilisation d'un langage donné ce qui est un frein à son utilisation. ATHAPASCAN-1, que nous allons présenter maintenant, a pout ambition de reprendre les idées de CILK dans un cadre plus général.

3.5.2.3 Athapascan-1

Le second système que nous allons étudier dans le cadre de l'ordonnancement dynamique d'applications irrégulières à grain fin a été développé au LMC à Grenoble dans le cadre du projet APACHE[Pla93]. Son nom est ATHAPASCAN-1, la version que nous allons étudier est la seconde, ATHAPASCAN-1b, qui succède à ATHAPASCAN-1a. ATHAPASCAN-1 s'appuie sur la librairie de processus légers ATHAPASCAN-0[Chr96, Gin97] qui lui sert de support exécutif et lui fournit sa portabilité.

Contrairement à CILK, ATHAPASCAN-1 se présente comme une bibliothèque C++ destinée à fournir un modèle de programmation intuitif et générant des applications pouvant être exécutées et, donc ordonnancées, efficacement. Une application ATHAPASCAN-1 se présente classiquement comme un ensemble d'activités créées dynamiquement par un mécanisme identique à celui de l'appel de fonction, comme le **spawn** de CILK. L'idée au centre de ATHAPASCAN-1 est que les communications, et donc les synchronisations, entre activités ont lieu *uniquement au moment du lancement de l'exécution d'une activité.* Pour cela, ATHAPASCAN-1 propose des variables partagées mais sans fournir de mécanisme de cohérence. Le mode d'accès aux variables formant les paramètres de la fonction correspondant à l'activité est précisé par le programmeur dans la déclaration des fonctions, ces modes de passage sont utilisés pour construire le graphe de dépendances des activités qui permet d'ordonnancer l'application.

Illustrons cela par un exemple simple : deux activités A et B effectuent un calcul sur des données indépendantes puis stockent leurs résultats dans une variable partagée. Le résultat final est une accumulation, classiquement une addition, des deux résultats fournis et est ensuite utilisé par une autre activité C. De ce mode d'utilisation des paramètres (le résultat est aussi un paramètre) par chaque activité, on peut en déduire que A et B peuvent être ordonnancées de manière indépendante tandis que l'activité C doit attendre pour démarrer que A et B soient terminées, c'est-à-dire qu'elles aient toutes deux stocké leur résultat dans la variable idoine. En utilisant uniquement le mode de passage des paramètres des activités, il est donc possible de construire le graphe des dépendances de l'application (représenté sur la figure 3.13).

Cette méthode a l'avantage d'être intuitive, en préservant la structuration des applications en fonctions tout en offrant une flexibilité supérieure à celle proposée par CILK puisque des dépendances très fines sur les données sont exprimables sans surcharger l'écriture.

Pour résumer, l'idée au cœur d'ATHAPASCAN-1 est de diriger l'ordonnancement par les accès aux données. L'application est structurée sous la forme d'appels de fonctions sur des variables partagées, la *cohérence* de ces variables, et en particulier les dépendances que cette cohérence implique,

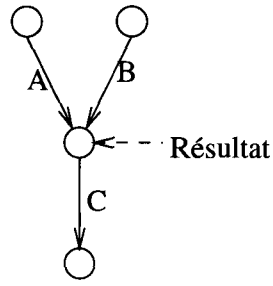


FIG. 3.13 - *Graphe de dépendances sur les variables utilisé pour ordonnancer les activités*

guidant l'ordonnanceur dans ses choix. L'expression de cette cohérence est simplement faite dans la déclaration des paramètres de la fonction formant l'activité. Chaque type de cohérence (lecture seule, écriture seule, ...) correspond à une classe générique C++ rendant la déclaration intuitive. En plus de ces variables partagées utilisées pour calculer l'ordonnancement, chaque activité a ses propres variables locales qui ne sont pas partagées et ne nécessitent donc pas de description de mode d'accès.

Exemple d'utilisation Nous allons prendre pour illustrer un exemple un peu plus compliqué que le précédent : le produit scalaire de deux vecteurs. Sa parallélisation consiste à décomposer les vecteurs en sous-vecteurs puis à calculer séparément le produit scalaire pour chaque sous-vecteur, le résultat global étant stocké dans une variable partagée. La cohérence des variables impliquées est de deux types : lecture seule pour les vecteurs et écriture par accumulation pour la variable résultat. Ces deux classes de cohérence s'appellent `a1_shared_r<type>` et `a1_shared_cw<type>`. La déclaration de l'activité effectuant un produit scalaire sur un sous-vecteur (ou un vecteur) s'écrit :

```
struct produit_scal : public a1_task {
  void operator() ( a1_shared_r<vector<type>> a, a1_shared_r<vector<type>> b,
                    a1_shared_cw<type> c )
    {
      // corps de la fonction effectuant le produit scalaire
    }
}
```

Détaillons un peu cette portion de code. Tout d'abord, ATHAPASCAN-1 étant une librairie C++, une activité ATHAPASCAN-1 est une classe qui hérite du type `ath1_task`. Cette classe ne contient qu'une méthode, l'appel de fonction, car elle correspond en fait à la fonction qui sera exécutée par l'activité[32]. Le point important à noter réside dans les paramètres dont le mode de passage est précisé en utilisant des classes génériques spécifiques, `a1_shared_r` pour les variables en lecture seule par exemple. Pour créer cette tâche il faut appeler la fonction `a1_new_task` avec comme paramètres le nom de la fonction (de la classe, comme nous l'avons vu) et les paramètres effectifs de l'appel. Pour la fonction précédente cela donne pour un calcul sur des sous-vecteurs de `double` A et B en stockant le résultat dans une variable C :

```
a1_new_task(produit_scal()<double>,A,B,C);
```

32. Cette façon, un peu déroutante, de déclarer les fonctions est standard dans les librairies C++.

à comparer avec ce qu'aurait été un appel séquentiel classique à une fonction identique :

```
produit_scal<double>(A,B,C);
```

Pour utiliser cette activité il faut traditionnellement utiliser ce que la communauté des chercheurs en parallélisme de données appelle une boucle de virtualisation, c'est-à-dire une boucle scindant les vecteurs dont on veut calculer le produit scalaire en sous-vecteurs. Ces derniers seront ensuite les paramètres d'un ensemble de création d'activités du type `produit_scal` qui effectuera les calculs en stockant leur résultat par accumulation dans une variable commune. ATHAPASCAN-1 fournit aussi un mécanisme permettant de diviser automatiquement un vecteur en sous-vecteurs[33] ce qui facilite la tâche du programmeur en lui évitant d'écrire cette opération fastidieuse.

Comme on le voit sur cet exemple, l'écriture d'une application ATHAPASCAN-1 est assez simple pour quelqu'un connaissant déjà C++. La difficulté majeure est d'assimiler le concept des classes de paramètres utilisées comme désignation du mode de partage des données.

Ordonnancement des activités On peut rapprocher ATHAPASCAN-1 du modèle théorique de l'algorithmique PRAM qui est synchrone. L'horloge globale servant de synchronisation dans ce modèle est remplacée dans le cadre distribué d'ATHAPASCAN-1 par les synchronisations sur les accès aux variables partagées, le type de synchronisation étant exprimé par le mode d'accès aux variables. ATHAPASCAN-1 constitue une adaptation aux architectures distribuées (donc globalement asynchrones) du modèle synchrone PRAM.

Au centre du mécanisme d'ordonnancement d'ATHAPASCAN-1 on trouve la notion d'ordre séquentiel des exécutions : en remplaçant tous les appels à `a1_new_task` par les appels de fonction correspondants on obtient une exécution séquentielle de l'application distribuée. Cette exécution définit l'ordre séquentiel des créations d'activités. L'ordonnancement doit respecter cet ordre séquentiel dans un contexte parallèle et distribué. Notons que comme l'exécution séquentielle représente un ordonnancement tout à fait valide de l'application, l'ordonnanceur peut à tout moment adapter le grain de parallélisme en passant en mode séquentiel, c'est-à-dire en interprétant toutes les créations de tâches comme des exécutions séquentielles, donc des appels synchrones et locaux de procédure.

Comme nous l'avons évoqué dans les exemples, le mode d'accès aux variables, donc les dépendances sur les données entre les activités, définit les relations de précédence, et donc le graphe de dépendances, mais il définit aussi les contraintes de localité. Une variable dont le mode d'accès pour une activité est lecture seule pourra être dupliquée entre les activités l'utilisant de cette manière. La duplication est bien évidemment possible car le respect des dépendances entre activités implique qu'au lancement de l'activité la valeur se trouvant dans la variable est la bonne. Cette possibilité de duplication permet une localisation de l'activité indépendante de la localisation de la donnéee, cet éloignement ne nécessitant qu'une communication au début de l'activité. Par contre si le mode d'accès de cette variable était lecture/écriture, les localiser sur des nœuds différents risque d'impliquer un nombre de communications important, éviter de les éloigner ajoute une contrainte de localité supplémentaire à l'ordonnancement.

L'indépendance est complète entre la description de l'application et l'ordonnanceur utilisé pour l'exécuter. Nous avons parlé dans les paragraphes précédents d'un ordonnanceur assurant une exécution séquentielle, d'autres, prouvés optimaux comme celui de CILK par exemple, ou encore un

33. On peut rapprocher cette opération du mécanisme de clonage léger qui agit sur les activités alors que celui présenté agit sur les données.

ordonnanceur basé sur le partage de charge[34] sont aussi utilisables. En d'autres termes, le module gérant le graphe de précédence, et donc celui qui s'assure que telle activité est prête à être exécutée alors qu'une autre doit attendre des résultats, est séparé de celui qui va prendre les décision de placement telles que le choix du processeur sur lequel une activité prête sera exécutée.

ATHAPASCAN-1 offre donc un environnement de programmation flexible basé sur une bibliothèque qui étend le langage C++ en y ajoutant la création d'activités et la possibilité de préciser des variables partagées dont le mode d'accès est spécifiable. L'ordonnanceur dynamique élément de ATHAPASCAN-1 utilise ces informations pour placer les activités afin de maximiser l'occupation des processeurs tout en minimisant les communications. Il garantit de plus le respect de l'ordre d'exécution des activités définis par l'exécution séquentielle. La politique de placement peut être, par contre, redéfinie pour des applications particulières.

3.5.3 Discussion sur les ordonnanceurs à grain fin

Nous avons étudié dans cette section trois approches différentes pour l'ordonnancement des applications à grain fin. Trois critères principaux les caractérisent :

Granularité des dépendances sur les données Ils utilisent tous les dépendances sur les données pour calculer l'ordonnancement des activités mais avec une granularité différente.

Capacité de paramétrage Certains environnements fournissent des briques de base pour définir des politiques d'ordonnancement *ad-hoc* d'autres imposent une politique correspondant exactement au modèle de conception imposée par l'environnement.

Fonctionnement réactif Deux types de politiques d'ordonnancement se dégagent : celles qui réagissent quand elles constatent une anomalie dans le comportement de l'application et celles qui utilisent les informations déjà connues, ou extraites de connaissances antérieures glanées à la compilation par exemple, pour calculer un ordonnancement dont il est alors possible dans certains cas de prouver la quasi optimalité.

Nous allons détailler ces trois points en précisant pour chacun où se situent les systèmes étudiés. Nous avons illustré sur la figure 3.14 leur positionnement dans un espace défini par ces trois axes.

Granularité des dépendances prises en compte Les trois systèmes que nous avons étudiés utilisent, de manière directe ou indirecte, le graphe de dépendances sur les données pour ordonnancer les activités. ATHAPASCAN-1 et CILK le construisent, au moins partiellement, afin de calculer l'ordonnancement optimal alors que MILLIPEDE réagit dynamiquement aux accès faits aux données sans le construire. La différence entre les politiques se fait sur la granularité des dépendances sur les données qu'ils prennent en compte :

La page MILLIPEDE est basé sur une mémoire virtuellement partagée, il utilise logiquement la page mémoire comme unité de partage d'ordonnancement.

Une variable CILK utilise le résultat des appels de fonction pour exprimer les dépendances entre les activités. On peut rapprocher cette idée du modèle *data-flow* qui conditionne aussi le départ, ou la continuation, d'une activité à la présence d'une ou plusieurs données. Cette simplicité dans l'expression des dépendances permet d'utiliser un modèle théorique assez simple pour trouver un algorithme démontré optimal d'ordonnancement.

34. Appelé *sender-initiated* dans l'équilibrage de charge.

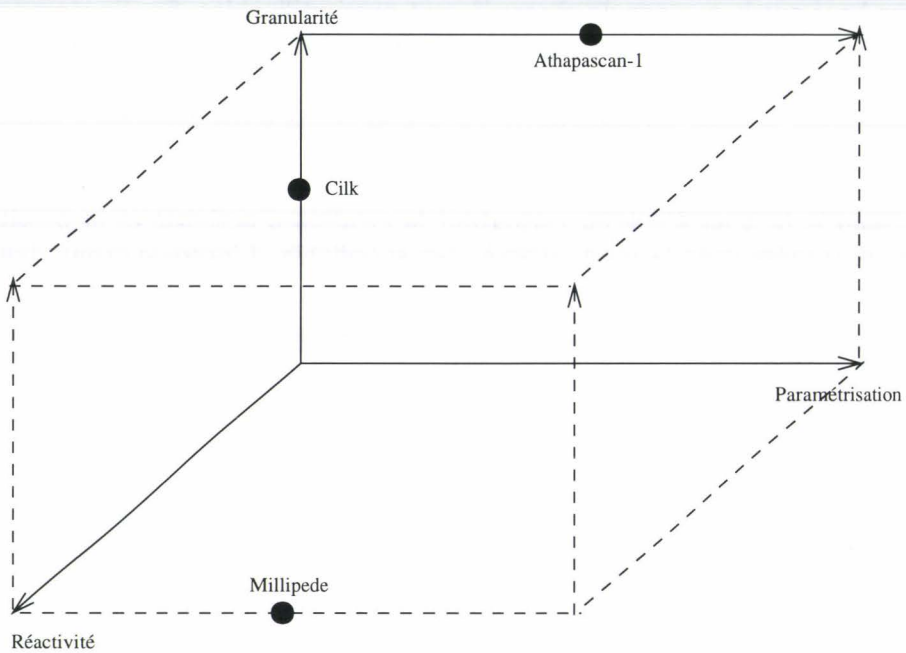


FIG. 3.14 - *Positionnement des environnements étudiés sur les trois axes {Réactivité, Granularité, Paramétrisation}*

Des variables ATHAPASCAN-1 utilise pour le calcul de l'ordonnancement le mode d'accès d'un nombre quelconque de variables partagées. Il s'agit du modèle le plus souple et le plus puissant car il est peu restrictif pour le programmeur tout en permettant la conception d'algorithmes d'ordonnancement sophistiqués.

ATHAPASCAN-1 et CILK proposent la granularité la plus fine de tous les environnements car ils prennent en compte les dépendances entre les variables. Au contraire MILLIPEDE gère des dépendances assez grosses puisqu'elles interviennent seulement au niveau de la page. Notons qu'en allouant une variable par page mémoire, la granularité de l'ordonnancement gérée par MILLIPEDE peut être aussi fine que celle de CILK mais avec un gâchis de place mémoire assez conséquent.

Capacités de paramétrage CILK et ATHAPASCAN-1 utilisent tous deux un ordonnancement non préemptif, c'est-à-dire qu'ils ne reviennent jamais sur leurs décisions de placement. Cela n'est pas gênant dans le cas de CILK car il s'adresse à un cadre plus restreint d'applications qu'ATHAPASCAN-1. Il est donc possible pour eux de prouver l'optimalité de leur algorithme d'ordonnancement. Dans le cadre, plus général, visé par ATHAPASCAN-1 cela peut être un problème[35]. Il est clair qu'en utilisant ATHAPASCAN-1 dans le même cadre que CILK, il est possible d'utiliser les mêmes algorithmes d'ordonnancement que ce dernier, algorithmes ou ordonnancements qui ont été prouvés optimaux.

Dans MILLIPEDE par contre il n'est possible de paramétrer que la partie prise de décision de la politique d'ordonnancement. Quand une décision a été prise son exécution et en particulier les éléments, les activités, sur lesquels elle va s'appliquer sont choisis par le support d'exécution. Le

35. L'équipe d'ATHAPASCAN-1 envisage d'ailleurs d'intégrer le concept de migration préemptive dans les prochaines versions d'ATHAPASCAN-1.

moment de la décision, et donc les critères de déséquilibre, peuvent être dépendants de l'application alors que l'action entreprise pour y remédier est dépendante du support d'exécution. Cette séparation peut poser problème ce qui a poussé les concepteurs à intégrer un mécanisme visant à éliminer de manière transparente le plus pénalisant d'entre eux, l'effet ping-pong.

L'architecture de ces systèmes met en évidence la difficulté de trouver une approche offrant un bon compromis entre généralité et efficacité, les approches garantissant un ordonnancement optimal ne sont que très peu modifiables. CILK est une illustration parfaite de cette difficulté, en utilisant ce langage l'utilisateur accepte d'être ordonnancé d'une manière précise et non modifiable[36]. ATHAPASCAN-1 offre le meilleur compromis entre les deux approches car il laisse la capacité au programmeur de choisir plusieurs politiques d'ordonnancement ou de définir la sienne tout en en proposant plusieurs dont l'optimalité est prouvée.

Réactivité de l'ordonnancement ATHAPASCAN-1 et CILK proposent tous les deux une approche d'ordonnancement sans rétroaction, quand une décision de placement est prise elle est irrévocable. Dans un cadre strict de conception où la date du début d'exécution d'un activité ne doit respecter que l'ordre séquentiel cela n'est pas gênant car il est toujours possible de prévoir, et donc d'anticiper, un déséquilibre. Dans un cadre irrégulier où le déséquilibre peut apparaître à tout moment cela est plus gênant. La réponse à ce problème dépend du but poursuivi par l'ordonnancement : s'il s'agit de garder tous les processeurs actifs en plaçant les activités dessus de façon à minimiser les communications, il est possible de réaliser un ordonnancement satisfaisant quand la granularité applicative est suffisamment fine pour que les activités soient nombreuses. Dans le cas d'ATHAPASCAN-1 où le support contrôle lui-même la granularité et le degré de parallélisme de l'application, cette approche est possible. Par contre si le degré de parallélisme de l'application est important et que le nombre d'activités exécutées *simultanément* est élevé, alors une politique d'ordonnancement qui ne serait pas réactive montre des limites pour une applications irrégulière.

Cela s'explique par le fait que ATHAPASCAN-1 et MILLIPEDE réagissent à un déséquilibre dans la répartition des activités sur les processeurs en lançant l'exécution de certaines qui sont « en attente», ou s'il n'y en a pas, en modifiant la granularité pour en créer. Dans le cas d'une application où aucune activité n'est en attente et où la granularité dépend de l'application, ils ne peuvent pas réagir à un tel déséquilibre. Une politique utilisant la migration pourra au contraire toujours réagir à un déséquilibre.

La réactivité n'est une caractéristique importante de l'ordonnancement que pour des applications hautement irrégulières dont la granularité et le degré de parallélisme sont importants. Si ces deux valeurs sont manipulables par l'ordonnanceur ou si le coût des activités peut être prédit avec exactitude elle n'est pas nécessaire.

3.6 Conclusion

Dans ce chapitre nous avons présenté les environnements offrant des activités ayant une fine granularité de calcul. Nous avons d'abord montré comment il est possible de programmer les applications hautement irrégulières de type Branch&Bound dans un tel cadre. Ensuite nous avons étudié la façon dont les processus légers s'intégraient dans un cadre conceptuel visant la facilité de programmmation et l'efficacité à l'exécution. Nous avons présenté les approches principales proposées dans la littérature pour atteindre ce but puis celle que nous proposons basée sur la virtualisation

36. Notons toutefois que les versions les plus récentes de CILK propose une mode d'exécution dit « *randomized* » (aléatoire) dont les auteurs préconisent l'usage pour le déverminage et qui ajoute une part d'indéterminisme dans l'exécution d'une application.

de l'architecture. Nous avons détaillé ce concept en insistant sur ses deux aspects : conceptuel et exécutif.

L'aspect conceptuel consiste à fournir au programmeur le moyen de découper son applications en activités. Nous proposons deux mécanismes pour cela : l'invocation distante de service et le clonage léger. L'aspect exécutif consiste à gérer les activités ainsi générées pour une exécution efficace de l'application. Le mécanisme que nous fournissions dans ce but est la migration d'activités. L'utilisation de ce mécanisme durant l'exécution nécessite la mise en place d'une politique d'ordonnancement dynamique de l'application. Nous avons étudié plusieurs systèmes proposant un tel ordonnancement d'activités de fine granularité en isolant leurs spécificités.

Nous allons nous focaliser dans le chapitre suivant sur cet aspect, l'ordonnancement, en présentant notre proposition pour un « virtualisateur » générique, largement paramétrable, destinée aux applications hautement irrégulières appartenant au domaine de l'optimisation combinatoire.

Chapitre 4

Ordonnancement des applications hautement irrégulières à grain fin : l'approche GTLB

L'exécution des applications irrégulières dans un contexte parallèle à grain fin a pour caractéristique principale d'avoir un comportement hautement imprévisible. La conséquence première de cette caractéristique est le déséquilibre qui apparaît entre le nombre d'activités présentes sur les processeurs de l'architecture parallèle. L'évolution de cette charge est totalement imprévisible, ce qui exclut toute tentative d'exécution efficace qui ne ferait qu'un ordonnancement par anticipation, c'est-à-dire sans prendre en compte immédiatement durant l'exécution la dynamique d'évolution de l'application.

La virtualisation de cette classe d'applications implique donc la prise en compte de facteurs dynamiques qui remettent perpétuellement en cause les décisions de localisation déjà prises. Il est donc crucial pour une bonne efficacité de disposer d'un mécanisme dynamique d'ordonnancement. Les critères sur lesquels ces décisions de placement sont prises étant particuliers à chaque application, ou à chaque classe d'applications, il n'est pas possible de proposer un ordonnanceur efficace dans tous les cas.

L'approche que nous proposons est celle d'un ordonnanceur « générique » qui peut être paramétré afin d'être adapté à une (ou une classe de) application particulière. Nous allons, dans ce chapitre, décrire notre proposition.

Nous commençons par décrire en quoi l'ordonnancement des applications hautement irrégulières est particulier spécialement par son aspect spéculatif. Nous en déduirons le cahier de charge auquel doit répondre un ordonnanceur générique pour cette classe d'applications. Nous étudierons ensuite pourquoi la migration offre de nouvelles possibilités par rapport aux politiques d'ordonnancement « traditionnelles ». Pour cela nous comparerons des politiques d'ordonnancement utilisant ou non la migration. Nous présenterons ensuite notre approche qui est basée sur la migration et la notion d'ordonnancement continue à coût bornable. Nous illustrerons son utilisation pour construire un ordonnancement respectant un critère simple, l'équilibre en terme de nombre d'activités sur les processeurs.

4.1 Ordonnancement des applications hautement irrégulières

Dans cette section, nous allons étudier la manière dont les applications hautement irrégulières doivent être virtualisées. Nous commencerons par détailler leurs particularités, ce qui nous amènera à préciser les contraintes, particulièrement sur le support d'exécution, que ces spécificités imposent.

Nous n'étudierons ici que les aspects liés à l'exécution car les aspects structurants, la manière de décomposer une application en activités indépendantes, ne diffèrent pas du cas général qui a déjà été vu dans le chapitre précédent. L'opérateur de décomposition que nous utiliserons dans tout ce chapitre est l'invocation distante de services.

4.1.1 Particularités des applications hautement irrégulières

Dans cette section nous allons étudier les conséquences des caractéristiques des applications hautement irrégulières sur les politiques d'ordonnancement à mettre en place pour une exécution efficace. Nous en dégageons trois principales : l'absence de dépendances utilisables pour calculer l'ordonnancement, le rôle de la politique d'ordonnancement dans la quantité de travail totale à exécuter et le degré de parallélisme réel dépendant de l'application.

4.1.1.1 Absence de dépendances exploitables pour l'ordonnancement

Nous avons vu dans l'introduction que la particularité première de la classe des applications hautement irrégulières réside dans l'absence d'un graphe de dépendances entre les tâches utilisables pour l'ordonnancement. Ce graphe existe mais il ne peut pas être prévu, il n'est connu qu'*a posteriori*. Cela est dû au haut niveau d'indéterminisme de l'application qui empêche de déduire de son comportement passé une évolution de façon complètement fiable.

Le graphe de dépendances des activités est donc à la fois imprévisible et très pauvre en informations exploitables pour l'ordonnancement. Chaque activité, une fois créée, ne se synchronise plus avec son créateur ou avec une autre. La seule exception à cette règle est représentée par les variables partagées sans politique de cohérence fixée qui ne sont, de ce fait, pas utilisables pour extraire des dépendances.

Cette absence d'informations sur les dépendances est un point très important car il interdit l'utilisation directe des politiques d'ordonnancement étudiées dans le chapitre précédent. En effet, elles utilisent toutes les dépendances sur les données comme critère principal pour l'ordonnancement, même si la granularité des dépendances prises en compte varie suivant les ordonnanceurs. Dans le cadre des applications qui nous intéressent, les caractéristiques applicatives utilisées pour calculer l'ordonnancement ne sont plus liées aux données manipulées. Au contraire elles sont liées aux activités, ce qui change complètement l'approche de conception des ordonnanceurs comme nous le verrons dans la section 4.1.2.

Nous sommes ici dans le cadre le plus « pur » de la virtualisation car l'ordonnanceur ne travaille que sur les activités sans utiliser à aucun moment quelque connaissance que ce soit sur les données. Son travail consiste donc à placer les activités sur les processeurs selon des critères qui ne sont pas dépendants des données. Par conséquent, aucune interaction entre l'ordonnanceur et l'application n'est nécessaire au moment de la conception de l'application. Ce n'était pas le cas dans les approches d'ordonnancement vues précédemment où une distinction était toujours faite entre les données qualifiées de locales et qui ne concernaient pas l'ordonnanceur, et les données globales (appelées partagées dans le cas d'ATHAPASCAN-1 ou de MILLIPEDE) qui elles rentrent en compte dans le calcul de l'ordonnancement. Bien que ces deux formes de variables soient les mêmes au niveau modèle de conception, elles sont traitées différemment d'un point de vue sémantique. Le **seul** élément de

l'application sur lequel la politique d'ordonnancement peut à la fois obtenir des informations et influencer le placement (ou la création) est donc les **activités**. Par conséquent il est important qu'elle dispose de fonctionnalités avancées (telles que la migration) sur ces dernières.

Pour les applications que nous visons, les seuls éléments qui nous intéressent sont les activités, leurs traitements sur les données étant sous leur seule responsabilité. La conséquence sur la politique d'ordonnancement, et donc sur les capacités sous-jacentes du support sur lequel cette dernière va être implantée, est qu'elle ne doit être consciente que d'un seul type d'élément applicatif, les activités. De plus, l'ordonnanceur doit en permanence remettre en cause ses choix car il ne dispose pas d'informations *a priori* sur le comportement de l'application, il ne peut que réagir *a posterori* aux observations qu'il fait sur ce comportement.

4.1.1.2 Influence de l'ordonnancement sur la quantité de travail

Une particularité très importante dans l'appréhension de l'ordonnancement de cette classe d'applications réside dans le fait que **la politique d'ordonnancement influe sur la quantité totale de travail accomplie pour résoudre le problème.** Nous allons détailler cette particularité et nous verrons dans la section suivante son incidence sur les politiques d'ordonnancement à mettre en œuvre.

Les applications de faible ou moyenne irrégularité sont *à travail constant* ce qui signifie que la quantité de calcul est la même quelle que soit la politique d'ordonnancement choisie. Le but d'un ordonnanceur dans ce cadre est donc de maintenir tous les processeurs actifs à tout moment en effectuant une bonne répartition des activités sur les processeurs et en prenant en compte la localité des données partagées afin de diminuer le coût des communications. Ce n'est pas le cas des applications que nous visons car l'ordre dans lequel les activités sont exécutées[1] a une influence sur le coût total de l'application.

Détaillons un peu les raisons de cette particularité. Nous avons vu dans le chapitre 2 l'influence du respect des priorités entre les évaluations des sous-arbres dans la réduction du temps pris par la résolution du problème. Dans le cadre d'une application à fine granularité de calcul, cela se traduit par la définition d'une *relation d'ordre* entre les activités, le respect de cette relation conditionnant la quantité de travail totale pour la résolution du problème. Cette particularité est la conséquence du caractère ***spéculatif*** de l'exploration de l'arbre de recherche. Cela change complètement les caractéristiques de l'ordonnancement en lui ajoutant une dimension dynamique supplémentaire assez forte. En effet si une nouvelle activité est créée avec une priorité supérieure à toutes les autres, elle doit être prise en compte par l'ordonnanceur, c'est-à-dire exécutée le plus tôt possible après sa création quitte pour cela à arrêter une (ou des) autres. ATHAPASCAN-1 permet à l'application de fixer une valeur de *priorité* au moment de la création d'une activité, cette valeur est utilisée pour déterminer la date de début d'exécution de l'activité en la faisant passer avant d'autres. Cela constitue un premier pas mais est néanmoins insuffisant car il n'est pas couplé à un ordonnanceur préemptif[2].

Une prise en compte immédiate, au moment de sa création, d'une activité prioritaire oblige à pouvoir remettre en cause à tout moment les décisions de placement déjà prises. Cette remise en cause est destinée à trouver un nouveau placement des activités qui soit optimal à la fois en quantité, c'est-à-dire gardant tous les processeurs actifs, mais aussi en qualité, en leur faisant exécuter les activités les « plus intéressantes ». Cependant, comme nous sommes dans le cas des applications

1. Ainsi que la manière dont l'application est divisée en activités d'ailleurs ... Nous en avons déjà parlé dans le chapitre 2 et nous y reviendrons dans le chapitre 5.

2. En tout cas dans les versions actuelles d'ATHAPASCAN-1 ...

basées sur des heuristiques, les priorités de chaque activité ne sont donc pas forcément « sûres », elles peuvent être contredites par la suite. Une activité qui semblait, à partir des informations partielles disponibles sur elle à sa création, très importante peut s'avérer finalement sans grande importance. Dans cette optique il faut que l'ordonnancement conserve la notion de concurrence de la virtualisation afin de garantir que *toutes les activités avanceront simultanément* bien que ce soit à des vitesses différentes. L'affectation de priorités aux activités ajoute simplement une dimension « spéculative » à l'ordonnancement.

Ordonnancer dans ce cadre est donc plus compliqué que de garder les processeurs actifs avec un mécanisme de préemption pour éventuellement arrêter totalement une activité afin de la remplacer par une nouvelle qui a été jugée plus prioritaire. Un exemple est donné sur la figure 4.1 qui représente un ordonnancement qui garde tous les processeurs actifs mais qui prend en compte non pas leur importance, mais uniquement leur date de création comme critère d'ordonnancement. Il s'agit par exemple du comportement de l'ordonnanceur de CILK si l'on suppose que toutes les activités sont prêtes à être exécutées dès leur création.

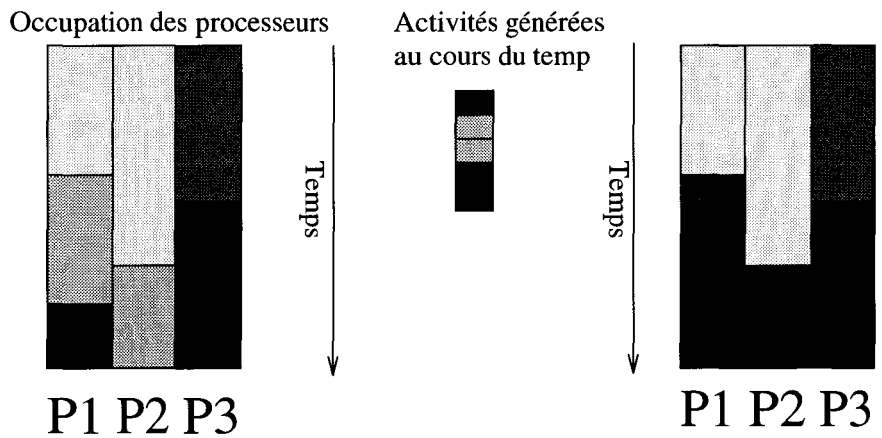


FIG. 4.1 - *Exemple d'ordonnancement gardant tous les processeurs actifs. À gauche sans prendre en compte l'importance relative de chaque activité par rapport aux autres, à droite en en tenant compte. Plus les activités sont importantes pour l'application plus elles sont sombres. Sur le schéma de droite les activités les plus prioritaires sont exécutées avant les moins prioritaires ce qui amène une occupation des processeurs par les activités différente. Nous ne présentons ici que le début de l'ordonnancement et donc pas la totalité des activités.*

L'ordonnancement doit satisfaire une contrainte supplémentaire qui est de garder les processeurs actifs et travaillant « principalement » sur *les activités les plus intéressantes* de l'application sans cependant négliger totalement les autres, moins prioritaires. Ce critère supplémentaire à prendre en compte augmente le coût de l'ordonnancement, un compromis est donc à trouver entre ce surcoût et le gain que l'ordonnancement procure. Un ordonnancement qui conserverait l'ensemble des processeurs actifs sur le travail le plus prioritaire mais dont le coût serait le double d'une politique laissant parfois des processeurs inactifs n'est probablement pas le plus adapté.

Une conséquence de ce caractère spéculatif de l'ordonnancement est qu'il est impossible d'en déterminer un optimal. Plus exactement, il faudrait essayer l'ensemble des ordonnancements possibles pour l'ensemble des activités. En effet, comme la création des activités est conditionnée aux décisions passées de l'ordonnanceur, on comprend que le seul moyen pour déterminer l'ordonnancement

optimal pour *un* jeu de données consiste à évaluer les performances de chacun en les essayant tous. Cette méthode est clairement impraticable, l'ordonnancement optimal n'est donc pas calculable.

4.1.1.3 Degré de parallélisme dépendant de l'application

Dans notre modèle, une application parallèle est composée d'activités qui n'ont pas de dépendances entre elles. Le degré réel de parallélisme intra-application, le nombre d'activités s'exécutant concurremment dans l'application, est borné par l'ordonnanceur. Dans le cas d'ordonnanceurs utilisant les dépendances directes entre les données il est, généralement, peu élevé. Quand une activité se termine, le processeur sur lequel elle s'exécutait va en chercher une autre dans un *pool* d'activités prêtes en attente d'exécution. La gestion de ce *pool* est conditionnée par la politique d'ordonnancement en particulier l'affectation d'une nouvelle activité à un processeur peut être faite en fonction de critères de localité. Parfois, afin d'utiliser les capacités de recouvrement calcul/communication des processus légers, plusieurs activités sont gérées au lieu d'une mais ce choix dépend du support d'exécution pas de l'application. Le degré réel de parallélisme est donc de un (ou d'une constante) multiplié par le nombre de processeurs, il n'est pas dépendant des caractéristiques de l'application mais de l'architecture parallèle et du support d'exécution.

Les applications qui nous intéressent font de l'exploration *spéculative* d'espaces de recherche, le degré de parallélisme qui conditionne le nombre de sous-espaces exploitables simultanément doit donc être fixé **uniquement par l'application**. En effet, la raison pour laquelle une activité décide d'en créer une nouvelle peut être qu'elle souhaite que les deux continuent concurremment leur exécution car elles correspondent à deux sous-espaces dont l'exploration doit se faire en parallèle. En limitant, pour des raisons techniques, le degré de parallélisme réel de l'application on la prive de cette possibilité.

La méthode qui consisterait à stocker ces créations d'activités en attendant qu'une autre se termine pour la remplacer est à rapprocher du stockage d'activités en attente d'exécution des approches étudiées dans le chapitre 2. Rappelons que ces approches étaient utilisées pour marier une granularité applicative fine et une grosse granularité exécutive. L'un des avantages du parallélisme à grain fin est justement de pouvoir s'affranchir de cette limite, il ne faut pas que l'ordonnanceur l'impose de nouveau.

Pourquoi ce degré de parallélisme est-il important dans le temps de résolution? Comme nous l'avons dit, l'exploration de l'espace de recherche se fait suivant une méthode spéculative, il est impossible de prévoir la pertinence des choix faits. Il est donc important de poursuivre en parallèle plusieurs chemins en leur donnant éventuellement des priorités différentes pour privilégier ceux qui ont été évalués comme, *a priori*, plus intéressants. Dans la famille d'environnements que nous utilisons, cela se traduit par augmenter le degré de parallélisme réel de l'application, il est donc important de lui laisser cette capacité.

Le but de ce haut degré de parallélisme est de trouver une bonne solution tôt dans l'exécution. Cela est important pour réduire la taille de l'espace de recherche comme nous l'avons vu mais cela l'est aussi dans le cas où on cherche une bonne solution, pas forcément optimale, dans un délai « raisonnable »[3]. C'est dans ce but que sont utilisées les méthodes utilisant des méta-heuristiques. Nous pensons qu'une méthode visant à prouver l'optimalité, utilisée avec une contrainte temporelle, qui explorerait simultanément différentes parties de l'espace de recherche et qui utiliserait des priorités pourrait aussi être compétitive dans ce cadre.

3. Raisonnable par rapport à des temps d'exécution qui peuvent dépasser la semaine, voire même l'année, pour prouver l'optimalité.

4.1.2 Conséquences sur les caractéristiques d'un ordonnanceur générique

Les trois points précédents ont mis en évidence qu'une politique d'ordonnancement pour la classe d'applications que nous visons ne pouvait pas utiliser de dépendances sur les données comme critère d'informations. De plus nous avons vu que le respect de l'importance respective des activités les unes par rapport aux autres jouait un rôle important dans les performances globales de l'application en réduisant la quantité totale de travail à accomplir pour résoudre le problème donné. Enfin, l'ordonnanceur ne doit pas limiter le degré de parallélisme de l'application mais doit au contraire le prendre en compte.

Les deux premières caractéristiques sont *a priori* contradictoires car elle semble indiquer que l'on ne dispose que de très peu de données pour calculer un ordonnancement qui doit connaître intimement les caractéristiques de l'application pour l'ordonnancer afin de minimiser la quantité de travail qu'elle représente. Pour corser le tout, l'application étant irrégulière et l'ordonnancement entièrement dynamique, il sera perpétuellement remis en cause à chaque création, ou terminaison, d'activités.

Le moyen utilisé actuellement pour respecter ces critères consiste en la conception d'une politique qui est très étroitement liée à l'application et qui sera donc peu générique. Cela nous éloigne du but premier de la virtualisation qui est justement de fournir un environnement qui soit le plus général possible tout en restant efficace. Il est donc nécessaire de trouver un compromis entre ces deux contraintes. Nous allons expliquer dans la suite de ce chapitre les éléments nécessaires et les choix à faire la réalisation de ce compromis.

La solution idéale serait de pouvoir définir un ensemble de politiques, chacune étant destinée à une sous-classe d'applications hautement irrégulières et pouvant cependant être adaptée assez finement aux besoins spécifiques de chaque élément de cette sous-classe. Un ordonnanceur destiné à cette classe d'applications doit donc être paramétrable sur deux niveaux :

1. **Critère d'ordonnancement** Il doit fournir un support générique sur lequel différentes politiques peuvent être aisément construites.

2. **Paramètrage dynamique de l'ordonnancement** Une fois la politique définie elle doit pouvoir être adaptée dynamiquement au comportement de l'application. Cette adaptation est entièrement dynamique car le haut degré d'irrégularité des applications visées exclut un paramétrage non évolutif.

Le deuxième point est nécessaire car le comportement d'une application travaillant sur des données différentes mais de même taille peut voir son temps d'exécution varier considérablement, d'un facteur d'un million par exemple (exemple vu sur le problème de l'affectation quadratique). Le comportement de l'ordonnanceur doit bien entendu être différent pour ces deux jeux de données. L'interface de paramétrage de l'ordonnanceur générique, qui est indépendante de la politique choisie, doit donc être assez souple pour pouvoir s'adapter sans difficulté aux besoins des applications.

L'ordonnanceur générique doit en conséquence disposer d'une interface entre la politique et les applications. Cette interface sera composée de deux aspects, l'un lié à la prise en compte des spécificités de l'application (la politique d'ordonnancement définie) et l'autre permettant à l'application de paramétrer le fonctionnement de l'ordonnanceur pour l'adapter finement à ses besoins (le paramétrage de la politique). Il est donc nécessaire d'avoir une interaction dans les deux sens :

Ordonnanceur → Application L'ordonnanceur agit sur l'application en déplaçant ses activités bien sûr mais il peut aussi agir sur d'autres points. Il peut par exemple influer sur la

création des activités, donc affecter la granularité applicative, afin de rentabiliser au mieux l'architecture parallèle.

Application → Ordonnanceur L'application influe sur la politique d'ordonnnancement en précisant l'importance relative des activités et en agissant sur d'autres paramètres, tel que le coût maximal de l'ordonnancement qu'elle tolère par exemple, pour les adapter à ses besoins.

L'interface de l'ordonnanceur vers l'application est mise en œuvre par une méthode préemptive, c'est-à-dire sans intervention de l'application, alors que les décisions de l'ordonnanceur sur la granularité nécessitent une coopération avec l'application qui se traduit par des appels explicites à une interface.

Les interactions dans le sens application vers ordonnanceur nécessitent elles un protocole de communication, donc une interface explicite, par le biais de variables ou de fonctions, cette interface pouvant éventuellement être extensible.

Dans la mesure où les applications que nous visons sont trop irrégulières pour être ordonnancées sur leurs données, le seul facteur propre à l'application sur lequel l'ordonnancement influe est les activités. Cela nous différencie des ordonnancements de ATHAPASCAN-1 ou MILLIPEDE où les connaissances sur l'accès aux données étaient utilisées à la fois pour calculer l'ordonnancement mais aussi pour l'influencer, par le biais du mouvement des pages mémoire en MILLIPEDE et par celui de la réplication permise par la connaissance du mode d'accès en ATHAPASCAN-1. Cette caractéristique implique que l'ordonnanceur doit disposer, pour être efficace, d'opérations agissant directement sur les activités sans être intrusives donc en restant transparentes pour l'application. La plus importante, après la création qui est dépendante de la granularité donc intimement liée à l'application, est bien sûr la migration. C'est la brique de base qui permet de construire un ordonnanceur agissant sur les activités sans perturber l'application. Elle est *indispensable* pour construire un ordonnanceur dynamique générique qui n'influe pas sur le degré de parallélisme applicatif.

Notre approche de l'ordonnancement comme élément permettant la virtualisation totale des applications hautement irrégulières a donc les exigences suivantes :

Transparence à la conception et à l'exécution pour l'application, la seule exception étant un paramétrage plus fin que l'application peut modifier facilement si elle désire l'adapter finement à ses besoins.

Flexibilité Bien qu'offrant un modèle générique fournissant de bonnes performances dans un cadre général, l'ordonnanceur doit être adaptable à d'autres critères d'ordonnancement que ceux pour lesquels il a été initialement conçu.

Paramétrage L'application doit pouvoir changer en profondeur les paramètres de l'ordonnancement. Cette caractéristique est différente de celle, déjà mentionnée, de flexibilité car elle ne se place pas au même niveau. La flexibilité consiste en prendre en compte avec un coût de réécriture minimal de nouveaux critères d'ordonnancement alors que le paramétrage concerne la modification à la volée, éventuellement par des outils d'observation extérieurs à l'application mais inclus dans le support d'exécution, des paramètres qui régissent le comportement de l'ordonnanceur.

Coût bornable La caractéristique de l'ordonnancement des applications hautement irrégulières vue en 4.1.1.2 mentionne que l'ordonnancement joue un rôle dans la quantité de travail nécessaire à la résolution du problème. L'un des problèmes que cela pose consiste à trouver un

équilibre entre le coût d'un ordonnancement proche de l'optimal et le gain qu'il procurerait. Le moyen de résoudre ce problème est de borner le temps alloué à l'ordonnanceur et de faire le meilleur ordonnanceur possible dans ce temps imparti. Il est clair que ce temps doit être un pourcentage du temps total de l'application. Il serait dénué de sens d'allouer une minute à l'ordonnanceur aussi bien pour une application durant une seconde que pour une application durant une heure.

Spéculatif Le calcul de l'ordonnancement optimal étant impossible, toute décision d'ordonnancement est spéculative et peut donc être remise en cause à tout moment.

Ces caractéristiques conditionnent les choix de conception de l'ordonnanceur, la structure de ce dernier devant être suffisamment générique et paramétrable pour s'adapter à un large spectre de critères et de comportements des applications. Il doit cependant offrir un schéma conceptuel assez puissant pour être d'une réelle aide au programmeur en le soulageant de la tâche d'ordonnancement. Nous présentons notre proposition pour un tel ordonnanceur générique dans la section 4.3.

4.1.3 Conclusion

Le cahier des charges d'un ordonnanceur générique pour les applications hautement irrégulières est donc le suivant :

1. **Les critères sont spécifiques à l'application** ce qui implique une grande flexibilité tant dans la définition de la politique que dans la modification du comportement à la volée. Il est cependant possible de regrouper les applications en classes, chaque classe ayant une politique d'ordonnancement spécifique.

2. **Action directe sur les activités** Cela implique la possibilité d'utiliser des mécanismes de déplacement d'activités comme la migration.

3. **Influence sur le coût de l'application** Le respect d'une politique liée à l'application permet de diminuer son temps d'exécution.

4. **Coût bornable** Le coût de l'ordonnancement doit être borné pour éviter de perdre les gains que le point 3) a permis.

De ce cahier des charges on déduit que l'ordonnancement doit être transparent au moment de la conception et que le mécanisme principal d'interaction entre l'ordonnanceur et l'application est la migration préemptive. En effet, elle permet une redistribution dynamique des activités sur les processeurs tout en étant transparente. Les critères utilisés pour ordonnancer sont également étroitement liés aux activités au lieu de l'être aux données comme dans les ordonnanceurs que nous avons présentés dans le chapitre 3. Cette particularité est dûe aux caractéristiques des applications visées pour lesquelles les accès aux données ne fournissent pas d'informations exploitables.

Un ordonnanceur efficace pour cette classe d'applications travaille donc *par et pour les activités.* Cette approche de l'ordonnancement se démarque des autres car elle considère les activités comme le point central de sa politique au lieu de les considérer comme un moyen. Nous allons voir dans la section suivante comment cette approche de l'ordonnancement se différencie de celle que nous avons étudié dans le chapitre précédent.

4.2 Ordonnancer = Placer + Migrer

Nous avons vu dans la section précédente que l'ordonnancement d'une application hautement irrégulière était différent des ordonnnancements classiques car il ne pouvait se faire qu'en utilisant les caractéristiques des activités et en agissant sur ces dernières. Le mécanisme essentiellement utilisé dans ce but est la migration. Nous allons dans cette section détailler en quoi cette approche est différente de celle utilisée traditionnellement par les ordonnanceurs en nous focalisant en particulier sur l'impact de la migration dans la politique d'ordonnancement.

4.2.1 Migration et ordonnancement

Dans cette section, nous allons étudier de manière plus formelle comment caractériser la partie dynamique de la virtualisation, en particulier par rapport à la notion traditionnelle d'ordonnancement.

Considérons une application A composée d'un ensemble d'activités. On pose :
$\mathbb{S}$ = {Ensemble des sites sur lesquelles s'exécute l'application A }
$\mathbb{T}$ = {Ensemble des activités de l'application A}
$\mathbb{D} = \{d / date_{début} \leq d \leq date_{fin}\}$ {Ensemble des dates}
Alors, ordonnancer $\mathbb{T}$ sur $\mathbb{S}$ revient à définir la fonction O suivante :

$$O : \mathbb{T} \rightarrow \mathbb{D} \times \mathbb{S}$$
$$t \rightarrow (d, s)$$

En conservant la même terminologie, ordonnancer en utilisant la migration consiste à définir la fonction V suivante :

$$V : T \rightarrow (\mathbb{D} \times \mathbb{S})^+$$
$$t \rightarrow (d, s)^+$$

La différence réside dans le fait que virtualiser revient à assigner un ensemble de doublets (site, date) à une tâche. Ces doublets correspondent à la séquence de sites et de moments où la tâche s'exécutera. Ordonnancer consiste à affecter un seul doublet à une tâche car la décision prise, par exemple, au moment de la création de cette dernière ne sera pas remise en cause durant cette exécution. L'ordonnancement est donc un cas particulier de la virtualisation, car si on restreint le cardinal de l'ensemble des doublets résultat de l'appel à 1 on obtient le même résultat qu'en effectuant un ordonnancement « traditionnel ».

Notons que la manière duale de voir l'ordonnancement avec migration consiste à considérer que la migration d'une activité consiste à terminer l'exécution d'une activité puis à en recréer une ayant exactement les caractéristiques de celle qui a été terminée. Cette dernière peut alors être ordonnancée suivant une politique sans migration c'est-à-dire en lui affectant une date et un site d'exécution au moment de sa prise en compte par l'ordonnanceur, après sa création. Cela revient d'ailleurs à diminuer la granularité des activités de l'application de manière transparente pour elle. Si nous reprenons les notations introduites ci-dessus, il faut pour cela introduire une fonction C :

$$C : \mathbb{T} \rightarrow \mathbb{T}'^*$$
$$t \rightarrow \{t' / t' \in \mathbb{T}'\}$$

Le rôle de cette fonction est de couper les activités en sous-activités, ces sous-activités étant ensuite ordonnancées avec la fonction O' :

$$O' : \mathbb{T}' \rightarrow \mathbb{D} \times \mathbb{S}$$
$$t' \rightarrow (d, s)$$

Cette approche suppose évidemment de disposer d'un mécanisme permettant de redémarrer une activité dans l'état exact dans lequel elle se trouvait au moment de l'arrêt. Cela peut être introduit

de deux façons :

Explicite Dans le code de l'application se trouvent des directives à certains endroits indiquant qu'il est possible de prendre une « photo » de l'activité. Cette photo servira ultérieurement à démarrer la nouvelle activité. C'est la méthode préconisée dans [Gal95].

Implicite À chaque instant, il est possible de geler de manière préemptive une activité pour la redémarrer exactement dans le même état dans un autre environnement. L'implantation de ce mécanisme est similaire à celui mis en jeu pour la migration.

La première approche modélise l'ordonnancement préemptif avec la fonction O', c'est-à-dire comme un ordonnancement non préemptif mais avec un découpage fin et à la demande de l'ordonnanceur des activités. La seconde, qui est celle au cœur de la virtualisation, implique la définition d'une nouvelle approche de l'ordonnancement. Cette approche doit prendre en compte la notion de mobilité des activités afin de l'utiliser au mieux en remettant constamment en cause les choix qu'elle a déjà pris et qui ne sont plus forcément valables car les conditions ont changé. L'ordonnancement dans ce cadre consiste à placer et déplacer dynamiquement les activités sur les processeurs afin de minimiser un ou plusieurs critères et cela sans pouvoir se baser sur des informations fournies par l'application. C'est une problématique proche de celle de la régulation de charge comme nous allons le voir.

4.2.2 Ordonnancement avec migration = Équilibrage dynamique de charge

Nous avons mentionné que l'ordonnancement de la classe d'applications qui nous intéressent se faisait par les activités et pour les activités. L'outil principal utilisé dans ce cadre est la migration car elle permet de remettre en cause à tout moment les décisions antérieures de placement des activités. Le parallèle est aisé avec les domaine des régulateurs de charge dont le rôle est de répartir des éléments, typiquement des données, d'une application entre les différents sites, les processeurs, sur lesquels elle s'exécute. Ici, au lieu de répartir les données (qui sont des calculs potentiels), nous répartissons les calculs réels c'est-à-dire le traitement sur les données au lieu des données.

Ici les éléments sur lesquels se fait la régulation sont les activités et le but du régulateur est de les placer pour respecter un (ou plusieurs) critère(s). Ce critère caractérise la politique d'ordonnancement mise en place en conditionnant les sites sur lesquels une activité s'exécutera. Les ordonnanceurs que nous avons étudié dans ce document comportent tous un mécanisme apparenté à la régulation de charge mais, dans le cas de ATHAPASCAN-1 et CILK, ils gèrent les activités qui n'ont pas encore été créées. Par exemple, le mécanisme de *work-stealing* de CILK est issu des études sur les politiques de régulation de charge dites « *receiver initiated* ». Grâce à la migration, des politiques similaires peuvent être mises en place sur les activités d'une application à tout moment de leur exécution, ce qui, lorsqu'on y ajoute des processus légers, permet d'augmenter le degré réel de parallélisme de l'application sans sacrifier ses performances.

Notre proposition d'ordonnanceur générique est issue de cette constation et propose d'ordonnancer en équilibrant la *charge en activités* de chaque processeur. Nous allons maintenant la détailler.

4.3 L'approche GTLB

Dans cette section, nous allons présenter GTLB[4], notre proposition pour un ordonnanceur générique pour les applications hautement irrégulières à grain fin. Nous allons commencer par

4. Le sigle GTLB signifie *Generic Threads Load Balancer* ou Équilibreur générique de processus légers en Français.

fixer les objectifs à atteindre en les détaillant sous l'angle applicatif. Ensuite nous présenterons les grandes lignes de notre modèle. Nous continuerons en détaillant l'interface et la façon dont une application peut paramétrer le comportement de l'ordonnanceur de manière simple. Enfin, nous illustrerons sur un exemple comment l'ordonnanceur s'utilise en pratique.

4.3.1 Objectifs

Le premier objectif de notre ordonnanceur est la **transparence**, il doit être indépendant de l'application. En particulier, le concepteur de l'application ne doit pas avoir à tenir compte du fait qu'elle sera ordonnancée. L'exception est bien sûr quand il veut influer sur le comportement de ce dernier. Cette transparence totale est la condition pour une bonne généricité de l'ordonnanceur car elle signifie qu'il peut être appliqué sur une application comme un élément, souvent une couche, du support d'exécution.

Le deuxième objectif que nous voulons atteindre est la **flexibilité**. Nous entendons par là que l'ordonnanceur doit être conçu suivant un schéma générique lui permettant d'utiliser plusieurs politiques ayant des finalités différentes sans remettre en cause sa structure interne. Il doit donc se présenter comme une interface minimale utilisable dans différents contextes.

Les applications irrégulières étant la cible première de cet ordonnanceur, il ne peut pas utiliser d'informations liées aux dépendances des informations. Cela implique qu'il ne peut pas anticiper sur le comportement de l'application[5]. L'ordonnanceur fonctionnera donc de manière essentiellement réactive, les anticipations qu'il fait ne sont que des tentatives de placement et seront très probablement corrigées dans la suite de l'exécution.

Le dernier objectif que notre ordonnanceur doit viser est celui du coût. Comme nous travaillons sur des applications hautement irrégulières, donc dans un contexte entièrement indéterministe, le coût de l'ordonnancement ne peut pas être prévu. Il doit résulter d'un compromis entre la « qualité » de l'ordonnancement et son coût, typiquement en exprimant le **coût maximal** et en réalisant le meilleur possible sans dépasser cette contrainte.

4.3.2 Caractéristiques

Nous avons longuement expliqué que, contrairement aux approches traditionnelles d'ordonnancement, celui que nous proposons ne peut pas se baser sur les dépendances entre les données, il ne peut utiliser que les caractéristiques des activités pour effectuer l'ordonnancement. Les critères à minimiser sont dépendants de l'application mais il est possible d'extraire des critères communs à un ensemble d'applications ayant des comportements similaires.

L'ordonnanceur aura donc une structure générique lui permettant de prendre en compte un large spectre de critères sur différents types d'applications. Son architecture et son fonctionnement doivent donc être complètement séparés de l'application. Pour cela nous allons utiliser les processus légers dans un cadre différent de celui que nous avons étudié depuis le début de ce document.

Jusqu'à présent nous avons présenté les processus légers comme le moyen permettant d'exécuter efficacement des activités de granularité quelconque, même très fine. Les processus légers sont regroupés dans des processus lourds qui sont eux-même hébergés sur des processeurs. L'affectation des processus légers dans les processus lourds, et donc sur les processeurs, est faite uniquement dans un but d'efficacité. L'approche que nous proposons est basée sur le constat que les processus légers peuvent aussi être utilisés comme des activités *à l'intérieur de l'application* qui s'occupent

5. À part en utilisant des méthodes heuristiques observant le comportement passé de l'application pour spéculer sur son comportement futur. Ce genre de méthodes sort du cadre de notre étude.

de tâches[6] liées à l'administration, au sens large, de l'application. Les processus légers servent dans ce cas à introduire de la *multiprogrammation légère* dans l'application. Cette notion de multiprogrammation légère étant au centre de notre proposition nous allons nous y attarder en faisant une analogie avec celle de multiprogrammation présente dans les systèmes d'exploitation traditionnels.

Tous les systèmes d'exploitation modernes sont multitâches, ils permettent d'exécuter simultanément plusieurs applications de manière transparente pour ces dernières. Cette fonctionnalité est utilisée pour faire cohabiter sur le même processeur des applications des utilisateurs et des outils liés à l'administration du système et, le cas échéant, à des fonctionnalités réseau. Par exemple, le gestionnaire de swap disque s'exécute concurremment avec les applications des utilisateurs sans affecter directement leur comportement. L'idée que nous défendons pour un ordonnancement réellement transparent est une transposition de ce modèle aux processus légers. De la même manière qu'une machine est multiprogrammée avec plusieurs processus lourds s'exécutant concurremment, le processus d'une application est multiprogrammée avec des processus légers applicatifs et d'autres chargés de gérer le comportement de l'application et, en particulier, son ordonnancement. Nous présentons cette architecture sur la figure 4.2.

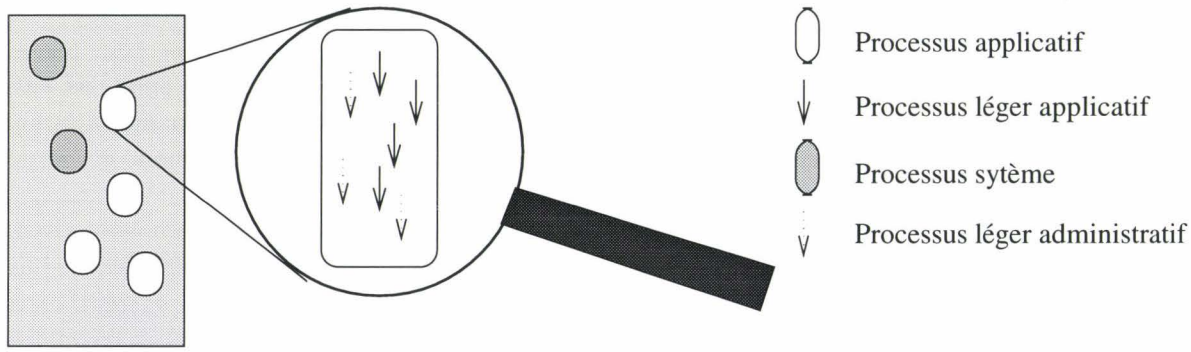


FIG. 4.2 - *Multiprogrammation légère et multiprogrammation lourde*

L'ordonnanceur se présente donc dans notre approche comme une entité qui existe dans l'application parallèlement à ses activités. Cela lui permet de fonctionner de manière transparente avec cette dernière. Si on considère les activités d'ordonnancement comme des éléments du support d'exécution on voit qu'il y a une séparation complète entre les activités du support et celles de l'application, tout comme les processus UNIX utilisent le démon NFS d'une machine sans s'en rendre compte. Dans le cadre qui est le nôtre, cela crée un **asychronisme total** entre l'application et la régulation, ce qui n'est pas le cas des approches d'ordonnancement non préemptives que nous avons vues précédemment. Grâce aussi à la migration, le placement et le déplacement des activités peuvent se produire n'importe quand durant l'exécution au lieu d'être restreints à certains moments précis comme la création ou l'accès à une variable partagée. C'est possible car le calcul de l'ordonnancement se fait parallèlement à l'exécution de l'application dans des activités séparées qui, dans notre modèle, sont exécutées concurremment. La structure d'une application virtualisée utilisant notre ordonnanceur se présente comme sur la figure 4.3.

L'application se présente comme un ensemble de processus lourds contenant des processus légers qui sont soit des activités applicatives, soit des activités du support d'exécution qui servent à l'ordonnancement. Comme toutes les activités d'un processus s'exécutent en temps partagé (contrainte de concurrence de la virtualisation), cette structuration de l'ordonnanceur en activités permet de

6. Le mot « tâche » est ici utilisé au sens de travaux et non au sens d'activités comme dans le reste de ce document.

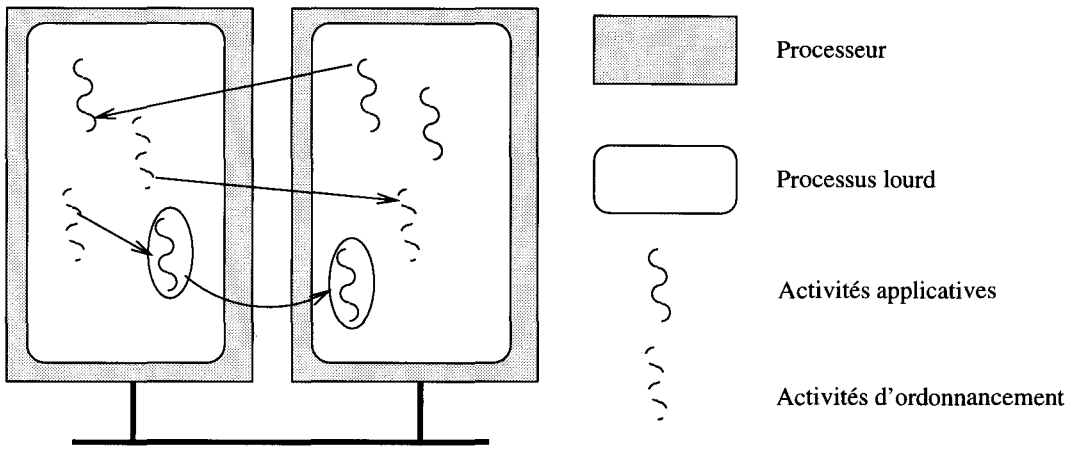


FIG. 4.3 - *Structure d'une application virtualisée utilisant notre ordonnanceur. Sur le processeur de gauche une activité d'ordonnancement migre une activité applicative vers le processeur de droite.*

borner le coût de l'ordonnancement en limitant le nombre d'activités d'administration et leurs priorités respectives. Notons que, sur un processeur, l'ordonnancement local se fait à deux niveaux : le premier consiste dans le partage du temps processeur entre les processus lourds, le second entre les processus légers d'un processus lourd.

L'idée au centre de notre approche est donc celle d'un ordonnancement qui se déroule de manière concurrente avec l'application par le biais de processus légers indépendants. L'application de ce concept pour la construction de notre ordonnanceur fait l'objet de la section suivante.

4.4 Fonctionnement

Le but de l'ordonnanceur est d'assurer le placement des activités suivant des critères qui peuvent varier d'une application à l'autre. Pour cela il peut agir sur la localisation des activités au moment de leur création ou durant leur exécution. Pour le **placement** l'application fait **explicitement** appel à l'ordonnanceur pour lui demander le meilleur endroit pour créer une nouvelle activité alors que pour le déplacement c'est l'ordonnanceur qui prend seul la décision de changer une, ou des, activité de site d'exécution. Cette décision est prise en **réaction** à la constation d'un déséquilibre alors que le placement est fait pour éviter l'apparition d'un tel déséquilibre. Bien que fonctionnellement identique, ces deux opérations d'ordonnancement sont donc utilisées dans des circonstances et avec des objectifs différents.

L'ordonnanceur que nous proposons contient une activité résidant en permanence sur chaque processeur, à l'intérieur du processus lourd qui encapsule les activités applicatives. L'activité est chargée de surveiller la charge locale du processeur sur lequel elle s'exécute et d'agir lorsqu'elle constate une modification dans la charge locale. Son action consistera à créer d'autres activités, d'un type différent, qui seront chargées de rétablir l'équilibre rompu par la variation de charge locale. Nous allons étudier la manière dont cette structure fonctionne pour ordonnancer une application dans la section suivante.

Le placement à la création nécessite, pour être efficace, d'avoir une connaissance globale pour prendre sa décision. En effet s'il n'a qu'une vision incomplète de l'état des processeurs il ne peut pas choisir le plus approprié pour éviter l'apparition d'un déséquilibre. Par contre un déplacement

d'activités réagit à la constation d'un déséquilibre qui est causé, si l'état précédent etait l'équilibre, par la modification de la charge d'un processeur. Il ne nécessite donc que des informations locales et pas une vue globale de l'état de charge des processeurs.

Le choix d'un placement nécessite donc d'avoir une vue globale de l'état de charge des processeurs. Cet état peut être maintenu sur un unique processeur ou au contraire être diffusé sur tous. Dans notre cas, nous voulons consulter cet état à chaque création d'activités nouvelles c'est-à-dire très fréquemment puisque nous visons des applications ayant des activités de courte durée. La communication, du type demande de service (le service étant « Sur quel site placer une nouvelle activité? »), implique deux messages[7] ce qui est un surcoût difficilement admissible, indépendamment du goulot d'étranglement que représente le stockage centralisé de cet état global. La distribution de cette connaissance implique donc la présence d'une gestion distribuée d'un état global, c'est-à-dire la mise en place d'une *politique d'information* visant à maintenir cet état global sur tous les processeurs.

Pour résumer, nous avons deux éléments principaux utilisés pour l'ordonnancement :

Une connaissance de la charge globale de la machine La politique de placement à la création des activités utilise une connaissance globale de l'état de l'architecture. Cette connaissance est stockée localement pour réduire le coût de la décision de placement.

Un rééquilibrage réactif Quand une modification de l'état d'un processeur est constatée, elle doit entraîner une opération de rééquilibrage qui sera faite par une (ou des) opération de migration.

Ces deux éléments sont liés, l'apparition d'un déséquilibre signale une modification de la charge des processeurs et donc implique une mise à jour des informations globales de charge. Dans notre ordonnanceur le déclenchement du rééquilibrage met aussi à jour les informations de charge sur le site choisi pour l'équilibrage, cette mise à jour étant progressivement, de proche en proche, étendue à l'ensemble des processeurs. De manière concrète cela se matérialise par la présence continue d'une activité sur chaque site dont le rôle est de surveiller la charge locale et de réagir lorsqu'un changement de cette dernière survient. Le but de cette réaction est double :

1. **Informer les autres processeurs du changement** Chaque processeur ayant une connaissance de l'état global quand une modification locale survient, elle doit être signalée aux autres processeurs pour qu'ils mettent à jour leurs informations.

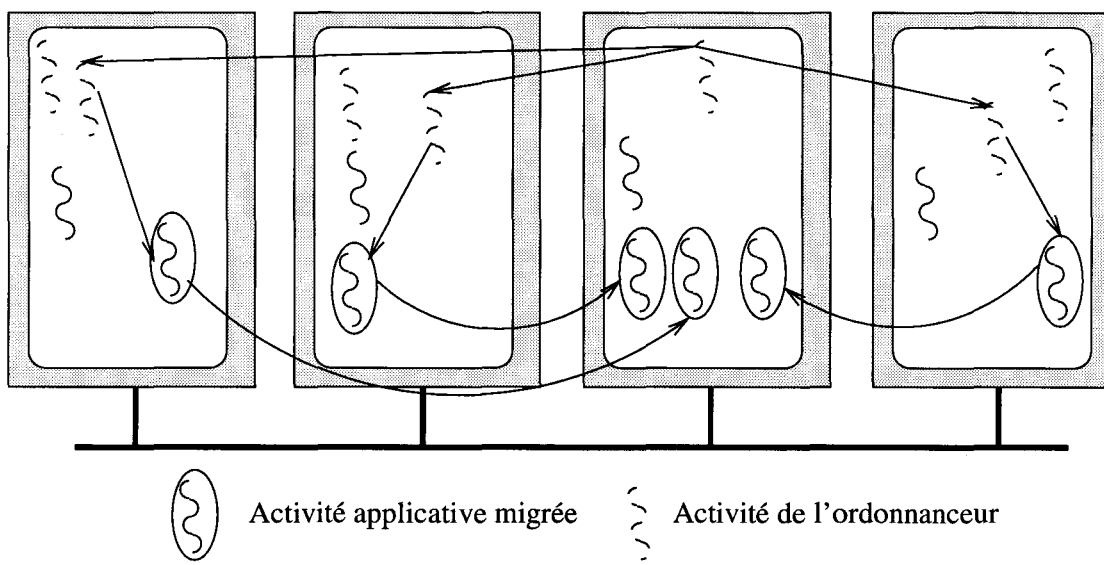
2. **Agir pour compenser le déséquilibre apparu** en transférant des activités de ou vers d'autres processeurs.

Le premier point n'est qu'une simple modification de valeur alors que le second nécessite l'utilisation de protocoles plus évolués. Dans notre modèle, et dans l'implantation qui en est faite dans PM^2, il n'est pas possible d'envoyer des messages, toute communication se faisant par des invocations distantes de services. Cela semble être un inconvénient dans ce cas ; nous allons montrer qu'il n'en est rien car il va nous permettre de faire « d'une pierre deux coups » en réalisant les opérations 1) et 2) en une seule communication.

Quand l'activité « moniteur » découvre une modification de la charge locale elle va en informer les autres processeurs en créant sur eux une activité dont le rôle est de mettre à jour les informations sur

7. Toute communication entre deux processeurs, même une demande de service, correspond toujours à un échange de messages à un niveau suffisamment bas dans l'implantation.

l'état global stockées localement et qui agira, si c'est possible, pour rétablir l'équilibre en déplaçant des activités. Alors que le premier point peut être fait en diffusant la mise à jour partout, donc en créant une activité sur chaque processeur, le suivant, l'opération de rééquilibrage proprement dite, ne peut pas être fait de cette manière. Il s'agit en effet d'une opération point à point, qui ne concerne que deux processeurs, celui qui a émis le message et celui qui le reçoit et qui essaie de corriger le déséquilibre, il n'est pas souhaitable d'effectuer $n-1$ opérations de régulation simultanément avec le même processeur. En effet, cela entraînerait, pour compenser le déséquilibre, un afflux de processus légers sur le site originellement peu chargé ce qui créerait un déséquilibre plus grand que celui qu'il était censé combattre. Nous représentons ce phénomène sur la figure 4.4 sur laquelle les activités en pointillés représentent les activités de l'ordonnanceur alors que celles représentées par un trait continu sont les activités applicatives.



BU LILLE 1

FIG. 4.4 - *Inondation d'un processeur par l'afflux de processus légers migrés pour compenser un déséquilibre*

Pour éviter ce comportement, il est nécessaire de trouver un compromis entre ces deux besoins pour assurer une bonne diffusion de l'information tout en limitant les problèmes d' « inondation » liés à la régulation. Pour cela, nous utilisons une architecture logique de communication en anneau (unidirectionnel) pour la diffusion de l'information. De cette manière, les informations sont propagées de proche en proche et le risque d'inondation diminue. Il ne faut cependant pas oublier que l'activité créée pour mettre à jour les informations est aussi chargée de compenser le déséquilibre induit par la variation de charge ce qui signifie que, lorsque c'est possible, elle va agir, en déplaçant des activités, ce qui modifiera de nouveau la charge. Si elle propage ensuite la valeur de charge qui lui a fait déclencher l'opération, elle va propager une valeur qui est fausse puisqu'il s'agit de celle qui avait été évaluée avant son action. Il est donc préférable dans ce cas d'attendre pour que l'information correspondante à la charge modifiée soit réémise par le processeur avec lequel l'opération de rééquilibrage a eu lieu. Notons aussi que cette opération a provoqué une modification de la charge d'un autre processeur qui émettra donc aussi un message, qui prendra la forme d'une activité dans notre modèle, susceptible d'entraîner une autre opération d'équilibrage. Ainsi, de proche en proche, on assistera à un lissage de la charge jusqu'à atteindre de nouveau un état d'équilibre.

Ce comportement a l'inconvénient de faire une suite d'équilibrages « locaux[8] » qui nécessite donc un certain nombre d'étapes pour arriver dans un état totalement équilibré. À cette objection, justifiée, nous apportons deux réponses :

Le placement peut se faire n'importe où. Il est utilisé pour répartir les activités de manière relativement homogène, les opérations de migration ne sont ensuite utilisées que pour compenser les déséquilibres qui apparaissent.

Les caractéristiques des applications Pour éliminer ces rééquilibrages locaux il faut définir un schéma global qui serait soit plus coûteux, avec des algorithmes distribués utilisant beaucoup de messages, soit moins souple car centralisé. De plus le comportement des applications que nous visons est hautement dynamique et un placement qui serait parfait à l'instant t peut devenir très rapidement inadapté. Dans l'approche que nous proposons l'équilibrage est continu et peut donc être perpétuellement remis en cause ce qui sera le cas dans la plupart des cas.

Le fonctionnement complet de l'ordonnanceur est illustré sur la figure 4.5 que nous allons commenter.

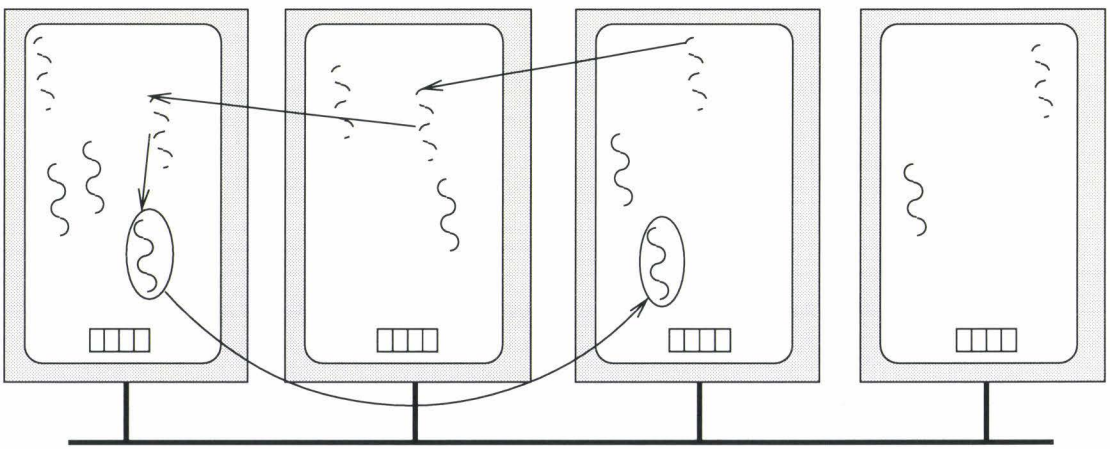


FIG. 4.5 - *Fonctionnement complet de l'ordonnanceur générique GTLB*

Sur chaque processeur est stocké l'état global de charge. Cet état est modifié à l'initiative des processeurs qui constatent une modification de leur charge la transmette à leur voisin. Pour cela ils créent une activité sur le processeur concerné. Cette activité va d'abord mettre à jour la valeur de charge dans la charge globale. Elle va ensuite tenter de compenser le déséquilibre éventuel causé par la modification de charge à l'origine de la création de l'activité. Si elle y arrive elle s'arrêtera là sinon elle propagera l'information qu'elle détenait, qui est toujours valable puisqu'aucune modification n'a eu lieu, sur son voisin en créant une activité identique. De cette manière une information qui n'est plus valide n'est pas propagée ce qui évite d'utiliser du temps processeur et du trafic réseau pour des informations qui ne sont plus pertinentes. L'anneau utilisé pour la propagation d'une modification est unidirectionnel afin d'éviter que deux opérations d'équilibrage concernant le même processeur se produisent au même moment. Sa configuration, la façon dont est calculée la notion

8. Locaux est un terme impropre car la notion de voisinage est purement virtuelle dans notre implantation. Dans la mesure où nous nous intéressons à des réseaux d'interconnexion quelconque nous ne pouvons pas prendre en compte les caractéristiques physiques du réseau pour déterminer la localité.

de voisinage, est fixée au démarrage de l'application et ne prend pas en compte, dans la version actuelle, les caractéristiques de l'architecture.

Une période d'équilibrage[9] consistera typiquement en une suite d'opérations d'équilibrages sur des domaines locaux. L'équilibrage se poursuivra ensuite de domaines en domaines pour finalement amener à un équilibre global sur l'ensemble des processeurs.

4.4.1 Bornage du coût relatif de l'ordonnancement

Nous avons vu que l'ordonnancement « parfait » des applications qui nous intéressent était un but généralement inaccessible ou, quand il était possible, trop coûteux pour être intéressant. L'ordonnancement doit donc résulter d'un compromis entre son coût et son efficacité. Dans le modèle que nous proposons, l'ordonnancement est réalisé par des activités. Ces activités se partageant le temps processeur avec celles de l'application, il est possible de borner leur coût en limitant leur nombre. En supposant l'absence de priorités et que le temps processeur est également réparti entre les activités par un mécanisme de partage de temps, on pose :

N : le nombre de processeurs

$N^i_{act}(t)$ le nombre d'activités applicatives sur le processeur numéro i à l'instant t

On peut alors borner le pourcentage du coût de l'ordonnancement sur un processeur pour l'instant t par la formule :

$$c(t) \leq \frac{N}{\min\limits_{i \in \{1,N\}} (N^i_{act}(t)) + N}$$

Le numérateur, qui majore le nombre d'activités de l'ordonnanceur présentes simultanément sur un processeur, est dû à une particularité de l'ordonnanceur qui garantit qu'il ne peut y avoir plus d'une activité de régulation créée par un processeur sur chaque processeur. Si le processeur i a créé une activité de régulation sur le processeur j, aucune autre activité de régulation créée par ce processeur i ne peut s'exécuter en même temps sur j.

Dans la mesure où un processeur ne crée jamais d'activités d'ordonnancement sur lui-même (autre que l'activité moniteur évidemment) ce nombre est borné par N, $N-1$ ont été créées par les autres processeurs et le dernier est l'activité moniteur. À chaque instant le coût de l'ordonnanceur est donc borné par le rapport entre le nombre total de processeurs, et donc le nombre d'activités de l'ordonnanceur présentes sur chaque processeur, et le nombre minimum d'activités sur chaque processeur. Nous avons représenté sur la figure 4.6 des courbes montrant le coût pour 10, 100 et 1000 processeurs. Ces courbes mettent en évidence le rôle du nombre d'activités applicatives dans le coût relatif de l'ordonnancement. Les applications que nous visons comportant un grand nombre d'activités cela n'est pas très gênant.

L'un des objectifs de notre ordonnanceur était d'être d'un coût bornable afin de ne pas passer plus de temps à ordonnancer une application qu'à l'exécuter. Cet objectif est atteint car le nombre d'activités de l'ordonnanceur est borné par le nombre de processeurs donc le coût est lui aussi borné par le rapport entre ce nombre et le nombre d'activités applicatives.

9. L'équilibrage est permanent dans notre modèle puisque les activités qui l'effectuent s'exécutent en permanence. Il n'y a donc pas de phase d'équilibrage au sens propre. Nous désignons par cette expression la suite d'événements qui va mener d'un état déséquilibré à un état plus équilibré.

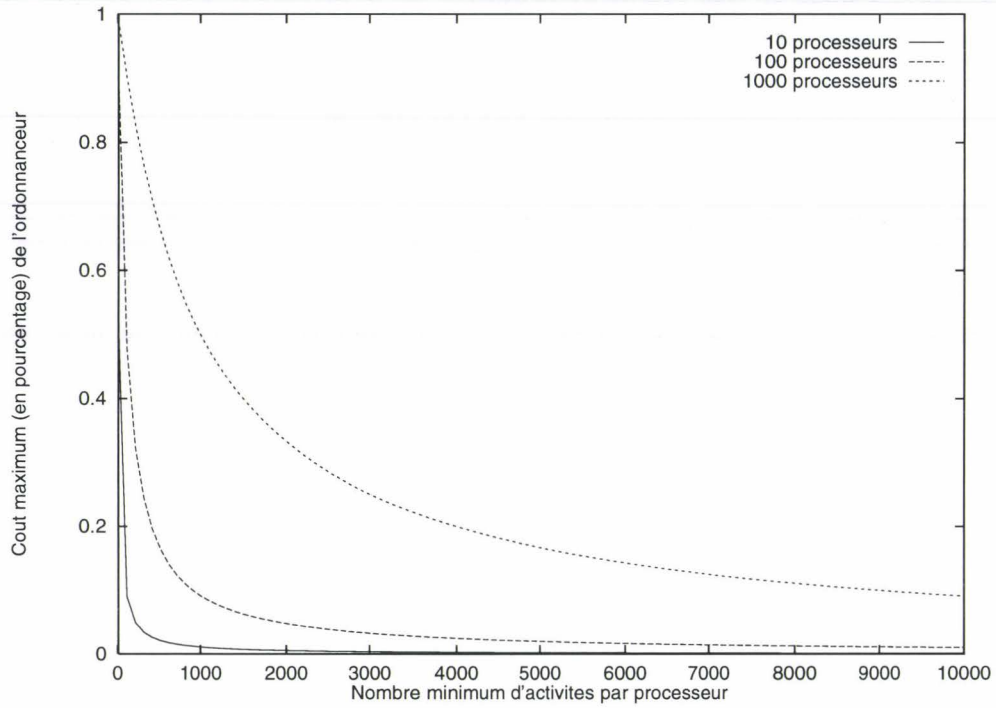


FIG. 4.6 - *Modélisation du coût maximal de l'ordonnanceur*

4.5 Implantation

Dans cette section, nous allons détailler la façon dont nous avons implanté le régulateur générique dont l'étude fait l'objet de ce chapitre. Nous allons tout d'abord présenter ses caractéristiques du point de vue de l'utilisateur puis nous détaillerons l'interface proposée pour définir de nouvelles politiques ou paramètrer des politiques existantes. Nous rentrerons ensuite dans les détails plus techniques de l'implantation.

4.5.1 Caractéristiques d'une politique

Au-delà du schéma de fonctionnement présenté dans les sections précédentes, l'ordonnanceur que nous proposons a une vocation générique, c'est-à-dire qu'il peut être facilement utilisé avec des critères d'ordonnancement différents de ceux proposés « en standard ». Une nouvelle politique se définit par deux choses :

1. **Le critère de régulation** est la valeur, ou l'ensemble de valeurs, à minimiser pour faire un bon ordonnancement. Typiquement, pour avoir tous les processeurs occupés de manière homogène, il s'agira du nombre de processus légers sur chaque processeur. Cette caractéristique est bien évidemment au cœur de la politique mise en œuvre. Nous montrerons dans ce document des politiques d'ordonnancement utilisant divers critères.

2. **Le comportement de l'ordonnanceur** Sous ce terme un peu générique nous plaçons la valeur de tous les paramètres qui conditionnent les réactions de l'ordonnanceur. Un exemple typique de ces paramètres est le délai entre deux lectures de la charge par l'activité moniteur ou le seuil à partir duquel on estime qu'il y a déséquilibre.

Nous avons vu dans la section précédente que le coût de la politique d'ordonnancement était lié au nombre d'activités qu'il utilisait. Nous avons également vu que ce coût est borné quelle que soit la politique mise en œuvre. Ce coût est une conséquence du comportement de l'ordonnanceur qui est entièrement paramétrable par l'application, nous verrons dans la suite qu'il est possible de fixer un intervalle à l'intérieur duquel ces valeurs fluctuent.

4.5.2 Définition de la politique d'ordonnancement

Le critère de régulation définit la valeur de charge de chaque processeur. Cette valeur permettra de détecter les déséquilibres qui seront compensés par l'ordonnanceur en migrant des activités et aussi à limiter leur apparition par le placement à la création. Elle conditionne le moment où l'ordonnanceur agira pour compenser une situation de déséquilibre en se basant sur l'« évaluation » de la charge. Les déplacements d'activité ont pour but de minimiser l'écart entre les évaluations de charge des processeurs entre lesquels l'équilibrage se fait. Cela implique que le choix des activités à migrer est fonction du critère de régulation, donc de la politique d'ordonnancement. De plus, la constatation d'un déséquilibre n'implique pas qu'on puisse automatiquement y remédier : si, par exemple, aucune activité n'est migrable à un instant donné, ce qui n'est pas un cas exceptionnel, alors aucune action n'est possible pour l'instant.

À la lumière de cette discussion, nous allons établir la liste des fonctionnalités qui permettent de définir un nouveau critère de régulation et ses fonctions associées donc, dans notre modèle, une politique d'ordonnancement.

Dans notre modèle, une politique d'ordonnancement est définie par les critères qu'elle prend en compte :

Variation importante de la charge L'émission de la charge d'un processeur vers son voisin ne se déclenche que lorsque une variation significative de la charge locale est détectée. La notion de variation significative est propre à la politique mise en œuvre.

Caratérisation du déséquilibre Une action d'équilibrage est effectuée lorsqu'un déséquilibre est détecté. Le critère qui définit un tel déséquilibre doit être exprimé par la politique. Cette caractérisation peut être éventuellement prise en charge par le support, l'application indiquant quel degré de déséquilibre elle peut tolérer. Nous y revenons dans la suite.

Possibilité d'équilibrage Comme nous l'avons signalé plus haut, la constatation d'un déséquilibre n'implique pas forcément qu'il est corrigible. La raison que nous avons donnée était liée aux caractéristiques exécutives des activités applicatives (leur migrabilité[10]). Ce cas peut être détecté par le support. Dans d'autres cas ce peut être la politique d'ordonnancement qui décidera si l'équilibrage est possible ou non.

Placement à la création Quand une nouvelle activité est créée, le choix du processeur sur lequel elle commencera son exécution dépend de la politique d'ordonnancement. Un élément de cette politique doit donc être écrit dans ce but. Notons que la gestion des informations globales qui seront généralement utilisées pour prendre cette décision est assurée par la partie générique de l'ordonnanceur. Le composant spécifique à la politique ne concerne que la prise de décision proprement dite, pas la gestion des informations utilisées pour la prendre.

Détermination des éléments à migrer Une fois qu'un déséquilibre est constaté, le choix des activités applicatives qui seront déplacées doit être fait afin de rétablir au mieux l'équilibre

10. Ce néologisme exprime la possibilité pour une activité d'être migrée ou non.

rompu. Le choix des activités à déplacer sera mieux fait par la politique, qui connaît le(s) critère(s) à prendre en compte dans ce but, que par la partie générique qui ne pourra les choisir que suivant une méthode aléatoire. On peut faire ici un parallèle avec MILLIPEDE où le support lui-même prenait cette décision. Il pouvait le faire car le critère utilisé, la réduction des communications, utilise des informations qui sont connues du systéme et pas de l'application. Dans le cadre, générique, qui nous intéresse la même approche, entièrement transparente, n'est pas possible.

De cette liste on peut déduire que de nombreux éléments doivent être donnés pour définir une nouvelle politique. Cela est dûe à la volonté d'être le plus générique possible qui est au centre de notre ordonnanceur. Nous pensons que laisser une grande marge de choix au programmeur lui permet de définir une politique exactement adaptée à sa classe d'applications voire même à son application. Le compromis est à trouver entre une trop grande généricité, qui donne trop de travail au programmeur en ne fournissant pas suffisamment d'éléments, et un cadre trop contraignant comme celui de MILLIPEDE.

Notre approche consiste à laisser une large marge de manœuvre en fournissant cependant avec le squelette d'ordonnanceur générique des composants réutilisables. Nous allons détailler comment cette approche est mise en œuvre dans la section suivante consacrée à l'interface.

4.5.3 Interface

Dans cette section nous allons détailler l'interface qui permet de paramétrer le régulateur pour une classe d'applications, ou une application, précise. Nous détaillerons aussi quelques points relatifs à l'implantation.

4.5.3.1 Choix d'implantation

Nous avons voulu privilégier dès le départ une réécriture minimale des applications conçues avec PM^2 pour les ordonnancer. Nous avons donc choisi de proposer l'ordonnanceur générique comme un élément qui se greffe sur une application PM^2 déjà existante. Dans ce but il est écrit, ainsi que son interface, entièrement en langage C.

Il est conçu comme une couche qui se positionne entre PM^2 et l'application. Cette dernière est conçue **indépendamment du fait qu'elle sera ordonnancée ou non**. Le paramétrage de l'ordonnanceur peut être fait par l'application elle-même (dynamiquement) ou bien statiquement en modifiant avant la compilation les paramètres régissant le comportement dynamique. La structure du système que nous proposons est donc classiquement vue comme des couches fournissant des fonctionnalités.

L'aspect générique tient d'une part à la redéfinition possible de nouveaux critères pour quantifier la charge à équilibrer et d'autre part à la paramétrisation du comportement de l'ordonnanceur. À cet effet l'ordonnanceur générique dispose de points d'entrée qui permettent de spécifier le(s) critère sur lequel se fera l'ordonnancement ainsi que son comportement. Nous allons détailler les interfaces de ces deux aspects dans les sections suivantes.

4.5.3.2 Déclaration du critère et des opérations associées

Nous avons vu que, dans le cadre qui nous intéresse, ordonnancer les activités d'une application revient à équilibrer la charge que ses activités représentent. Suivant le (ou les) critère qui est utilisé pour définir la charge, la politique d'ordonnancement varie.

Dans cette optique, pour construire une nouvelle politique d'ordonnancement il est nécessaire de donner une nouvelle définition pour la charge des processeurs. Pour s'intégrer dans l'ordonnanceur générique, cette définition doit suivre un cahier des charges précis en fournissant notamment, pour un nouveau critère, l'ensemble des fonctions dont l'ordonnanceur a besoin. Nous avons identifié ce cahier des charges dans la section 4.5.2.

Définition d'un nouveau critère Pour déclarer un nouveau critère de régulation il faut essentiellement définir un nouveau type de données[11] qui contiendra la valeur de charge ainsi que des opérations qui doivent être définies sur ce type. Si nous étions dans un environnement à objets, la charge serait typiquement une classe dont hériterait toute nouvelle définition de charge. Ces opérations sont au nombre de six. Nous les récapitulons dans le tableau 4.1.

Nom	Usage
`get_load`	Lit la charge locale
`alloc_load_info`	Alloue de la mémoire pour une information de charge
`free_load_info`	Libère la mémoire utilisée pour une information de charge
`pack_load_info`	Empaquette une information de charge pour pouvoir la transmettre
`pack_load_info`	Dépaquette une information de charge transmise
`load_value`	Convertit la charge en un entier

TAB. 4.1 - *Ensemble des opérations à définir sur la charge*

Nous n'allons pas nous attarder sur les traditionnelles, comme l'empaquetage et le dépaquetage par exemple, qui sont bien connues de tous les programmeurs d'applications parallèles. Nous allons cependant détailler le rôle de la fonction qui convertit une information de charge, dont les caractéristiques sont inconnues de la partie générique du régulateur, en un entier.

Nous avons mentionné précédemment que le trop grand nombre de fonctions à définir pour créer une nouvelle politique d'ordonnancement pouvait être un frein et qu'un compromis était nécessaire entre une trop grande flexibilité et la réutilisabilité de composants existants pour construire une nouvelle politique. En définissant un type commun à toutes les politiques, ici un entier, il est possible de fournir des éléments qui se baseront sur cette propriété pour ordonnancer et qui seront donc utilisables avec toutes les définitions de charge. C'est dans ce but que nous avons ajouté dans l'interface de la charge cette fonctionnalité. Elle permet de définir un nouveau critère sans avoir à préciser l'ensemble des fonctions caractérisant une politique qui sont données dans la section suivante et en utilisant à leur place des fonctions déjà existantes travaillant uniquement sur la valeur d'évaluation de la charge définie par `load_value`.

Si nous prenons le cas d'une classe d'applications dont les critères de régulation sont le nombre de processus légers présents sur chaque processeur et la quantité mémoire disponible la fonction `load_value` pourrait retourner $\frac{nb\_threads*3+memoire\_libre}{4}$.

Opérations associées Après avoir détaillé la définition d'un nouveau critère de charge et les opérations qui lui sont associées, nous allons maintenant regarder l'interfaçage permettant de spécifier une nouvelle politique utilisant un critère de charge original défini comme présenté dans la section précédente.

Nous avons présenté dans la section 4.5.2 les fonctionnalités d'une politique d'ordonnancement,

11. Dont le nom est `load_info`.

nous allons logiquement retrouver ces éléments dans l'interface que nous proposons. Nous récapitulons les fonctions et leur utilité dans le tableau 4.2.

Nom	Rôle
`significant_load_change`	Déterminer si un changement dans la charge locale est suffisamment important pour motiver une transmission de la nouvelle charge
`severe_disbalance`	Déterminer si le déséquilibre entre deux processeurs est assez important pour lancer une opération de rééquilibrage
`balancing_possible`	Déterminer si un équilibrage de charge est possible entre deux processeurs
`place_thread`	Trouver le site sur lequel créer une nouvelle activité
`choose_threads`	Choisir les activités applicatives qui seront migrées pour rétablir l'équilibre

TAB. 4.2 - *Fonctions nécessaire à la définition d'une nouvelle politique d'ordonnancement*

Dans la section 4.6 nous présenterons un exemple concret de politique d'ordonnancement. À cette occasion, nous détaillerons les aspects syntaxiques, en précisant en particulier les paramètres de chaque fonction ainsi que leur utilisation.

4.5.4 Détails sur l'implantation

4.5.4.1 Fonctionnement des activités de régulation

Nous allons, dans cette partie, détailler la manière dont fonctionnent les activités de régulation qui sont créées par les activités moniteur pour signaler une modification de charge et essayer de rétablir l'équilibre. Cela nous amènera à présenter quelques structures de données internes utilisées par l'ordonnanceur générique.

Le mécanisme unique de communication de PM^2 étant l'invocation distante de services, c'est de cette manière que sont créées ces activités de régulation. Elles comportent en paramètres les coordonnées et la charge du site à l'initiative de l'activité. L'activité commence par stocker cette charge dans le tableau contenant la charge globale. Ensuite elle va tester, dans cet ordre, si un déséquilibre existe entre le processeur d'où elle vient et celui sur lequel elle s'exécute. Si c'est le cas elle va tester si elle peut corriger ce déséquilibre. Dans l'affirmative, elle va invoquer la fonction générique qui va effectuer cet équilibrage. Dans le cas où cette opération n'est pas possible, s'il n'y a pas de déséquilibre ou s'il n'est pas corrigible, la nouvelle information de charge est simplement transmise au voisin car elle est supposée toujours valide. Ce n'est pas le cas si une action d'équilibrage a été entreprise car elle va entraîner une modification des valeurs rendant donc erronée l'information contenue dans l'activité.

L'ordonnanceur n'a pas de mécanisme global de synchronisation et fonctionne en mode multiprogrammé, il se pose donc le problème classique des accès concurrents à la structure de données partagée qui stocke l'état global sur chaque processeur. En effet, plusieurs activités créées par différents processeurs (ou par le même si les délais d'acheminement des messages créants les activités ne sont pas uniformes) peuvent exister simultanément dans le même processus. Les mécanismes que nous avons mis en place pour résoudre ces problèmes sont étudiés dans la partie suivante.

4.5.4.2 Fonctionnalités particulières

Dans cette section nous allons détailler l'implantation des aspects les plus techniques de la partie générique de l'ordonnanceur en justifiant chaque choix par des aspects performance ou fonctionnalité. Cela nous permettra de présenter aussi les paramètres qui régissent le fonctionnement dynamique du régulateur et dont la présence est liée aux choix techniques d'implantation.

Envoi automatique de la charge Dans la description que nous avons faite de l'ordonnanceur nous avons précisé que l'activité moniteur ne prenait la décision de notifier aux autres processeurs sa charge locale que lorsque cette dernière avait subi une « forte » variation, cette notion étant dépendante de la politique choisie. Les informations qu'utilisent les autres processeurs pour le placement sont mises à jour à cette occasion uniquement. Cela signifie que si la charge évolue peu les informations utilisées pour ce placement peuvent être légèrement différentes, pas suffisamment pour nécessiter un équilibrage mais assez pour, peut-être, modifier une décision de placement. Dans cette éventualité il faut fournir la possibilité de réémettre cette charge à intervalles réguliers, indépendamment de sa variation. Dans ce but, un paramètre existe qui indique au bout de combien de lectures sans envoi la valeur de charge sera propagée, quelle que soit la variation. Il est évident que cette fonctionnalité peut être désactivée si la politique de régulation n'en a pas l'utilité.

La synchronisation locale Comme nous l'avons mentionné à la fin de la section précédente, il est nécessaire de synchroniser les accès à la structure de données locale qui contient l'état global de la charge. Cette structure est un simple tableau dont la taille est fixée au début de l'exécution de l'application [12]. La synchronisation est classiquement faite en garantissant l'exclusion mutuelle par l'utilisation de sémaphores. Nous avons choisi, afin d'augmenter le degré de parallélisme de la structure, d'affecter un sémaphore par case du tableau. Cela évite d'avoir trop de processus légers bloqués en attente de pouvoir accéder à la structure.

Cet aspect de la synchronisation concerne la première série d'opérations que ce service accomplit. La seconde, détection du déséquilibre et rééquilibrage, nécessite aussi la mise en place d'une synchronisation pour éviter d'avoir, par exemple, deux activités qui tentent de déplacer la même activité applicative. Cette synchronisation prend aussi la forme d'une exclusion mutuelle mise en place avec un sémaphore.

Il ne faut pas perdre de vue que l'ordonnanceur travaille dans un environnement hautement dynamique, ce qui signifie que la durée de validité d'une information peut être très courte. À cet égard, une attente trop longue sur un sémaphore peut rendre l'information caduque ou une tentative d'équilibrage inutile car la charge a déjà varié durant ce temps. Nous avons pris cette dimension temporelle en compte en utilisant des sémaphores temporisés. Ces sémaphores permettent de borner le temps d'attente sur l'opération de synchronisation (appelée P dans la littérature). Lorsque le délai est dépassé le sémaphore débloque l'activité en attente qui teste alors si la synchronisation s'est bien effectuée. Si la sortie a eu lieu à cause du délai alors elle se terminera sans rien faire de plus, pas même propager l'information de charge qu'elle contient. En effet être bloqué longtemps signifie que beaucoup d'activités d'ordonnancement sont exécutées et donc que l'application n'est pas équilibrée. Elle est donc dans une phase d'équilibrage qui implique deux conséquences : 1) la charge va varier fortement et donc 2) d'autres modifications en provenance du même processeur que l'activité vont arriver. Il est donc inutile de propager une information dépassée et qui sera remplacée très peu de temps après.

12. La reconfiguration dynamique de l'architecture parallèle, en ajoutant ou en supprimant des processeurs durant l'exécution par exemple, sort du cadre visé par l'ordonnanceur présenté.

Les deux paramètres de la régulation que sont le temps durant lequel une activité attendra pour modifier la charge globale et celui qu'elle attendra pour tenter un équilibrage sont deux paramètres qui peuvent être fixés pour une politique donnée ou modifiés dynamiquement par l'application.

La bufferisation des activités Durant les périodes d'équilibrage intensif on constate la création d'un grand nombre d'activités dont un pourcentage non négligeable ne dépasse pas le stade de la première barrière de synchronisation présentée ci-dessus. Cela est dûe au changement incessant de charge locale qui provoque des créations, rapprochées dans le temps, d'activités sur le site voisin qui se retrouve « inondé » d'activités en provenance du même site. Ce comportement est voulu par certaines politiques mais pas par d'autres qui, pour l'éviter, doivent recourir à des artifices comme la modification de la fréquence de lecture de la charge locale, ce qui n'est pas une solution très élégante et a l'inconvénient de faire remonter un problème purement technique jusqu'à un niveau conceptuel assez élevé qui est celui de la politique d'ordonnancement.

Le problème vient de la création, à des intervalles de temps très rapprochés, d'activités sur le processeur voisin dans le cas de fluctuations très fortes. Pour éliminer ce phénomène, nous avons ajouté un élément qui est chargé de « bufferiser » la création des activités sur les sites distants. Au lieu de créer directement une activité de régulation sur un processeur l'activité moniteur va insérer cette demande de création dans une structure de données locale qui sert de zone de stockage temporaire. Une activité locale à chaque processeur est chargée de prendre, à intervalle régulier, l'élément le plus ancien de cette structure et à lancer la création distante de l'activité correspondante à cette requête.

Ce mécanisme permet d'éviter l'inondation d'un processeur par des messages provenant d'un autre processeur. De plus, si un nouvel élément inséré dans la structure concerne le même site destinataire et le même site émetteur qu'un élément déjà présent il remplacera ce dernier. Donc, si une suite de notifications de modification de charge très rapprochées dans le temps doivent être envoyées vers un processeur, seule la plus récente le sera ce qui diminue à la fois la charge du réseau et le nombre de processus légers sur le site récepteur. Cela se fait au détriment de la vitesse de réaction de l'ordonnanceur, un compromis doit donc être trouvé entre ces deux caractéristiques. Il dépend des caractéristiques de l'application, de la qualité de l'ordonnancement voulue et donc de son coût.

Comme pour la précédente, cette fonctionnalité peut ne pas être utilisée par une politique si elle n'en pas besoin. De plus, le délai entre deux envois du sommet de la structure peut être modifié par la politique. Ce paramètre ainsi que celui qui règle le délai entre deux lectures de la charge locale permettent de fixer la réactivité de l'ordonnanceur à l'apparition d'un déséquilibre. Plus ils sont bas et plus l'ordonnancement sera prompt à constater un déséquilibre mais plus son coût sera élevé.

Le compteur associé au placement La mise à jour « en cascade » des informations de charge globale sur l'ensemble des processeurs, qui est dûe à la propagation des messages suivant une structure en anneau, peut poser problème si l'application est très dynamique. En effet, les messages peuvent ne pas sortir d'un voisinage réduit dans l'anneau car ils corrigent un déséquilibre et ne sont donc pas propagés. Les informations globales étant utilisées pour prendre les décisions de placement, ce placement peut être fait en utilisant des informations sur la charge de certains processeurs trop anciennes pour être pertinentes. Cela arrivera typiquement dans le cas d'un nombre important de processeurs avec de fortes variations locales, la charge sera globalement bien équilibrée mais avec des variations locales qui empêcheront la propagation des nouvelles informations aux processeurs plus éloignés. Pour résoudre ce problème, nous avons simplement ajouté à la structure de données

globales une information indiquant, pour chaque processeur, combien d'activités ont été créées depuis la dernière mise à jour de la charge sur ce processeur. Il s'agit en quelque sorte du degré de fiabilité de l'information de charge de chaque processeur. Plus elle est élevée moins fiable est l'information.

En dernier ressort, si les informations autres que locales sont toutes inutiles, la création peut se faire uniquement en local, l'ordonnanceur utilisera ensuite la migration pour la déplacer sur un processeur plus approprié. Il est clair que la politique de placement peut ignorer totalement ce degré d'obsolescence pour prendre sa décision puisqu'elle peut être redéfinie par le programmeur pour l'adapter à son application.

4.5.4.3 Paramètres de fonctionnement

Nous avons vu dans les sections précédentes comment définir un nouveau critère de charge ainsi que les opérations qui lui sont associées. Nous avons aussi détaillé certains points techniques de l'ordonnanceur qui ont été mis en place pour améliorer son efficacité. Nous avons mentionné au passage quelques paramètres régissant le comportement dynamique de l'ordonnanceur, c'est-à-dire la manière dont il réagit aux modifications de son environnement. Par cela nous entendons tout ce qui définit la manière dont l'ordonnanceur se comportera durant l'exécution de l'application et cela indépendamment de la politique choisie. Cela permet de séparer les aspects description de la politique des aspects comportement qui peuvent être modifiés dynamiquement. En particulier, le coût de l'ordonnanceur sera plus influencé par les valeurs qui sont affectées aux paramètres que par la politique mise en place. Cela est vrai aussi pour son efficacité bien entendu.

Dans cette section, nous donnons la liste des paramètres qui régissent ce fonctionnement dynamique ainsi que leur rôle (tableau 4.3).

Nom	Rôle	Intervalle
`LB_INFO_SAMPLE_TIME`	Délai entre deux lectures de la charge locale	Oui
`MAX_NUMBER_OF_PROBE_WITHOUT_SEND`	Nombre de lectures de la charge locale sans envoi de modification	Oui
`delay_before_aborting_store_operation`	Temps maximum d'attente pour stocker une nouvelle valeur de charge	Non
`delay_before_aborting_lb_operation`	Temps maximum d'attente pour effectuer un équilibrage	Non
`DELAY_BETWEEN_TWO_SENDS`	Délai entre deux créations distantes d'activités d'ordonnancement	Oui

TAB. 4.3 - *Liste des paramètres régissant le comportement de l'ordonnancement*

Tous ces paramètres peuvent être fixés au début de l'exécution mais leur valeur peut aussi évoluer durant cette dernière. Nous avons pris en compte cette particularité en laissant la possibilité au programmeur de spécifier un intervalle dans lequel certaines valeurs évolueront. La façon dont cette évolution se fait est fixée dans l'ordonnanceur et ne peut pas être modifiée. Elle est basée sur l'observation du comportement du paramètre. Par exemple, le paramètre `LB_INFO_SAMPLE_TIME` augmentera si la charge n'a pas varié entre deux lectures car cela indique une stabilité et donc que la fréquence des prises d'information peut diminuer. Au contraire si la charge varie beaucoup ce paramètre diminuera automatiquement pour prendre en compte plus rapidement les modifications de charge. Cette particularité permet d'apporter de la souplesse dans le comportement de l'ordon-

nanceur en lui donnant des capacités d'auto-adaptativité rudimentaires sans imposer cette charge supplémentaire au programmeur de la politique.

4.5.5 Intégration dans une application

Nous avons vu le fonctionnement interne de l'ordonnanceur générique ainsi que l'interface qu'il propose pour définir une nouvelle politique d'ordonnancement. L'un de ses buts principaux est de pouvoir s'intégrer à une application sans nécessiter de réécritures lourdes. En supposant que la politique d'ordonnancement existe déjà, l'application doit simplement effectuer au début de son exécution un appel à la fonction qui va initialiser l'ordonnanceur et lancer les activités de surveillance qui réguleront ensuite de manière transparente. Au stade de la conception la seule chose à modifier est la déclaration des services de l'application. Cela est dûe à la manière dont PM^2 gère cette déclaration et prend simplement la forme d'un appel à une fonction spéciale de l'ordonnanceur qui se chargera d'enregistrer les services qui lui sont spécifiques. Ces contraintes étant uniquement liées à PM^2 nous ne les détaillerons pas ici.

4.6 Exemple d'utilisation

Dans cette partie, nous allons présenter comment implanter avec l'ordonnanceur générique une politique spécifique de régulation. Cela nous permettra d'illustrer sur un exemple concret le rôle de chaque élément définissant une politique dans notre modèle. La politique choisie est volontairement très simple, le but de cet exemple étant de montrer la facilité avec laquelle on construit une nouvelle politique. Nous allons commencer par détailler le critère de régulation que nous voulons respecter, ensuite nous présenterons la manière dont ce critère est implanté et les éléments nouveaux qu'il est nécessaire de définir pour le prendre en compte. Nous illustrerons chaque élément par le code source l'implantant réellement. Enfin nous terminerons en montrant l'efficacité de cette méthode d'ordonnancement par des résultats expérimentaux.

4.6.1 Critère utilisé

Le critère que nous allons utiliser comme exemple est celui du nombre d'activités actives simultanément sur un processeur. Notre but est de conserver le même nombre d'activités applicatives actives sur chaque processeur. Nous ne pouvons bien entendu respecter ce critère que lorsque c'est possible c'est-à-dire lorsqu'il est supérieur au nombre de processeurs.

4.6.2 Implantation de cette politique

Définition de la charge Le premier élément à définir est le type de charge, donc le critère de régulation, qui sera utilisé. Ici la structure de données à utiliser est triviale puisqu'il s'agit tout simplement d'un entier contenant le nombre d'activités du processeur. Le type de la charge étant un enregistrement, il ne contiendra, dans ce cas, qu'un seul champ dont le nom est fixé par l'ordonnanceur. Le code correspondant à cette déclaration est :

```
typedef struct st_LB_INFO_CHARGE {
  int nb_activites;
} load_info;
```

Nous n'allons pas donner ici le source de toutes les fonctions qui doivent être définies pour cette structure, dont la liste est dans le tableau 4.1 car elles sont très simples et leur description alourdirait le document. Nous n'en détaillerons que deux : la lecture de la charge locale et son empaquetage car elles illustrent bien les contraintes et les avantages du lien avec PM^2.

La fonction de lecture de la charge fait tout simplement appel à une fonction de MARCEL[13] qui fournit cette information. Il s'agit d'une information de relativement bas niveau, puisqu'il faut appeler directement la bibliothèque de processus légers pour l'obtenir mais elle est pourtant d'un intérêt pratique indiscutable. Le source de cette fonction est :

```
void get_load(load_info *ou)
{
  ou->nb_activites=pthread_activethreads_np();
}
```

Et le source de la fonction empaquetant les données est[14] :

```
int pack_load_info(load_info *quoi)
{
  pm2_p_int(PM2_IN_PLACE,&(quoi->nb_activites),1,1);
}
```

Définition des fonctions de régulation Nous avons vu comment définir la charge et quelques-unes des fonctions qui lui sont associées. Nous allons maintenant étudier comment programmer les fonctions définissant la politique mise en place. Elle s'appuie bien sûr sur les informations de charge gérées par l'ordonnanceur générique. L'ensemble de ces fonctions se trouve dans le tableau 4.2, nous ne rappelons pas leurs rôles en détail ici. Nous nous contentons de décrire leur fonctionnement.

1. La fonction `significant_load_change` est celle qui indique si une variation de charge locale est suffisante pour motiver la création d'une activité de régulation sur son voisin afin de lui notifier cette modification.

   ```
   int significant_load_change(load_info *new,load_info *old)
   {
     if(abs(new->nb_activites - old->nb_activites)>1)
       return 1;
     else
       return 0;
   }
   ```

   Dans cet exemple nous avons choisi de fixer la variation de charge à un seuil strict. Si la charge a varié de plus de une activité depuis le dernier envoi, et non la dernière lecture, alors la nouvelle valeur est propagée. Dans ce cas nous avons utilisé un critère statique, indépendant du nombre d'activités, ce qui n'est pas toujours une bonne idée. Un critère relatif, tel qu'un pourcentage par exemple, est souvent plus adapté.

13. MARCEL[Nam97b] est la librairie de processus légers utilisée par PM^2.

14. Les sources données dans ce document utilisent la dernière version de PM^2 appelée PM^2-HiPerf. Elle diffère légèrement de l'ancienne version sur certains points.

2. La fonction **severe_disbalance** est aussi liée étroitement à la définition de la charge. C'est elle qui juge si un déséquilibre est suffisament important pour motiver une opération d'équilibrage. Elle s'exécute donc sur un processeur distant de celui qui a créé l'activité. Pour la politique d'exemple, elle se présente comme suit :

```
int severe_disbalance(machine_load *c1,machine_load *c2)
{
  if(abs((c1->load.nb_activites) - (c2->load.nb_activites))>1)
    return 1;
  else
    return 0;
}
```

Ici aussi nous avons choisi de définir statiquement le déséquilibre. Notons qu'avec ces deux définitions tout envoi de charge entraînera automatiquement un équilibrage. Ici aussi l'utilisation d'une différence relative (d'un pourcentage) serait plus adaptée. La structure de donnéees **machine_load** est définie par l'ordonnanceur et contient la charge (de type **load_info**) ainsi que des informations supplémentaires sur le processeur auquel elle correspond.

3. La fonction **balancing_possible** indique si un équilibrage est possible après qu'un déséquilibre a été constaté. Il existe en effet des cas où il n'est pas possible, pour des raisons applicatives (la migration doit être faite dans l'autre sens) ou des raisons de plus bas niveau (il n'y a pas de processus légers migrables). Le but de cette fonction est de prendre en compte le premier aspect et pas le second, ce qui est fait dans la partie générique comme nous le verrons dans la fonction suivante. Le source correspondant est à cette fonction est :

```
int balancing_possible(machine_load *src,machine_load *dest)
{
    if(src->load.nb_actives > dest->load.nb_actives)
      return 1;
    else
      return 0;
}
```

4. La fonction **choose_threads** est au cœur de la politique d'ordonnancement, c'est elle qui choisit les activités applicatives qui seront migrées pour rétablir l'équilibre. Elle reçoit en paramètre un tableau contenant les identifiants des activités applicatives migrables et, dans ce dernier, elle choisit celles qui seront migrées. Elle reçoit aussi la charge des processseurs qui sont impliqués dans l'opération d'équilibrage, car la valeur de ces charges est un critère prépondérant dans le choix. Ici la quantité d'activités à migrer pour équilibrer dépend bien évidemment du nombre d'activités sur chaque processeur. Le source de cette fonction pour notre exemple de critère est :

```
pthread_t *choose_threads(int *nb_choisi,int nb_candidats,
                pthread_t *candidats,machine_load *from,machine_load *to)
{
  int num_a_migrer=(from->load.nb_actives - to->load.nb_actives)/2;
```

```
  if(num_a_migrer>=nb_candidats) {
    *nb_choisi=nb_candidats;
    return candidats;
  }
  *nb_choisi=num_a_migrer;
  return candidats;
}
```

Dans ce cas il n'y a pas de choix à faire puisque seul le nombre compte. Nous verrons dans le chapitre suivant que la méthode de choix peut être nettement plus compliquée.

5. La fonction `place_thread` détermine, en utilisant les informations sur la charge globale maintenues par l'ordonnanceur, où une nouvelle activité sera créée au début de son exécution. Contrairement aux autres, cette fonction ne concerne pas le fonctionnement réactif à un déséquilibre mais au contraire l'anticipation de son apparition. Le raisonnement est le suivant : on pondère la charge de chaque machine par un coefficient d'exactitude qui est le nombre de placements locaux sur ce processeur depuis la dernière mise à jour, afin d'éviter que tous les sites n'inondent le même, et on prend celui pour lequel cette valeur pondérée est la plus faible. La variable `charge_cmin` est utilisée pour éviter de recalculer à chaque fois la valeur de la charge minimale trouvée jusqu'à présent.

```
int trouver_place_thread(machine_load *tab_charge,int card_charge,
                         machine_load *charge_locale)
{
  int i;
  int indice_cmin,indice_change_min;

  if((card_charge==0) || (tab_charge==NULL))
    return charge_locale->num_machine;      /* Placement local */

  indice_cmin=0;
  charge_cmin=tab_charge[0].load.nb_activites + tab_charge[0].number_of_change;

  for(i=1;i<card_change;i++)
    if((tab_charge[i].load.nb_activites+tab_charge[i].number_of_change) <
        charge_cmin) {
      indice_cmin=i;
      charge_cmin=tab_charge[i].load.nb_activites+
                  tab_charge[i].number_of_change;
    }

  if(charge_cmin<charge_locale->load.nb_activites)
    return tab_charge[indice_cmin].num_machine;
  else
    return charge_locale->num_machine;
}
```

Notons que cette fonction n'est pas directement appelée par l'application. Cette dernière appelle un point d'entrée de l'ordonnanceur qui se charge, en plus d'appeler `trouver_place_thread`

de mettre à jour les informations concernant le nombre de fois où chaque site a été désigné comme cible d'une nouvelle activité depuis la dernière mise à jour de la charge. Cette information est contenue dans le champ `number_of_change` qui est utilisé dans la fonction ci-dessus. Rappelons que ce placement n'est pas définitif et qu'un mauvais choix dans cette étape aura « simplement » pour conséquence de provoquer des opérations d'équilibrage dans la suite de l'exécution.

Nous avons vu comment mettre en place une politique d'ordonnancement qui utilise le nombre d'activités de chaque processeur comme critère de régulation. Nous allons maintenant regarder l'efficacité de cette politique de régulation sur une application témoin dont les caractéristiques sont proches de celles que nous étudions.

4.6.3 Résultats expérimentaux sur une application témoin

Dans cette section nous allons étudier l'implantation du régulateur sur une application ayant un comportement similaire à celles visées. Elle génère un nombre fixé d'activités à des moments aléatoires, chaque activité ayant aussi une durée aléatoire. La politique d'ordonnancement utilisée est celle décrite dans la section précédente, elle tend à équilibrer le nombre d'activités s'exécutant sur chaque processeur. Chaque activité n'est qu'une boucle vide, effectuée un certain nombre aléatoire de fois. Le délai écoulé entre deux créations d'activités est lui aussi aléatoire ce qui rend l'application « doublement » irrégulière. En terme de création et de durée d'activités elle a donc un comportement semblable aux applications visées bien qu'elle soit à travail constant, ce qui permet d'évaluer le coût réel de l'ordonnancement.

Nous allons étudier les performances que procure l'ordonnanceur pour cette application en faisant varier différents paramètres afin de constater leur rôle dans le comportement de l'ordonnanceur et en particulier dans son coût.

L'architecture parallèle sur laquelle ces tests ont été menés est une ferme de processeurs ALPHA à 133MHz disposant de 64Mo de mémoire et reliés par un giga-switch. La version de PM^2 utilisée est la version Hi-Perf[NM97] qui utilisait directement le protocole de communication TCP.

Nous avons représenté sur la figure 4.7 l'accélération obtenue sur cette application avec l'ordonnanceur générique utilisant la politique présentée. Les trois courbes représentent différentes fréquences de lecture de la charge.

On constate sur ces courbes que la fréquence ne joue pas un rôle très important dans les performances puisque les trois courbes ont un comportement similaire pour un petit nombre de processeurs. Les résultats décroissent par contre quand le nombre de processeurs augmente. Ce comportement est logique et correspond aux résultats théoriques trouvés quant au coût de l'ordonnancement. En effet, le nombre d'activités applicatives reste constant quel que soit le nombre de processeurs alors que le nombre d'activités d'ordonnancement augmente. Le coût relatif de l'ordonnancement par rapport à l'activité augmente donc ce qui induit logiquement une baisse des performances. Pour vérifier ce phénomène nous avons exécuté l'application dans les mêmes conditions mais en faisant exécuter deux cents activités au lieu de cent. Les résultats sont présentés sur la figure 4.8.

On constate expérimentalement cette propriété, plus le nombre d'activités augmente et moins l'ordonnancement coûte cher en valeur absolue puisque l'accélération augmente. Dans les cas extrêmes où le nombre d'activités est très grand par rapport au nombre de processeurs le coût d'ordonnancement sera alors négligeable.

Un résultat étonnant montré par ces deux courbes est le peu d'impact de la fréquence de lecture de la charge sur les performances de l'application. En fait elle influe bien mais le coût qu'elle induit

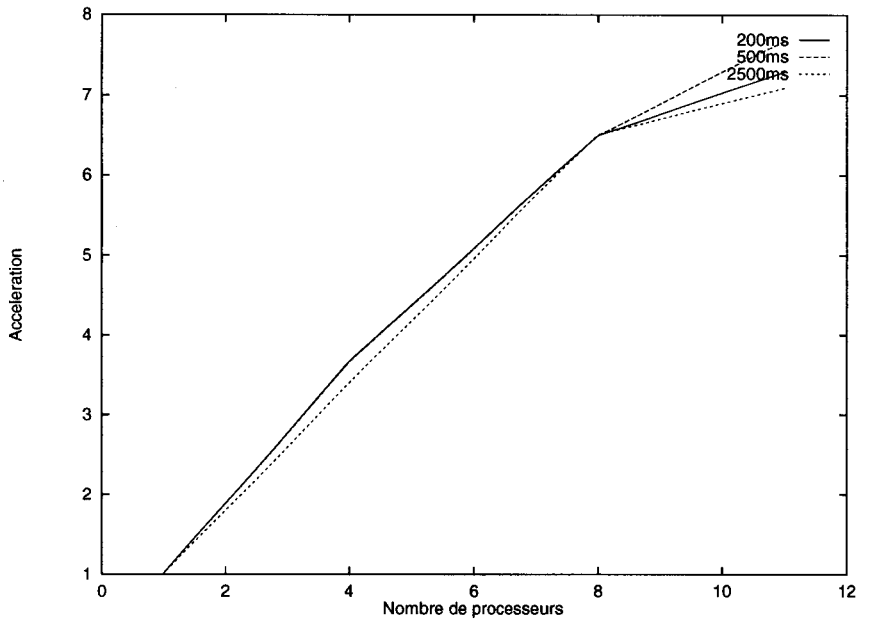


FIG. 4.7 - *Accélération obtenue avec l'ordonnanceur pour une application synthétique de cent activités*

par la création d'activités supplémentaires est compensé par le gain en qualité d'équilibrage. Il en résulte donc un équilibre qui fait que les performances globales de l'application restent proches. Les meilleurs résultats sont cependant obtenus dans le second cas avec la fréquence la plus élevée, même pour cette application qui ne comporte pas un grand nombre d'activités. il semble donc plus intéressant de choisir une fréquence élevée en faisant quasiment travailler l'ordonnanceur en permanence. Ce choix résulte toujours du compromis qui est au cœur de la conception de notre ordonnanceur entre la qualité de l'ordonnancement et son coût. Le choix dès le départ d'un placement spéculatif des activités nous condamne de manière quasi automatique à ne pas obtenir de performances optimales, c'est-à-dire dans ce cas une courbe d'accélération qui soit linéaire. Nous pensons qu'atteindre cette optimalité du placement est utopique car son calcul serait trop coûteux pour être intéressant.

4.7 Conclusion

Dans ce chapitre nous avons étudié notre proposition pour un ordonnanceur générique destiné aux applications hautement irrégulières. Pour cela, nous avons étudié les besoins en terme d'ordonnancement des applications que nous visons. Nous en avons déduit les caractéristiques que devait satisfaire l'ordonnanceur générique : fonctionnement concurrent et continu en parallèle avec l'application, coût bornable, transparence à la conception et à l'exécution et qualité de l'ordonnancement paramétrable ce qui permet de prendre en compte la dimension spéculative de l'ordonnancement.

Nous avons ensuite décrit notre proposition d'ordonnanceur générique en mettant l'accent sur sa souplesse qui permet de l'apdater simplement à de nouvelles politiques et à de nouveaux comportements. Enfin, nous avons présenté son utilisation pour implanter une politique basée sur le nombre d'activités présentes sur chaque processeur.

Dans le chapitre suivant nous allons présenter une autre politique d'ordonnancement implantée

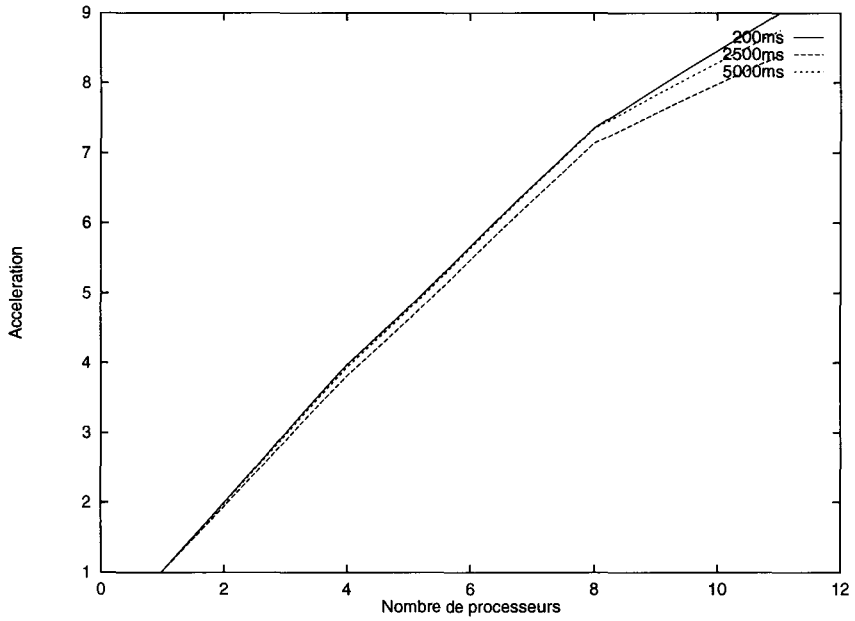


FIG. 4.8 - *Accélération obtenue avec l'ordonnanceur pour une application synthétique de deux cents activités*

avec ce régulateur qui utilise comme critère les priorités des activités applicatives.

Chapitre 5

Priorités et applications irrégulières distribuées

Dans ce chapitre nous allons revenir sur les applications étudiées dans le chapitre 2 mais dans le contexte du parallélisme à grain fin et de la virtualisation. Le chapitre 3 présentait les grandes lignes de la parallélisation de ce type d'applications. Nous l'étudierons plus en détail dans ce chapitre avec application de Branch&Bound qui résout le classique problème du voyageur de commerce. L'angle d'étude sera celui des priorités qui représente une approche novatrice pour ce domaine applicatif. Il consiste à fixer des priorités aux activités applicatives afin d'accentuer le caractère spéculatif de l'exploration arborescente en favorisant dynamiquement certaines activités qui semblent plus prometteuses que les autres.

Nous commençons par rappeler dans quel but les priorités d'activités ont été introduites à l'origine dans un cadre général puis dans le cadre du parallélisme à grain fin. Nous concluons cette partie avec l'utilisation des priorités dans le contexte des applications irrégulières que nous étudions. Nous regardons les performances qu'elles peuvent apporter dans un contexte local.

L'utilisation de priorités applicatives dans le parallélisme à grain fin se fait, originellement, dans un contexte purement local. Le cadre que nous visons étant distribué, nous devons en étudier les conséquences sur le respect des priorités. C'est l'objet de la section suivante où nous montrons que leur utilisation directe n'est pas possible et nécessite la mise en place d'un mécanisme de régulation sur les activités. Nous montrons comment la construction d'un tel mécanisme est simple en utilisant le régulateur GTLB présenté dans le chapitre précédent.

5.1 Présentation des priorités

Nous allons dans cette section présenter dans quel but la notion de priorité a été introduite à l'origine dans les systèmes informatiques puis nous présenterons son utilisation dans un cadre totalement différent qui est celui des applications irrégulières. Ces deux études nous permettront d'introduire l'utilisation des priorités dans le contexte des applications à grain fin qui est celui de ce travail.

5.1.1 Dans les systèmes d'exploitation

Dans les systèmes d'exploitation le mécanisme de priorité a été introduit afin de pouvoir favoriser certains processus par rapport à d'autres. Il revêt différentes significations suivant les systèmes,

celles-ci dépendant de la politique d'ordonnancement *locale* implantée dans le système. On identifie deux familles principales :

FIFO Les processus sont stockés dans un ensemble de files, chaque file correspondant à une valeur de priorité. Le premier processus dans la file la plus prioritaire est exécuté quand le processeur est disponible, c'est-à-dire lorsque le processus en cours se termine ou se bloque.

Round-Robin [1] Les processus s'exécutent en temps partagé, suivant un système proportionnel et préemptif. Les processus sont stockés dans une file et disposent du processeur chacun à leur tour. Le temps est divisé en quanta, chaque processus dispose d'un certain nombre de quanta, proportionnel à sa priorité, pendant lesquels il s'exécute sur le processeur. Lorsqu'il les a épuisés le suivant dans la file est exécuté. Lorsque tous les processus prêts dans la file ont épuisé leurs quanta, le cycle est recommencé au premier de la file.

La première famille n'est maintenant plus utilisée dans les systèmes modernes qui disposent tous de la multiprogrammation. La seconde est la plus répandue car elle offre le meilleur rapport coût/efficacité. Elle est simple à mettre en place, peu coûteuse et procure un bon comportement réactif tant que le système n'est pas trop chargé[Tan92, SG94]. Certains ordonnanceurs font aussi décroître la valeur de priorité au cours du temps afin de diminuer l'influence d'un processus très long sur le temps d'exécution des autres, à durée de vie plus courte.

Cette notion de priorité est étroitement liée au système sous-jacent et n'a pas beaucoup de sens au niveau applicatif. Nous allons maintenant étudier une notion qui est intimement liée à l'application et qui lui est propre, qui n'a pas d'existence hors d'elle, à savoir l'utilisation des priorités dans les applications hautement irrégulières.

5.1.2 Dans les applications irrégulières

Nous avons vu dans les chapitres précédents le rôle que joue l'ordre dans lequel les explorations de sous-espaces sont faites dans le coût total de résolution du problème. C'est pour cette raison que ce coût est fonction de l'ordonnancement des activités applicatives. Intuitivement, cela signifie que certaines activités sont plus importantes que d'autres et qu'elles doivent être traitées avant les autres. Nous l'avons exprimé au niveau exécutif, en terme d'activités plus prioritaires, mais cela correspond au niveau applicatif à des sous-arbres, ou des sous-espaces, plus importants que d'autres.

Dans les implantations décrites dans le chapitre 2, cette priorité était prise en compte par l'utilisation de files de priorité. Ces structures stockent des données auxquelles sont associées une valeur de priorité. Elles offrent une opération spéciale : l'extraction du minimum (ou selon le cas du maximum) c'est-à-dire de la donnée à laquelle est associée la plus petite (ou la plus grande) valeur de priorité. Lorsque ces applications sont construites sur le modèle maître/esclave, un esclave va chercher du travail en demandant le nœud le plus prioritaire à la structure de données partagée.

La valeur de priorité associée à un sous-arbre est calculée en utilisant une heuristique. Ce calcul est généralement basé sur une évaluation de la valeur de la solution partielle représentant la racine de ce sous-arbre. Cette valeur peut éventuellement être pondérée par la profondeur correspondant à la solution partielle afin d'homogénéiser les valeurs pour l'ensemble de l'arbre de recherche. Cette valeur étant calculée par une heuristique, il est clair qu'elle n'est qu'une spéculation sur l'importance respective du sous-arbre qu'elle value. L'indéterminisme du comportement se retrouve aussi dans cet aspect du problème.

1. Tourniquet en français.

5.2 Priorités et parallélisme à grain fin

Les priorités sont affectées aux processus légers afin de les classer par ordre d'importance comme pour les processus lourds. Mais, bien entendu, la granularité est différente; il peut y avoir des dizaines de milliers de processus légers dans un processus alors qu'il est rare de trouver une machine UNIX avec plus de mille processus lourds.

La façon dont les valeurs de priorités sont affectées aux processus légers varie suivant les politiques d'ordonnancement locales du support qui implante les threads. Nous allons en étudier deux : celle de SOLARIS[Sun88, Sun93] et celle de MARCEL qui est au centre du support d'exécution PM^2.

5.2.1 Priorités dans Solaris

Le système SOLARIS gère les priorités suivant le principe du round-robin à tourniquets multiples. Il définit un nombre fixé de niveaux de priorité et met en place pour chacun de ces niveaux un tourniquet contenant les processus prêts à être exécutés. Ces processus sont ensuite exécutés en temps partagé de manière préemptive suivant la méthode *round-robin* que nous avons déjà présentée dans la section consacrée aux processus lourds.

La conséquence de cette approche est que les processus de basse priorité ne peuvent s'exécuter que lorsque plus aucun processus ayant une forte priorité n'est prêt. On ne trouve pas de notion d'équité, ou de concurrence pour suivre la terminologie de la virtualisation, dans ce système. Le partage de temps n'est équitable qu'entre les processus de même priorité et cela peut mener à des situations de famine si un processus de haute priorité est bloqué dans une boucle infinie.

5.2.2 Priorités dans MARCEL

La gestion des priorités dans MARCEL suit une approche différente de celle présentée auparavant. Au centre de cette approche on trouve le principe d'équité qui veut que tous les processus légers puissent s'exécuter concurremment sans risque de famine tout en gardant la possibilité d'en privilégier certains sur d'autres. Notons que ce partage du temps processeur entre les processus légers est purement local, il ne concerne que les processus légers situés sur le même processeur. Nous verrons dans la suite que l'extension de cette notion de priorités à un contexte distribué n'est pas triviale et pose de nombreux problèmes.

L'ordonnancement local est préemptif avec priorités. Il est géré suivant la méthode *round-robin* avec tous les processus légers prêts à être exécutés dans une file unique. Pour chaque tour du tourniquet chaque processus léger dispose d'un nombre de quanta de temps processeur proportionnel à sa priorité. Quand ce quota est épuisé, le processeur est affecté au processus suivant dans la liste. Classiquement, lorsque la fin de la liste est atteinte, un nouveau tour de liste commence.

Par exemple si deux processus légers de priorité respective dix et un sont dans le tourniquet, l'un disposera de dix quanta de temps processeur pendant que l'autre ne disposera que d'un seul. Nous avons représenté ce fonctionnement sur la figure 5.1 qui illustre comment le processeur est partagé entre les processus légers.

La quantité réelle de temps processeur affectée à chaque processus léger à chaque tour du tourniquet est, par conséquent, dépendante du nombre et des priorités des autres processus légers présents dans le même processus lourd. Seuls les processus légers prêts à être exécutés rentrent en compte dans ce calcul, ceux qui sont bloqués, typiquement sur une opération de synchronisation, ne consommant pas de temps processeur. Cette politique d'ordonnancement local garantit donc que pour deux processus légers p et p' de priorités respectives *priorité*(p) et *priorité*(p') p s'exécutera

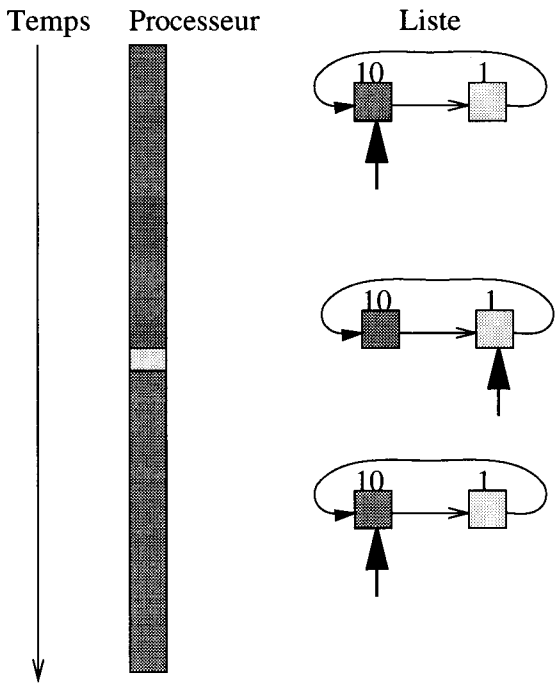


FIG. 5.1 - *Affectation du processeur à deux processus légers, un de priorité un et un de priorité dix*

$\frac{priorité(p)}{priorité(p')}$ plus vite que p'. Cette propriété est vraie à tout instant[2] comme nous allons le montrer.

Avec cette politique d'ordonnancement local, il est possible de définir la vitesse à laquelle chaque processus léger s'exécute. Cette vitesse correspond au *pourcentage de temps processeur* qui lui est affecté. Cette vitesse absolue $v(p)$ s'exprime avec la formule suivante :

$$v(p) = \frac{priorité(x)}{\sum_{y \in \{processus\ actifs\ sur\ le\ processeur\}} priorité(y)} \tag{5.1}$$

À l'aide de cette formule on peut vérifier la propriété énoncée ci-dessus :

$$\forall\ p, p' \in \{processus\ actifs\}, \frac{v(p)}{v(p')} = \frac{priorité(p)}{priorité(p')} \tag{5.2}$$

L'ordonnanceur de MARCEL se caractérise donc par une prise en compte de tous les processus légers, sans risque de famine, et un partage inégal du temps processeur entre eux, qui est conditionné par leur valeur de priorité. Ces deux caractéristiques répondent aux besoins de la virtualisation et des applications hautement irrégulières : la concurrence est nécessaire pour la virtualisation et la mise en place d'une hiérarchie entre les processus légers l'est pour répondre aux besoins des applications comme nous allons le voir maintenant.

2. Modulo un tour de tourniquet bien sûr.

5.2.3 Utilisation des priorités exécutives dans les applications irrégulières

La description des applications irrégulières que nous avons étudiées nous a montré qu'il était important de pouvoir avantager l'exploration de certains sous-espaces sur d'autres afin de favoriser l'apparition « d'anomalies d'accélération » [LW86, Man92]. Dans les implantations de tels algorithmes , cette caractéristique était prise en compte en explorant prioritairement certains sous-espaces lorsqu'un processeur était disponible. Un sous-arbre très prioritaire n'est donc pas pris en compte au moment où il a été « engendré » mais plus tard dans l'exécution. Cette contrainte est dûe, comme nous l'avons vu, à la différence entre les granularités applicative et exécutive.

Avec les processus légers cette contrainte n'existe plus mais, dans les modèles que nous avons étudiés jusqu'à présent, toutes les activités sont parfaitement égales et la notion de priorité n'est pas prise en compte. La mise à disposition du programmeur de la possibilité d'affecter des priorités aux processus légers permet de faire correspondre à cette *priorité applicative* une *priorité exécutive.* L'utilisation des priorités dans un algorithme de type Branch&Bound est illustrée sur la figure 5.2. Sur cette figure sont représentés deux sous-arbres, chacun étant exploré par un processus léger. Sur la figure de droite les deux processus légers ont la même priorité, lorsqu'une meilleure solution est trouvée dans le sous-arbre de droite, elle permet de couper une petite portion du sous-arbre gauche car le reste a déjà été exploré. Sur la figure de gauche, au contraire, le processus léger a une plus grande priorité et la découverte de la nouvelle solution permet donc de couper une plus grande partie du sous-arbre de gauche.

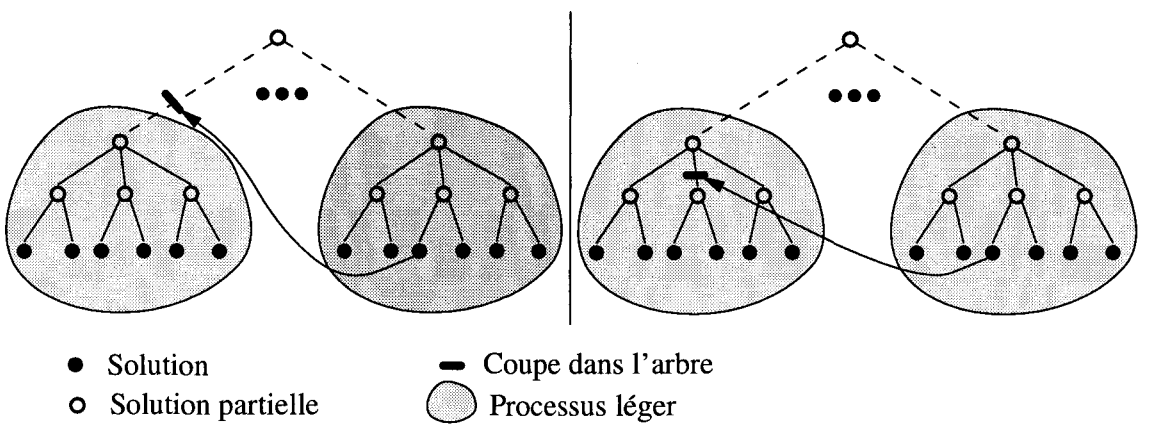


FIG. 5.2 - *Exemple d'exploration spéculative d'un arbre de recherche*

Dans les applications qui nous intéressent, les priorités sont utilisées pour favoriser les activités qui explorent les sous-arbres les plus intéressants. Ce favoritisme peut être fait dans deux buts :

1. Dans le cas d'une exploration exhaustive, la découverte d'une nouvelle solution permet de diminuer l'espace total à parcourir et donc le coût de recherche de la solution optimale.

2. En favorisant les sous-arbres les plus intéressants, il permet de trouver une bonne solution plus tôt dans l'exécution. C'est dans ce but que sont utilisées les méthodes dites non-exactes, comme la recherche tabou ou les algorithmes génétiques, les méthodes exactes étant considérées comme trop coûteuses pour cet usage. En utilisant les priorités pour favoriser la découverte d'une bonne solution, dont on ne cherche pas à prouver l'optimalité, elles peuvent être une alternative aux méthodes heuristiques.

Le comportement des problèmes abordés étant irrégulier, favoriser une activité par rapport à une autre est un choix qui est basé sur un calcul fait à partir des données disponibles, qui sont partielles. Les données complètes pour prendre une décision seraient le coût que représente chaque activité et le gain que son exécution procurerait[3]. Cela introduit donc un degré de spéculation supplémentaire dans l'algorithme d'exploration qui est dû cette fois au choix de la méthode de calcul des priorités. Ce calcul est intimement lié à l'application, il est impossible de donner une approche générale. Nous allons détailler les paramètres rentrant en compte dans ce choix pour les applications que nous avons étudiées et donner des pistes pour un raisonnement plus global.

5.2.4 Comment affecter une priorité à une tâche?

Dans cette section nous allons donner des pistes sur la manière de fixer la priorité des activités. Cette décision étant étroitement liée aux applications, il est impossible de généraliser notre discours à l'ensemble des applications irrégulières. Nous nous focaliserons donc sur les applications sur lesquelles portent notre étude, c'est-à-dire celles d'exploration arborescente suivant la méthode Branch&Bound.

Comme nous l'avons signalé, l'affectation d'une priorité est une spéculation sur l'importance de l'activité car les données sur lesquelles cette décision est basée ne représentent forcément qu'une vue partielle. Les méthodes utilisées pour calculer cette priorité sont donc heuristiques. Dans le cas général, indépendamment du problème traité, de l'exploration arborescente, les informations disponibles pour calculer l'importance d'une activité au moment de son lancement sont :

1. la valeur de l'évaluation partielle du nœud du sous-arbre formant la racine du sous-arbre que l'activité va explorer,

2. l'endroit où se trouve ce sous-arbre dans l'arbre global, c'est-à-dire, en pratique, la profondeur dans l'arbre du nœud racine du sous-arbre.

Ces informations paraissent bien pauvres pour prendre une bonne décision mais elles sont en fait riches d'enseignement. En effet, intuitivement, si un nœud a une valeur proche de la meilleure solution alors qu'il est à mi-hauteur de l'arbre, il y a très peu de chances qu'il amène à une bonne solution et sa priorité doit donc être basse. Au contraire, un nœud proche du sommet de l'arbre et ayant une évaluation peu élevée[4] doit être privilégié.

Détaillons maintenant différentes approches pour utiliser ces deux valeurs afin de calculer la priorité d'une activité :

1. **Privilégier la profondeur** Affecter la priorité en ne tenant compte que de la profondeur. Cette approche revient à privilégier les activités qui explorent « le fond » de l'arbre, elle trouvera un grand nombre de solutions qui convergeront progressivement mais pas, de manière générale, vite vers la meilleure solution.

2. **Privilégier l'évaluation** Affecter la priorité en ne tenant compte que de l'évaluation. Au contraire du précédent, cela revient à privilégier les activités qui explorent le haut de l'arbre car leur évaluation est plus faible que celles s'occupant du bas. Cela retardera donc le moment où des solutions seront trouvées ce qui rend cette méthode de calcul peu intéressante aussi.

3. Si ces données étaient connues, l'algorithme ne serait plus irrégulier et la méthode que nous utilisons ne serait plus adaptée de toute façon ...

4. Tout au long de cette discussion nous nous placerons dans le cas d'une minimisation mais le raisonnement est bien évidemment identique pour une maximisation.

3. **Pondérer l'évaluation par la profondeur** Affecter la priorité en pondérant une valeur par l'autre. Pour éviter les phénomènes décrits ci-dessus on calcule, en utilisant la meilleure solution connue jusqu'à présent, si le sous-arbre risque d'amener à une meilleure solution. Le raisonnement que l'on utilise est le suivant :

   - Calcul de la valeur pondérée correspondant à la meilleure solution trouvée: $v_{opt} = solution \times \frac{profondeur}{taille\ du\ problème}$
   - Si cette valeur est supérieure à l'évaluation du nœud racine alors la priorité est forte,
   - sinon elle est faible.

   La valeur réelle de la priorité est ensuite proportionnelle à la différence entre v_{opt} et l'évaluation du nœud racine de l'activité concernée.

   Cette heuristique met toutes les activités sur un pied d'égalité, par rapport à la profondeur, contrairement aux deux précédentes. Cependant elle ne prend pas en compte l'aspect progressif de la recherche qui veut que la différence entre certains niveaux sera faible alors que celle entre d'autres sera forte. Cette particularité est liée à la construction de la solution et à la structure des données mais on retrouve ce phénomène sur de nombreux problèmes réels. En linéarisant le calcul de la priorité cette dimension, qu'on peut qualifier de phénomène de saut des valeurs entre les niveaux, n'est pas prise en compte. Nous verrons dans la suite si cela a ou non un impact significatif sur les performances.

4. **Utilisation d'un « vecteur de priorités »** Dans l'approche précédente, il y a linéarisation des valeurs alors que souvent, dans les problèmes réels, la progression se fait par « bonds » de valeurs et non pas de manière continue. Pour répondre à ce problème, nous proposons d'utiliser un vecteur contenant, pour chaque niveau, la priorité la plus faible rencontrée durant l'exploration. La valeur de la priorité d'une activité est alors calculée en comparant la valeur de l'évaluation partielle de sa racine avec la valeur située à la même profondeur dans le vecteur de priorités.

Notons que les propriétés des propositions 1) et 2) peuvent être inversées en échangeant le rôle du critère, nous avons décrit l'utilisation typique qui en est faite. On peut considérer qu'elle sont plus ou moins équivalentes car leur comportement est très similaire.

Dans la méthode 4), plus on approche de la fin de l'exécution et plus la priorité est fixée de manière sûre car les informations utilisées pour le calcul sont de plus en plus vérifiées : si un grand nombre de branches ont déjà été explorées, la valeur de chaque élément du vecteur de priorités est de meilleure qualité qu'au début de l'exécution. Donc, si un nœud a une valeur supérieure à celle du vecteur il doit être fortement avantagé. Ce n'est pas le cas dans les stratégies 1) et 2) où la connaissance glanée durant l'exécution ne rentre à aucun moment en ligne de compte. Elle est par contre utilisée dans la méthode 3) mais pas de manière aussi poussée.

Cependant, la méthode 4) induit un coût supplémentaire car elle oblige à maintenir le vecteur de priorités. Indépendamment des aspects liés à la distribution de ce vecteur sur les processeurs, que nous aborderons dans la section suivante, sa gestion a un coût local. En effet à chaque nouvelle évaluation d'un nœud il faut regarder s'il n'améliore pas la meilleure valeur trouvée jusqu'à présent et, le cas échéant, modifier le vecteur. Ce coût peut sembler négligeable mais le nombre d'opérations faites pour l'exploration d'un nœud est typiquement très faible et en rajouter quelques-unes peut avoir avoir des conséquences non négligeables.

Nous avons vu dans cette section comment affecter une priorité à une activité dans les applications visées. Cette opération est très dépendante de l'application sur laquelle elle s'appliquera, il est donc difficile de donner des solutions qui fonctionneront dans tous les cas. Nous avons présenté les caractéristiques des quatre approches les plus intuitives mais nous ne nous sommes pas prononcé sur les performances qu'elles induisent car une telle généralisation n'est pas possible. Nous présentons dans la section 5.2.5.2 les résultats que nous avons obtenus avec ces heuristiques sur le problème que nous avons étudié, celui du voyageur de commerce.

Heuristiques spécifiques à un problème donné Dans la section précédente nous avons présenté des heuristiques permettant de calculer la priorité suffisamment générales pour être utilisées avec n'importe quelle application d'exploration arborescente. La généralité de ces méthodes les rend intéressantes pour les applications dans lesquelles on ne peut extraire que peu d'informations à partir des données, comme le voyageur de commerce, et pour des applications dont on ignore les caractéristiques.

Pour certaines applications, il est possible de développer des heuristiques adaptées et spécifiques dont la validité des choix est plus importante que pour des heuristiques génériques. C'est le cas, par exemple, de la méthode dite de « potentialité » développée par Salah Dowaji dans [Dow95, DR95] dont l'originalité justifie une étude plus approfondie.

La potentialité d'un nœud de l'arbre d'exploration se définit comme suit :

$$Potentialité(n) = (dist(n, Fin), travail\_potentiel, nb\_ancêtres)$$

Dans cette formule *dist(n,Fin)* représente la position du nœud dans l'arbre de recherche et *nb_ancêtres* est utilisé pour différencier deux nœuds ayant les mêmes caractéristiques. La troisième valeur, le *travail_potentiel* fait l'originalité de l'approche. Elle représente la valeur du travail potentiel associé à n, donc elle indique le coût que la résolution du sous-problème correspondant au nœud représente. Cette valeur est malheureusement entièrement dépendante du problème et n'est donc pas une propriété utilisable de manière générale. Ainsi dans le cas que nous allons étudier ensuite, le voyageur de commerce, cette valeur n'est pas calculable à cause de la structure des sous-problèmes.

Cette valeur de potentialité est donc utilisable pour affecter une priorité qui prend en compte la quantité de travail associée à l'exploration d'un sous-arbre. Il est ainsi possible de mettre en place un ordonnancement qui pénalise les activités coûteuses et dont l'évaluation partielle est défavorable, ce qui en augmente la qualité. Cet ordonnancement utilisera la valeur de priorité des activités et n'induit donc pas de coût supplémentaire par rapport à un ordonnancement basé sur la profondeur, mis à part celui du calcul de la priorité que l'on peut négliger.

5.2.5 Application sur le problème du voyageur de commerce

Nous allons, dans cette section, présenter un exemple d'utilisation des priorités dans un cadre purement local, c'est-à-dire non distribué, sur une application réelle. Cette étude nous permet de mettre en évidence le bénéfice du parallélisme à grain fin et des priorités sur une application sans exploiter de parallélisme réel. Son but est de justifier notre approche avant de l'appliquer sur une architecture distribuée.

Pour ce faire nous avons programmé l'application très connue du voyageur de commerce[5] en utilisant du parallélisme à grain fin sur un seul processeur. Nous ne décrivons pas ici la problèmatique, le lecteur la trouvera dans l'annexe A.3. Dans cette section, nous étudierons aussi les

5. Que nous nommerons TSP (pour Traveling Salesperson Problem) dans la suite du document.

performances procurées par les différentes heuristiques de calcul de la priorité présentées dans la section précédente sur une application réelle.

5.2.5.1 Description de l'implantation

L'implantation choisie effectue un découpage dynamique de l'exploration de l'arbre en sous-arbres, donc en créant des activités nouvelles, à la demande suivant le degré de parallélisme souhaité par le programmeur. Ce degré est un paramètre de l'exécution du programme et son choix influe bien évidemment sur le découpage, et donc sur la granularité applicative. Ce choix peut paraître en contradiction avec le précepte que nous énonçons dans le chapitre 2 selon lequel la granularité applicative devait être sous la responsabilité de l'application. Il n'en est rien car la manière dont le découpage est fait est entièrement dirigé par l'application ainsi que le degré de parallélisme qui n'est pas une contrainte dûe au support mais bien un paramètre comme un autre de l'exécution.

Pour des raisons d'efficacité, et parce que le grain manipulé efficacement par le support n'est quand même pas l'instruction, nous avons choisi de ne créer une nouvelle activité que lorsque le nœud racine du sous-arbre correspondant à cette activité n'était pas trop profond. En effet, comme il est peu efficace de créer une nouvelle activité pour explorer seulement dix nœuds par exemple, cette caractéristique est un garde-fou pour éviter de tels comportements.

À la création de chaque activité, une priorité lui est affectée qui dépend de la méthode choisie. Nous avons détaillé les quatre principales dans la section précédente, nous n'y reviendrons donc pas ici.

Notre but avec cette application n'est pas de trouver la nouvelle heuristique qui révolutionnera le monde du TSP mais de valider notre utilisation des priorités sur une application réelle. En conséquence nous avons choisi un algorithme de parcours classique et « naïf », en profondeur d'abord, ainsi qu'une fonction d'évaluation partielle qui ne fait aucune spéculation, minimalisation ou maximalisation, sur le domaine restant à explorer. En restant dans le cas le plus général possible nous souhaitons obtenir des résultats les plus généraux possibles.

5.2.5.2 Résultats expérimentaux

Toujours dans le même but, nous avons décidé d'utiliser notre application de TSP sur des problèmes générés aléatoirement au lieu d'utiliser les problèmes classiques de la littérature. Nous avons exécuté notre application sur un ensemble de problèmes caractérisés par la valeur qui sert à initialiser le générateur de nombres aléatoires qui décriront les données du problème. Nous avons résolu chaque problème avec un nombre maximal d'activités différentes variant de un (correspondant donc à une exécution séquentielle) à mille. Pour chaque exécution nous avons utilisé les quatre heuristiques différentes pour calculer la priorité.

Nous avons choisi de présenter le résultat le plus important qui est le nombre de nœuds développés pour chaque problème en pourcentage de l'exécution séquentielle. Dans le contexte de ces essais, monoprocesseur, le temps d'exécution est une conséquence directe de cette valeur.

Nous avons effectué des tests de cette implantation du problème du TSP sur un PC sous LINUX doté d'un Pentium à 133MHz et de 48Moctets de mémoire. Les tests ont été effectués sur des problèmes générés aléatoirement de taille douze et treize villes. Les résultats sont synthétisés dans les tableaux 5.1 et 5.2 qui correspondent à des tailles de problème différentes.

Ces résultats mettent en évidence la supériorité de l'heuristique qui utilise le vecteur des meilleurs solutions partielles. Cela n'est pas surprenant car c'est l'heuristique qui utilise la plus grande quantité d'informations donc qui est à même de faire le meilleur choix. Cela n'est cependant

Heuristique \ Nombre d'activités	1	10	50	100	1000
Profondeur	100	83	82	82	81
Valeur	100	84	83	83	82
Pondéré	100	84	83	83	82
Vectoriel	100	86	82	80	**75**

TAB. 5.1 - *Pourcentage du nombre de nœuds développés en fonction de l'heuristique et du nombre de processus légers pour cent problèmes générés aléatoirement du TSP de taille 12*

Heuristique \ Nombre d'activités	1	10	50	100	1000
Profondeur	100	93	82	82	82
Valeur	100	94	89	89	83
Pondéré	100	92	90	90	83
Vectoriel	100	97	81	80	**76**

TAB. 5.2 - *Pourcentage du nombre de nœuds développés en fonction de l'heuristique et du nombre de processus légers pour cent problèmes générés aléatoirement du TSP de taille 13*

vrai que lorsque le degré de parallélisme est important, dans le cas contraire les autres heuristiques sont supérieures ou équivalentes. Cela est explicable par le fait que trop peu d'explorations sont exécutées en parallèle pour que l'influence des priorités soient décisives. De plus les informations utilisées pour mettre à jour le tableau, et donc la justesse de l'heuristique, mettent plus de temps à être judicieuses lorsque le degré de parallélisme est moins élevé. Cette caractéristique est une conséquence de la stratégie de parcours et de découpe en activités que nous avons implantée et n'est pas vraie dans le cas général (avec n'importe quelle stratégie de parcours).

L'utilisation d'un vecteur de priorités dans un cadre local comme celui de ces expérimentations ne pose pas de problèmes particuliers. Il n'en est pas de même dans un cadre distribué où le maintien d'une *valeur globale*, puisqu'il s'agit de cela, n'est pas trivial. C'est l'un des problèmes que nous devons résoudre lors du passage de cette méthode de résolution basée sur les priorités vers un tel contexte. Le second, et le plus ardu, est celui de l'extension de la sémantique des priorités, qui est locale, vers un environnement distribué. Notre proposition pour atteindre ce but fait l'objet de la section suivante.

5.3 Priorités et distribution

Jusqu'à présent dans ce chapitre nous avons vu les priorités sous un angle purement local, monoprocesseur. Le contexte que nous visons étant celui des applications distribuées, nous nous devons d'étendre cette notion de priorité vers ce contexte. Nous allons dans cette section étudier ce passage, les problèmes qu'il pose ainsi que la solution que nous préconisons. *Ce travail a été présenté dans [DNGM96, Den97].*

5.3.1 Présentation du problème

Notre travail se situe dans le cadre d'ESPACE dont la base est le système PM^2. La notion de priorité locale que nous utilisons est donc celle définie dans l'ordonnanceur de MARCEL. Elle a été

présentée dans la section 5.2.2 page 119.

5.3.1.1 Définition

La priorité d'une activité est définie, en contexte local, par rapport aux autres activités. Par exemple, une activité de priorité vingt se verra allouée deux fois plus de temps processeur qu'une activité de priorité dix. Cela nous a permis de définir la vitesse d'une activité de la manière suivante :

$$v(p) = \frac{priorité(x)}{\sum_{y \in \{processus\ actifs\ sur\ le\ processeur\}} priorité(y)} \tag{5.3}$$

Cependant si deux processeurs ont des threads de priorité différentes alors les priorités ne seront, dans le cas général, pas respectées globalement : une activité dont la priorité est inférieure à une autre située sur un processeur différent aura une vitesse supérieure. Ou bien, l'égalité donnée dans l'équation 5.2 ne sera plus respectée même si l'ordre de vitesse l'est. Nous allons détailler ces problèmes en nous aidant de la situation présentée sur la figure 5.3.

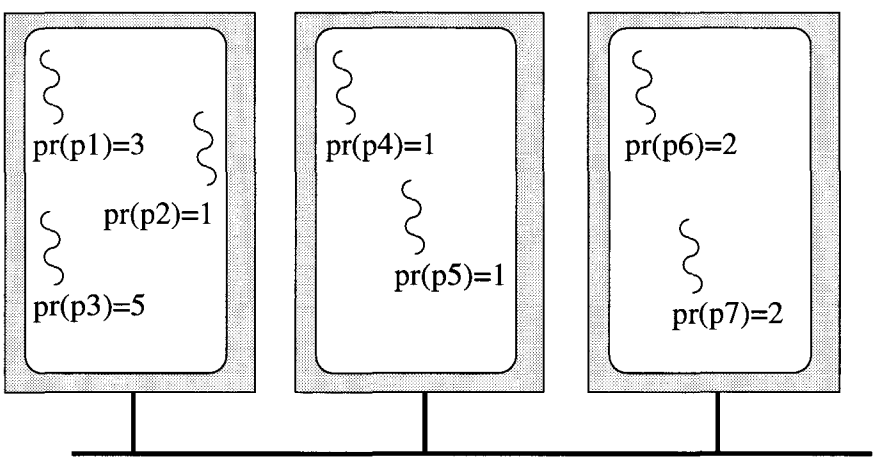


FIG. 5.3 - *Exemple de situation où les priorités locales ne sont plus respectées dans un contexte global*

1. p_1 a une vitesse $sp(p_1) = \frac{1}{3}$, p_6 a une vitesse de $sp(p_6) = \frac{1}{2}$. Bien que p_1 ait une priorité supérieure à celle de p_6, il s'exécute moins vite.

2. p_3 a une vitesse de $sp(p_3) = \frac{5}{9}$. p_3 s'exécute bien plus vite que p_6 mais pas $\frac{5}{2}$ fois plus vite.

La raison de ce non respect des vitesses tient au fait qu'elles dépendent des autres activités présentes sur le même processeur, elles sont uniquement locales. Pour étendre cette notion à l'ensemble des processeurs d'une architecture distribuée il faudrait que le terme du dénominateur de la formule 5.2 soit identique sur tous les processeurs, c'est-à-dire que la propriété :

$$P : \forall i, j \in \{Processeurs\} S(i) = S(j) \tag{5.4}$$

avec la fonction S définie comme suit :

$$S(i) = \sum_{p \text{ s'exécutant sur } i} Priorité(p) \tag{5.5}$$

est vérifiée à tout instant.

La conséquence est que la somme des activités s'exécutant sur chaque processeur doit être identique pour tous. Cette propriété est une conséquence de la politique locale d'ordonnancement avec priorités de MARCEL qui rend la vitesse d'exécution d'une activité dépendante de la somme des priorités des autres activités situées sur le même processeur.

Il existe cependant des cas pour lesquels cette propriété n'est pas vérifiable comme par exemple sur la figure 5.4. De plus, quand le calcul d'un placement des activités qui satisfait cette propriété est possible, son coût serait la plupart du temps trop important pour qu'ils soit envisageable de l'effectuer[6]. Cela est d'autant plus vrai que le caractère hautement dynamique du comportement des applications visées rendrait souvent le placement calculé obsolète dès sa découverte.

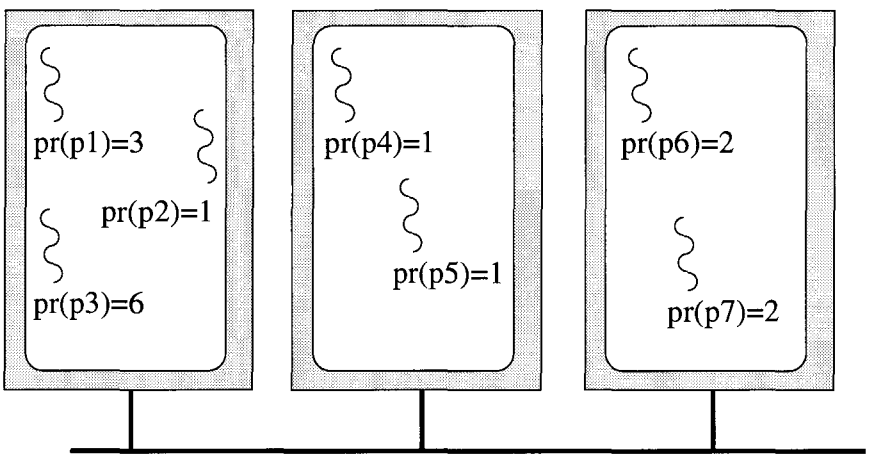


FIG. 5.4 - *Exemple de situation où il n'est pas possible de satisfaire la propriété P (la valeur de P_3 est différente de la figure 5.3)*

Pour remédier à ce problème, nous avons introduit une seconde propriété pour définir un « bon » degré de satisfaction du respect des priorités. C'est l'objet de la section suivante.

5.3.1.2 Le critère de qualité (ϵ)

La propriété P a l'inconvénient de ne fixer aucune tolérance dans le respect des priorités ce qui la rend, en pratique, non vérifiable la plupart du temps. Or, dans des applications réelles, il est rare d'avoir besoin d'un respect aussi strict des priorités et un déséquilibre léger est tolérable. Pour prendre en compte cette notion, on définit une seconde propriété P' comme suit :

$$P' : \forall i, j \in \{Processeurs\} | S(i) - S(j)| \leq \epsilon \tag{5.6}$$

6. On peut d'ailleurs noter que calculer un tel placement est une variante du problème du « sac à dos » qui est célèbre en optimisation combinatoire et qui fait partie des applications que nous visons dans ce manuscrit.

ϵ est le critère de qualité de la régulation, son choix affectera le degré de respect des priorités et aussi le coût de l'ordonnancement : plus la valeur de ϵ sera faible, meilleure sera le respect des priorité mais plus il coûtera.

Dans l'énoncé de la propriété, nous avons exprimé ϵ comme une valeur, en pratique un entier, mais cela n'est pas forcément une bonne idée. Il parait en effet illogique de fixer une valeur pour une application qui contiendra cent ou un million d'activités suivant les données du problème. Il paraît plus logique d'exprimer ϵ relativement à la valeur que nous voulons borner, ici la somme des priorités. Dans ce cas on définit une troisième propriété P'' :

$$P'' : \forall i, j \in \{Processeurs\}\ (S(i) \geq S(j)\ \frac{S(j)}{s(i)} \leq \epsilon)\ ou\ (S(j) \geq S(i)\ \frac{S(i)}{s(j)} \leq \epsilon) \qquad (5.7)$$

ϵ représente alors le degré de déséquilibre relatif tolérable par l'application et il est indépendant de la valeur de la somme des priorités sur les processeurs, il exprime un rapport minimum à respecter entre les processeurs.

Nous avons vu dans cette partie que le respect des priorités dans un contexte distribué était assuré, dans le contexte d'un ensemble d'ordonnanceurs locaux définissant la priorité d'une activité comme celui de MARCEL, si la somme des priorités des activités s'exécutant sur chaque processeur était la même (propriété P vue en 5.4). Nous avons ensuite vu comment cette propriété pouvait être relâchée avec les priorités P' (5.6) et P'' (5.7). Nous allons maintenant étudier comment ces propriétés sont mises en œuvre dans le régulateur LBMP[7].

5.3.2 Le régulateur LBMP

Le but du régulateur LBMP est de mettre en pratique les définitions présentées dans la partie précédente qui permettent d'étendre la notion de priorités à un contexte distribué. Il s'appuie sur l'ordonnanceur générique présenté dans le chapitre précédent en le spécialisant pour prendre en compte ce nouveau critère. Nous allons étudier la réalisation de cette spécialisation en l'illustrant avec des mesures de performance.

Le régulateur LBMP s'appuyant sur GTLB, il ordonnance les activités en influant sur leur placement à la création et aussi durant leur exécution. Par exemple, la situation, déséquilibrée, présentée sur la figure 5.3 page 127 peut être corrigée afin de satisfaire les priorités en effectuant simplement deux opérations de migration comme le montre la figure 5.5.

5.3.2.1 Spécialisation de GTLB

Nous allons détailler, comme nous l'avons fait pour une autre politique de régulation dans la section 4.6, comment cette politique est mise en place avec le régulateur générique en nous focalisant sur les points liés à la qualité de la régulation.

Critère de régulation Le critère est simple à exprimer puisqu'il s'agit de la somme des priorités. La bibliothèque MARCEL fournit une primitive permettant de connaître directement la somme des priorités des activités actives dans un processus lourd. Nous utiliserons cette fonction directement pour calculer cette valeur. Nous ajouterons une information supplémentaire qui est la priorité moyenne des activités présentes sur un processeur. Elle nous servira pour calculer le placement d'une nouvelle activité.

7. Le sigle LBMP signifie *Load-Balancing with Migration directed by Priorities.*

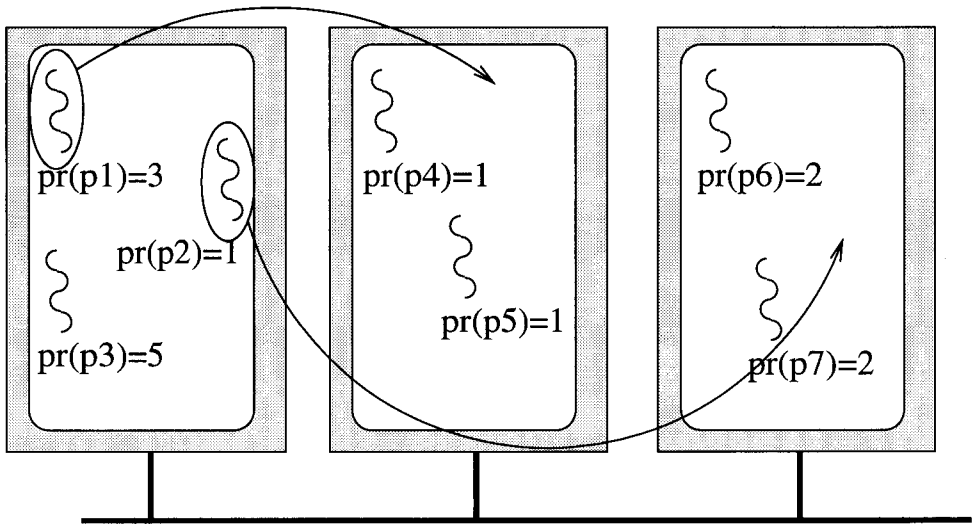


FIG. 5.5 - *Correction d'une situation déséquilibrée afin de satisfaire les priorités*

Le critère de régulation et la fonction qui permet de lire sa valeur sont donc :

```
typedef struct st_LB_INFO_CHARGE {
  int somme_prio;
  int prio_moyenne;
} load_info;


void get_load(load_info *ou)
{

    ou->somme_prio=pthread_priosum_np();
    if(pthread_activethreads_np()>0)
      ou->prio_moyenne=ou->somme_prio/pthread_activethreads_np();
    else
      ou->prio_moyenne=0;  /* 0 because no active tasks */
}
```

Implantation de cette politique Le point clé de cette politique est ϵ dont la valeur (et le type) conditionne la politique de régulation. Dans la structure générique que nous proposons il est possible de l'introduire à deux moments :

1. **Émission de la charge** Si la charge a varié de plus de ϵ et que la situation était préalablement équilibrée alors une action est nécessaire.

2. **Déséquilibre important** Un déséquilibre est jugé important seulement suivant le critère défini par la valeur et le rôle d'ϵ. La décision de rétablir l'équilibre est donc prise en fonction de cette valeur.

Une opération de rééquilibrage est toujours provoquée après la constatation d'un déséquilibre, c'est-à-dire que le point 2) n'est évaluée que si le point 1) a été vérifié. Dans ces conditions, il n'y a pas de sens à mettre un critère plus strict sur 2) que sur 1). Il est par contre possible de définir deux « niveaux » de réaction. Le premier, lié à l'émission de la charge, affectera la modification des informations globales et donc la quantité d'informations utilisée pour le placement. Le second définira la condition d'activation de la migration d'activités car il définit le critère d'un déséquilibre suffisamment important pour motiver un équilibrage immédiat.

Nous avons choisi dans l'exemple ci-dessous d'utiliser ϵ au premier *et* au second endroit. Si la charge a varié sur le processus destinataire, celui sur lequel s'exécute l'activité de régulation, il est nécessaire de refaire le test utilisant ϵ pour s'assurer que l'opération de migration est toujours utile. Enfin nous considérons ϵ comme un degré de déséquilibre relatif c'est-à-dire comme un pourcentage.

```
int significant_load_change(load_info *new,load_info *old)
{
  int differential=abs(new->somme_prio-old->somme_prio);
  int important_change=0;

  if( (max(load_value(new),load_value(old))*epsilon/100) < differential)
    important_change=1;

  return important_change;
}
```

Le choix des activités à migrer pour rétablir l'équilibre est un problème non trivial à résoudre car il revient au problème bien connu du « sac à dos ». En effet il convient de choisir des processus légers dont la somme des priorités doit être la plus proche possible d'une valeur sans la dépasser. Cela est équivalent à remplir un sac à dos avec des objets dont la somme des poids ne doit pas dépasser une valeur donnée. Nous avons choisi ici de ne pas chercher la solution optimale mais de procéder de manière itérative, sans « retour en arrière[8] ». Ce choix est dicté par des soucis de performance car cette opération ne doit pas prendre trop de temps, la situation sur laquelle elle agit fluctuant fortement au cours du temps. La méthode que nous avons adoptée[9] est la suivante : nous trions tout d'abord les processus légers par ordre décroissant de priorité puis nous parcourons ce tableau. Nous choisissons les activités au fur et à mesure tant que leur choix ne dépasse pas la valeur fixée. Lorsque c'est le cas, on ne choisit pas cette activité mais on essaie la suivante et ainsi de suite jusqu'à avoir épuisé le tableau de processus. Ainsi on ne trouve pas forcément la solution optimale mais le coût de la recherche est en $O(n)$ seulement, n étant le nombre de processus légers candidats, le coût du tri étant en $O(n \log n)$.

Pour calculer le placement nous prendrons en compte la somme des priorités en lui ajoutant le produit du nombre d'activités créées sur ce processeur depuis la dernière mise à jour des informations par la moyenne des priorités du même site. Ce n'est évidemment qu'une approximation grossière de la charge réelle du site mais cela permet d'éviter le phénomène d'inondation que nous avons déjà décrit.

Voilà le source des fonctions jugeant si un déséquilibre est suffisant pour décider d'un équilibrage et si cet équilibrage peut être acoompli. Cette dernière fonction ne fait que tester si la somme des

8. Backtracking en anglais.

9. Il s'agit de la méthode « gloutonne » classique pour le problème du sac à dos.

priorités du destinataire potentiel de la migration est inférieure à celle de l'émetteur.

```
int severe_disbalance(machine_load *c1,machine_load *c2)
{
  int res;

  res=abs((c1->load.somme_prio)-(c2->load.somme_prio))>=
    ( ( max(c1->load.somme_prio,c2->load.somme_prio)*epsilon)/100);

  return res;
}
/* ------------------------------------------------------------------ */
int balancing_possible(machine_load *src,machine_load *dest)
{
    if((src->load.somme_prio) > (dest->load.somme_prio))
      return 1;
    else
      return 0;
}
```

Le source de la fonction de choix des activités à migrer est un peu plus compliqué car il implique un tri de tableau ce qui oblige à définir une fonction supplémentaire de comparaison et un tableau auxiliaire. Ce tableau contient une structure spécifique qui ne contient que le critère de tri et l'identifiant de chaque processus léger migrable. Les fonctions `tmalloc` et `tfree` sont simplement les versions *thread safe* des appels équivalents de la librairie C traditionnelle.

```
/* Fonction utilisée pour le tri */
static int cmp_st_choice(st_elt_choice* i,st_elt_choice* j)
{
 return i->prio < j->prio;
}

pthread_t *choose_threads(int *nb_choisi,int nb_candidats,
              pthread_t *candidats,machine_load *from,machine_load *to)
{
  int i;
  st_elt_choice *tab_prio;
  pthread_t *tab_elu;
  int somme_prio_choisis=0; /* somme de la priorite des threads choisis */
  int value_to_reach=(from->load.somme_prio-to->load.somme_prio)/2;

  *nb_choisi=0;
  if(nb_candidats==0)
    return NULL;

  /* Recuperation de la priorite de chacun des threads candidats */
  *nb_choisi=0;
  tab_prio=tmalloc(sizeof(st_elt_choice)*nb_candidats);
```

```
  for(i=0;i<nb_candidats;i++) {
    pthread_getprio(candidats[i],&(tab_prio[i].prio));
    tab_prio[i].th_id=candidats[i];
  }

  /* Tri par ordre croissant */
  qsort(tab_prio,nb_candidats,sizeof(st_elt_choice),&cmp_st_choice);

  tab_elu=tmalloc(nb_candidats*sizeof(pthread_t));
  i=nb_candidats-1;
  while(i>=0 && somme_prio_choisis<value_to_reach) {
    if((somme_prio_choisis+tab_prio[i].prio)<=value_to_reach) {
      tab_elu[*nb_choisi]=tab_prio[i].th_id;
      somme_prio_choisis+=tab_prio[i].prio;
      (*nb_choisi)++;
    }
    i--;
  }

  tfree(tab_prio);

  return tab_elu;
}
```

La dernière fonction que nous allons voir est celle qui calcule le site sur lequel créé une nouvelle activité. Son fonctionnement est similaire à celle page 113 qui réalisait la même opération pour un critère différent, nous ne le détaillerons pas de nouveau ici.

```
/* ------------------------------------------------------------------ */
int trouver_place_thread(machine_load *tab_charge,int card_charge,
                         machine_load *charge_locale)
{
  int i;
  int indice_cmin,indice_change_min;

  if((card_charge==0) || (tab_charge==NULL))
    return charge_locale->num_machine;

  indice_cmin=0;
  charge_cmin=tab_charge[0].load.somme_prio+
    tab_charge[0].number_of_change*tab_charge[0].load.prio_moyenne;

  for(i=1;i<card_change;i++)
    if((tab_charge[i].load.somme_prio+
        tab_charge[i].number_of_change*tab_charge[i].load.prio_moyenne) <
        charge_cmin) {
      indice_cmin=i;
      charge_cmin=tab_charge[i].load.somme_prio+
```

```
            tab_charge[i].number_of_change*tab_charge[i].load.prio_moyenne;
    }

  if(charge_cmin<charge_locale->load.somme_prio)
    return tab_charge[indice_cmin].num_machine;
  else
    return charge_locale->num_machine;
}
```

5.3.2.2 Résultats expérimentaux

Pour mesurer l'efficacité de notre régulateur nous avons modifié l'application de TSP présentée dans les sections précédentes afin de la rendre utilisable avec PM^2. Cette transformation a consisté à transformer en service PM^2 la fonction assurant l'exploration d'un sous-arbre dans la version multithreadée « pure ». Nous avons dû aussi définir un nouveau service qui est utilisé pour propager une nouvelle valeur sur tous les processeurs lorsqu'elle est découverte.

Nous avons testé cette application sur dix problèmes de taille seize générés aléatoirement. L'architecture choisie est le cluster de processeurs Alpha décrits précédemment, la version de PM^2 était Hi-Perf PM^2 avec le protocole TCP. Les accélérations moyennes obtenues sont représentées sur la figure 5.6. La raison pour laquelle nous avons choisi des problèmes de taille plus importante que pour les tests faits sur un seul processeur est que les temps d'exécution des problèmes plus petits sont trop faibles pour que le gain apporté par les priorités soient significatifs ce qui aurait faussé un peu nos mesures. De plus dans ce cas les activités ont une durée de vie si courte que leur placement initial est alors très important ce qui n'est pas vraiment le cadre que nous visons.

Le gain moyen en nœuds obtenu est de 7 à 8 % pour 4 processeurs et d'à peu près 10% pour 8 processeurs par rapport à une version multithreadée tournant sur 1 processeur pour $\epsilon = 1\%$. Ces valeurs sont de 4 et 7 pour $\epsilon = 20\%$. Nous avons utilisé l'heuristique basé sur le vecteur de priorités présentée précédemment pour fixer les valeurs de priorité.

On constate que pour 4 processeurs le gain obtenu en relaxant la contrainte d'équilibre en fixant ϵ à 20% n'est pas suffisant pour absorber le déficit en qualité de régulation. Cette situation s'inverse pour 8 processeurs ce qui est cohérent avec le comportement déjà étudié de l'ordonnanceur : plus de processeurs augmente le coût relatif.

Le gain en nombre de nœuds peut être considéré comme décevant mais il s'explique par le caractère distribué et asynchrone des mises à jour de la meilleure solution. Alors que dans la version multithreadée non distribuée cette mise à jour est immédiate, elle prend ici un certain temps, temps pendant lequel du travail inutile est effectué. Cette particularité est liée à l'utilisation d'un réseau faiblement couplé et il n'a pas de solution simple [10]. Le même problème se pose pour la mise à jour du vecteur de priorités utilisé par l'heuristique mais il ne peut être résolu de la même façon. En effet, dès qu'une nouvelle solution est trouvée elle est immédiatement envoyée à tous les processeurs en utilisant un service spécifique. Cette implantation est souhaitable car il s'agit d'une information **importante**. Si nous appliquions le même traitement au vecteur de priorités le coût en communication supplémentaires serait trop important en regard du gain. Elle est donc propagée avec une meilleure solution donc moins souvent que ce qui serait souhaitable mais, là encore, un compromis a dû être trouvé.

Nous avons aussi constaté que les « petites » activités avaient une durée de vie si courte qu'elles

10. À part l'utilisation d'un mécanisme de mémoire partagée très efficace mais cela sort du cadre de cette étude.

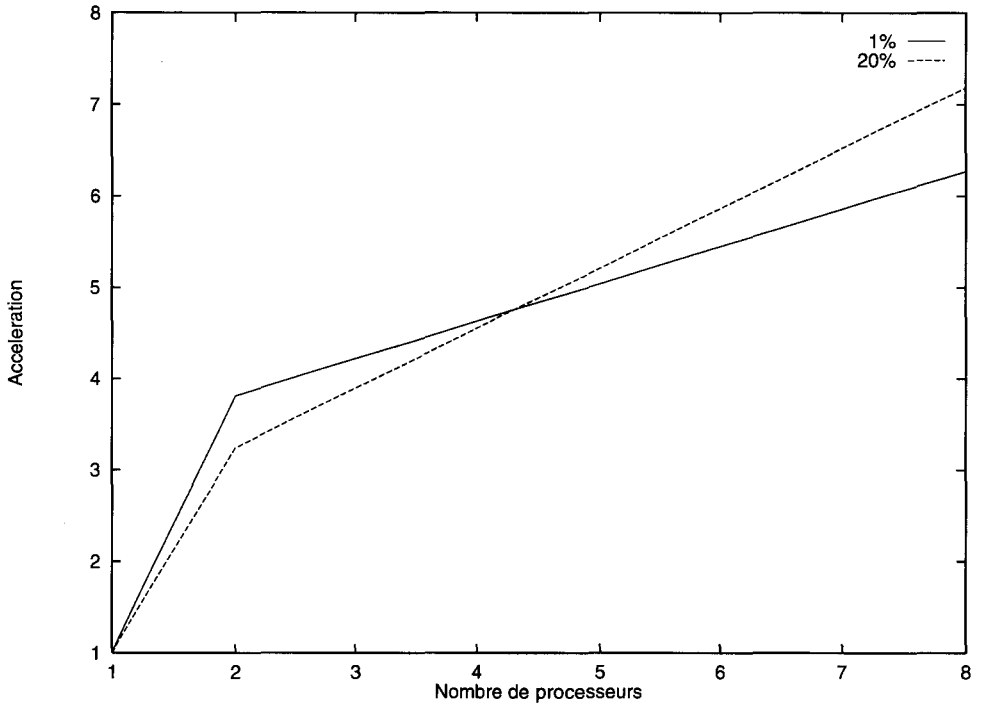


FIG. 5.6 - *Accélération moyenne obtenue sur 10 problèmes du TSP de taille 16 pour deux valeurs de* ϵ

n'avaient pas le temps d'être déplacées ce qui donne au placement à la création un rôle plus important que ce que nous envisagions en définissant l'ordonnanceur. Il se comporte cependant de manière tout à fait honorable même dans ce cas un peu extrême.

La propriété remarquable est que les résultats en terme d'accélération sont assez comparables avec ceux obtenus pour un critère différent dans le chapitre précédent. Cela semble montrer que le coût est bien indépendant de la politique mise en œuvre quand cette dernière oblige l'ordonnanceur à fonctionner en continu. Notons que dans les deux cas les priorités des activités de régulation étaient égales à la moyenne des priorités des activités applicatives afin de les placer sur un pied d'égalité. En modifiant cette valeur, on affecterait aussi le coût, et l'efficacité, de l'ordonnanceur.

Ces résultats demandent cependant à être confirmés sur des applications différentes d'optimisation combinatoire. Nous y revenons dans les perspectives.

5.4 Conclusion

Dans ce chapitre nous avons présenté la notion de priorités dans un cadre général en étudiant leur signification dans différents systèmes, ainsi que dans MARCEL. Puis, nous avons montré en quoi leur utilisation pouvait être utile puis dans le cadre des applications d'exploration arborescente. Nous avons insisté en particulier sur leur utilisation afin de favoriser l'apparition de bonnes solutions quand la recherche est spéculative. Nous avons étudié les conséquences de différentes heuristiques pour fixer les priorités dans le cadre d'une application d'optimisation combinatoire connue, celle du voyageur de commerce.

Nous avons ensuite posé le problème du respect des priorités locales définies par l'ordonnanceur

en temps partagé de MARCEL dans le cadre distribué qui est celui d'une application PM^2. Nous avons montré que pour qu'elles soient respectées dans un tel environnement il était nécessaire de mettre en place un équilibrage entre les processeurs de la somme des priorités des processus légers actifs. Nous avons présenté comment l'ordonnanceur GTLB était instancié pour prendre en compte cette nouvelle contrainte en illustrant notre discours avec des mesures de performances.

Chapitre 6

Conclusion et perspectives

Le travail décrit dans cette thèse s'est déroulé dans le cadre de l'équipe GOAL du Laboratoire d'Informatique Fondamentale de Lille au sein du projet ESPACE. L'objectif de ce projet est de fournir un cadre méthodologique pour la programmation des applications hautement irrégulières sur des architectures distribuées ainsi qu'un support assurant l'exécution efficace de telles applications. Un des soucis permanents de l'équipe ESPACE est de valider nos choix de structuration et de conception sur des applications réelles.

Le concept au cœur du projet ESPACE, la virtualisation, est composé de deux éléments. Le premier est la définition d'éléments structurants qui permettent de découper de manière simple et intuitive une application parallèle en activités indépendantes. Le second concerne la prise en compte de ces activités par le support pour une exécution efficace. Notre contribution a essentiellement porté sur ce second point, elle est issue des études que nous avons menées sur la parallélisation d'applications de recherche arborescente issue du domaine de l'optimisation combinatoire.

Ces applications ont comme particularité d'avoir une granularité applicative extrêmement fine donc un degré potentiel de parallélisme très élevé. Or, les supports d'exécution traditionnels ne sont pas conçus pour prendre en compte directement ce type de comportement car ils proposent une granularité exécutive trop importante. Il est donc nécessaire pour exécuter ce type d'applications de mettre en place des mécanismes qui permettent de faire correspondre ces granularités applicatives et exécutives différentes.

Le support d'exécution PM^2, qui constitue la base du projet ESPACE, supprime cette contrainte en proposant comme activités de base les processus légers qui sont de granularité fine. Ces entités sont les briques sur lesquelles s'appuie le modèle de programmation proposé par ESPACE. Ce modèle est basé sur la notion de virtualisation de l'architecture. Cette notion vise à affranchir le programmeur d'applications parallèles de l'architecture en lui donnant le moyen de structurer son application en activités qui seront ensuite exécutées de manière efficace par le support d'exécution sous-jacent.

Le modèle de programmation du projet ESPACE s'appuie sur deux éléments: les opérations structurantes qui permettent de diviser une application en activités (l'invocation distante de services et le clonage léger) et les composants qui assurent une prise en charge efficace des ces activités. Notre travail a essentiellement porté sur cet aspect du projet.

6.1 Travaux réalisés

Dans ce cadre, nos travaux ont porté sur la conception d'un ordonnanceur générique pour assurer la gestion des activités d'applications hautement irrégulières d'optimisation combinatoire. Notre but est aussi de valider son utilisation sur des applications réelles.

6.1.1 Ordonnanceur générique

Le concept sur lequel est basé l'aspect exécutif de la virtualisation est une séparation entre la localisation des activités d'une application parallèle et les processeurs de l'architecture sur lesquels elle s'exécute. Dans les applications qui nous intéressent, l'absence de relation autre que la parenté entre les activités rend cette indépendance totale.

Ces activités posent par contre d'autres contraintes d'ordonnancement liées à leur haut degré d'indéterminisme et au fait qu'elles ne soient pas à travail constant: un lien existe entre l'ordonnancement et la quantité totale de travail. Pour prendre en compte ces caractéristiques, nous avons défini un ordonnanceur générique qui peut être instancié pour créer des politiques d'ordonnancement différentes adaptées à cette classe d'applications.

Cet ordonnanceur a la particularité d'être transparent pour l'application, car il est basé sur la notion de multi-programmation légère, et d'avoir un coût bornable. Une application PM^2 peut être ordonnancée globalement sur une architecture quelconque sans nécessiter de réécriture ou d'adaptation. Cet ordonnanceur permet de plus de mettre en place des politiques d'ordonnancement *spéculatives* qui travaillent en permanence dans le respect du quota de temps qui leur est imparti.

Nous avons montré comment cet ordonnanceur générique était instancié pour définir une politique simple, régulant le nombre d'activités sur chaque processeur. Nous avons confirmé de manière expérimentale les résultats théoriques concernant le coût relatif de l'ordonnanceur dans l'exécution d'une application à travail constant.

6.1.2 Utilisation des priorités

La seconde contribution de ce travail concerne l'utilisation des priorités dans les applications hautement irrégulières. Le parallélisme à grain fin permet d'augmenter le nombre d'activités applicatives concurrentes ce qui augmente le degré de spéculation de l'application. Pour améliorer la qualité de cette recherche spéculative, nous avons proposé d'affecter des priorités aux processus légers qui exécutent les activités afin de favoriser ceux qui sont susceptibles de dimininuer la quantité totale de travail.

Le problème se pose alors de décider sur quels critères heuristiques nous nous basons pour décider quelles activités doivent être favorisées. Nous avons étudié dans ce but l'influence de plusieurs heuristiques sur la résolution de différentes instances du célèbre problème du voyageur de commerce.

L'utilisation des priorités dans un cadre distribué pose des problèmes nouveaux qui nécessitent la mise en place de techniques d'ordonnancement s'apparentant à de la régulation de charge. Nous avons présenté une spécialisation de l'ordonnanceur générique dont le but est de résoudre ce problème en égalisant la somme des priorités des processus légers actifs sur chaque processeur. Nous avons ainsi étendu la notion, originellement uniquement locale, de priorités à un cadre distribué.

6.2 Perspectives

Les perspectives que nous envisageons à ce travail sont multiples et concernent plusieurs domaines.

6.2.1 Évolution du régulateur

Le régulateur GTLB a été conçu dans le but de prendre en compte efficacement les applications d'exploration arborescente sur des architecture homogènes. Nous pensons que le modèle qu'il propose est suffisamment général pour pouvoir être utilisé dans d'autres cadres.

En particulier, le projet ESPACE s'oriente vers les architectures parallèles multi-clusters. Ces architectures sont composées d'un ensemble hétérogène de processeurs homogènes. La migration n'étant possible que dans un contexte hétérogène avec PM^2, l'ordonnancement d'une application sur ce type d'architectures pose de nouveaux problèmes qui nécessitent une approche mixte. Cette approche utilise le placement à la création pour obtenir un équilibrage « approximatif » entre les clusters et un autre plus fin entre les processeurs pour répartir équitablement les activités au sein d'un cluster.

Une évolution plus profonde du régulateur générique serait de l'utiliser comme élément de base d'un ordonnanceur distribué destiné aux applications de plus faible irrégularité que celles initialisement visées. Deux approches sont possibles pour la réutilisation de ce travail dans un tel cadre :

1. Utiliser la migration pour amener les activités vers les données partagées de l'application. Une fois sur le même site, elles peuvent les lire, et éventuellement en faire une copie, afin de pouvoir de nouveau être placées sur un processeur quelconque. En cas de modification de la valeur, une abstraction doit bien sûr être mise en place pour éviter les accès concurrents. Une telle approche, semblable à celle mise en place dans ARIADNE[MV], est une évolution du modèle existant auquel il faut adjoindre la notion de descripteurs globaux pour que les processeurs sachent où migrer pour accéder à une donnée. Elle a l'inconvénient d'amener les activités à faire des copies des données utilisées ce qui peut amener des pertes de performance et une saturation prématurée de la mémoire[1].

2. Utiliser le régulateur générique en conjonction avec un élément gérant les dépendances entre les activités comme le fait ATHAPASCAN-1 ou MILLIPEDE. La gestion des files d'activités qui ne peuvent pas être exécutées immédiatement, en attente de données ou d'un processeur libre, s'apparente en effet à l'équilibrage dynamique de charge que réalise le régulateur générique. L'implantation de nouvelles politiques de gestion de ces activités serait ainsi facilitée et bénéficierait des capacités de multi-programmation.

   Cette approche permettrait d'ajouter un degré d'indéterminisme dans l'exécution, chose qui peut être utile pour la phase de déverminage[2] ou pour effectuer un ordonnancement spéculatif avec un système qui n'était pas conçu dans ce but à l'origine.

La perspective globale d'une telle intégration serait d'unifier les différentes approches d'ordonnancement existantes au sein d'un noyau commun. Son but serait de gérer efficacement toutes les classes d'irrégularité en adaptant la politique mise en œuvre aux caractéristiques, à la classe d'irrégularité, de l'application.

6.2.2 Validation sur de nouvelles applications

Nous avons validé l'utilisation des priorités et de notre ordonnanceur sur une application d'optimisation combinatoire, celle du voyageur de commerce. La difficulté principale, qui est liée à PM^2,

1. Sauf si un ramasse-miette fonctionne en arrière-plan, comme suggéré dans la section suivante
2. CILK propose d'ailleurs une version de son support qui supprime le déterminisme de l'exécution dans ce but.

pour le développement d'applications nouvelles concerne la gestion des pointeurs au moment de la migration. En effet il est nécessaire d'enregistrer les pointeurs locaux, ceux qui pointent vers une case mémoire située dans la pile de l'activité, auprès du support d'exécution.

Cette contrainte est forte et complexifie la programmation des activités applicatives augmentant ainsi leur coût. Il limite surtout la réutilisation directe de codes existants. Une nouvelle version de la librairie de processus légers au cœur de PM^2, MARCEL, est en cours de développement pour remédier à ce problème. Son idée est tirée de la migration implantée, par exemple, dans MILLIPEDE : l'espace mémoire occupé par un processus léger est réservé simultanément pour lui sur tous les processeurs. La migration d'une activité ne requiert plus alors de translater sa pile ce qui élimine aussi la translation des pointeurs locaux.

Avec cette nouvelle version, le portage d'applications existantes devient plus aisé et il nous sera possible de valider le régulateur sur des problèmes différents. Celui qui nous intéresse plus particulièrement est bien évidemment l'affectation quadratique pour lequel l'utilisation de la potentialité comme valeur de priorité semble prometteuse.

Nous avons aussi l'intention de tester notre approche de l'ordonnancement sur des problèmes hautement irréguliers différents des problèmes d'optimisation combinatoire. La simulation de particules nous semble être un bon candidat dans cette optique.

6.2.3 Multi-programmation légère et priorités

Le concept qui nous semble particulièrement séduisant est celui de multi-programmation légère qui consiste à considérer une application comme des activités applicatives et des activités chargées de veiller à une bonne exécution de celles-ci.

Dans ce document nous avons utilisé ce mécanisme pour ordonnancer l'application mais ses applications potentielles sont plus larges que ce cadre. Citons par exemple la mise en place d'un « ramasse-miettes » ou encore d'un mécanisme de trace pour lequel il serait possible de borner les perturbations induites.

La possibilité de borner le coût relatif de ces outils dans l'exécution de l'application en les implantant de cette manière est très intéressante. Elle permet en effet de limiter les perturbations induites par leur utilisation ce qui est l'objection principale qui est faite à ces outils. Nous pensons que la multi-programmaiton légère est une alternative intéressante aux méthodes habituelles pour la mise en œuvre de ces fonctionnalités.

Dans ce document, nous avons abordé le coût de ces activités de gestion sans prendre en compte leur priorité. C'est une approche qui est réductrice par rapport aux capacités réelles de l'environnement. En effet, en modifiant les valeurs de priorité de ces activités, il est possible de modifier à la fois leur coût mais aussi leur impact. Dans le cas d'un « ramasse-miettes », il peut, par exemple, fonctionner en permanence mais avec une priorité très faible ce qui lui permettra de perturber au minimum l'application tout en accomplissant quand même son travail. Nous envisageons l'introduction de deux niveaux de priorité, un niveau applicatif et un niveau système, qui permettrait de séparer complètement les deux types d'activités. L'ordonnancement local comporterait bien entendu la possibilité de fixer le pourcentage de temps processeur alloué à chaque classe d'activités.

L'utilisation des priorités de processus légers dans ce cadre est un domaine encore peu exploré et qui est potentiellement très intéressant.

Annexe A

Description des applications étudiées

A.1 Description de l'algorithme Branch&Bound

Le but de l'algorithme de Branch&Bound est de résoudre un problème d'optimisation combinatoire, de complexité difficile en parcourant un espace de recherche pour minimiser une fonction de coût sur les élément de cet espace.

La méthode du Branch&Bound est basée sur l'énumération intelligente de toutes les solutions réalisables du problème. Le principe général est de décomposer un problème en plusieurs sous-problèmes de taille inférieure. Chaque sous-problème est résolu ou élagué si son évaluation est telle qu'il n'est pas possible que sa résolution amène à une meilleure solution que celle déjà trouvée. Si ce n'est pas le cas il est à son tour séparé en sous-problèmes jusqu'à trouver une solution ou à l'élaguer.

Le travail principal d'un B&B consiste donc le construction d'un arbre de sous-problèmes engendrés par division récursive à partir du problème initial. La contruction complète est évitée grâce à la découverte de nouvelles solutions qui permettent d'exclure certains sommets ou d'élaguer certaines branches de l'arbre.

On distingue trois méthodes d'exploration principales de l'arbre des sous-problèmes :

Meilleur d'abord : Les sous-problèmes sont choisis dans l'ordre des valeurs de leur borne inférieure (c'est-à-dire la valeur de leur évaluation partielle)

Largeur d'abord : Les sous-problèmes sont explorés suivant leur niveau dans l'arbre de recherche.

Profondeur d'abord : Les sous-problémes sont explorés suivant leur profondeur.

L'exploration « largeur d'abord » n'est plus utilisée car elle nécessite trop de mémoire pour stocker tous les sous-problèmes en attente d'exploration.

Le choix de la méthode d'exploration à adopter dépend des problèmes traités. Il est impossible de dire dans l'absolu que l'une est supérieure à l'autre et doit être utilisé dans tous les cas.

A.2 L'affectation quadratique

L'affectation quadratique a été introduit par Koopmans et Bekcman en 1957 [KB57]. Il consiste à placer n unités sur n sites de façon à minimiser le coût de cette affectation. Ce coût dépend de deux paramètres : la distance entre les sites et les flots entre les unités.

La formulation de ce problème est la suivante :

Étant donné deux matrices $(n \times n)$, avec $F = (f_{ij})$ avec f_{ij} le flot entre les unités i et j et $D = (d_{kl})$ avec d_{kl} la distance entre les sites k et l, il faut trouver une permutation p de l'ensemble $N = \{1, 2, \ldots, n\}$ qui minimise la fonction de coût total suivante :

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i)p(j)}$$

Un exemple concret d'application de ce problème est celui de l'implantation de n usines sur n sites avec une usine par site. Les distances inter-sites sont dans la matrice D. Par ailleurs, lorsque les le système fonctionne, les usines échangent une certaine quantité d'éléments (des matières premières, du personel ...). Le flux qui transite entre les usines est stocké dans la matrice F.

Le but de la résolution du problème est de trouver une implantation qui minimise le coût total de transport entre les usines. Ce coût est proportionnel à la quantité de matière transportée multipliée par la distance entre les usines.

A.3 Le voyageur de commerce (Traveling Salesperson Problem)

Le problème du voyageur de commerce est un des problèmes d'optimisation combinatoire les plus connus. Il consiste à trouver le chemin le plus court qui permet de visiter un ensemble de villes à partir d'une ville de départ donnée. Les distances entre les villes sont connues et chaque ville doit être visité une et une seule fois.

On définit un ensemble $\{c_1, c_2, \ldots, c_n\}$ de n villes et pour chaque paire de villes distinctes $\{c_i, c_j\}$ une distance. Le but du problème est de trouver un ordre σ entre les villes qui minimise la distance totale que représente la tournée, c-est-à-dire la quantité

$$\sum_{i=1}^{n-1} d(c_{\sigma(i)}, c_{\sigma(i+1)}) + d(c_{\sigma(n)}, c_{\sigma(1)})$$

Dans notre cas $\sigma(1)$ est fixé, il n'est pas choisi par l'algorithme.

Il esr clair que ce problème est un cas particulier du problème plus vaste de l'affectation quadratique.

Ce problème est très connu et il a fait l'objet d'un nombre considérable de publications. Une bibliothèque regroupant des jeux de données pour ce problème est disponible à

`http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html`.

Bibliographie

[AFR+94] Authié (Gérard), Ferreira (Afonso), Roch (Jean-Louis), Villard (Gilles), Roman (Jean), Roucairol (Catherine) et Virot (Bernard). – *Algorithmique Parallèle : Analyse et Conception.* – Hermès, 1994.

[AGF94] Ahmad (Ishfaq), Ghafoor (Arif) et Fox (Geoffrey C.). – Hierarchical scheduling of dynamic parallel computations on hypercube multicomputers. *Journal of Parallel and Distributed Programming*, no20, 1994, pp. 317–329.

[BK94] Barriuso (Ray) et Knies (Alan). – *SHMEM user's guide for C.* – Cray Research, août 1994.

[BKF91] Burkard (R.), Karisch (S.) et F. (Rendl). – Qaplib - a quadratic assignment problem library. *European Journal of Operational Research*, no55, 1991, pp. 115–119. – Disponible à `http://www.diku.dk/~karisch/qaplib`.

[BL94] Blumofe (Robert D.) et Leiserson (Charles E.). – Scheduling multithreaded computations by work stealing. *In : Proceedings of the 35th annual symposium on Foundations of computer Science (FOCS)*, pp. 356–368.

[BN83] Birell (A.D.) et Nelson (B.J.). – *Implementing Remote Procedure Calls.* – Rapport technique n° CSL-83-7, XEROX, 1983.

[BPT96] Bachelet (Vincent), Preux (Philippe) et Talbi (El-Ghazali). – Parallel hybrid metaheuristics: Application to the quadratic assignment problem. *In : Proceedings of Parallel Optimization Colloquium*, pp. 233–241.

[BT96] Bernard (Pierre-Eric) et Trystram (Denis). – *Algorithme Parallèle de Dynamique Moléculaire.* – Rapport technique n° 20, Laboratoire LMC-IMAG équipe Apache, Juin 1996.

[BTC96] Bernard (Pierre-Eric), Trystram (Denis) et Chapron (Yves). – Parallélisation d'un algorithme de dynamique moléculaire. *In : Actes de RenPar'8*, éd. par Castanet (Richard) et Roman (Jean), pp. 97–100.

[CCK+95] Casas (Jeremy), Clark (Dan), Konuru (Ravi), Otto (Steve), Prouty (Robert) et Walpole (Jonathan). – *MPVM: A migration transparent version of PVM.* – Rapport technique, Department of Computer Science and Engineering, Oregon Graduate Insitute of Science&Technology, 1995.

[CHM97] Cronk (D.), Haines (M.) et Mehrotra (P.). – Thread migration in the presence of pointers. *In : Proceedings of 30th Hawaii International Conference on Systems Sciences (HICSS'97).*

[Chr96] Christaller (Michel). – *Athapascan-0 : vers un support exécutif pour applications parallèles irrégulières efficacement portables.* – Grenoble I, Thèse de PhD, Université Joseph Fourier, Novembre 1996.

[cil97] MIT Laboratory for Computer Science. – *Cilk 5.1 Reference Manual*, Septembre 1997. Disponible à `http://theory.lcs.mit.edu/~cilk`.

[CKO+94] Casas (Jeremy), Konuru (Ravi), Otto (Steve), Prouty (Robert) et Walpole (Jonathan). – Adaptive load migration systems for pvm. *In : proceedings of Supercomputing '94*, pp. 390–399.

[Col95] Colin (J.N.). – *DTMS : Un environnement pour la programmation distribuée à grain indéterminé.* – Thèse de PhD, Université de Mons-Hainaut, 1995.

[dBKT95] de Bruin (A.), Kindervater (G.A.P.) et Trienekens (H.W.J.M.). – Asynchronous parallel branch and bound and anomalies. *In : Proceedings of Irregular'95*, éd. par Springer, pp. 363–377.

[Den97] Denneulin (Yves). – Programmation parallèle avec priorités et régulation de charge. *In : RenPar'9*, éd. par Schiper (André) et Trystram (Denis), pp. 171–174. – Lausanne, Mai 1997.

[DG96] Diderich (Claude G.) et Gengler (Marc). – Strategies for parallelizing branch and bound algorithms. *In : Proceedings of Parallel Optimization Colloquium*, pp. 177–186.

[DGR97] Doreille (Mathias), Galilée (François) et Roch (Jean-Louis). – Graphe de flots de données Athapascan et ordonnancement. *In : RenPar'9*, éd. par Schiper (André) et Trystram (Denis), pp. 163–166. – Lausanne, Mai 1997.

[DLCMM96] Denneulin (Yves), Le Cun (Bertrand), Mautor (Thierry) et Méhaut (Jean-François). – Distributed branch and bound algorithms for large quadratic assignment problems. *In : 5th Computer Science Technical Section on Computer Science and Operations Research.* – Dallas-USA, Janvier 1996.

[DM97] Denneulin (Yves) et Mautor (Thierry). – Techniques de régulation de charge - applications en optimisation combinatoire. *In : Icare'97*, éd. par Barth (D.), Chassin De Kergommeaux (Jacques), Roch (Jean-Louis) et Roman (Jean).

[DNGM96] Denneulin (Yves), Namyst (Raymond), Geib (Jean-Marc) et Méhaut (Jean-François). – Lbmp: Load-balancing with migration directed by priorities. *In : European School of Computer Science: Parallel Programming Environment for High Performance Computing*, pp. 115–118.

[Dow95] Dowaji (Salah). – *Contribution à l'étude des problèmes d'équilibrage de charge dans des environnements distribués.* – Thèse de PhD, Université Paris VI - Pierre et Marie Curie, juillet 1995.

[DR95] Dowaji (S.) et Roucairol (C.). – Influence of priority of tasks on load balancing strategies for distributed branch-and-bound algorithms. *In : IPPS'95, Workshop on Solving Irregular Problems on Distributed Memory Machines*, pp. 83–90. – Santa Barbara (USA), avril 1995.

[Dub96] Duboux (Thibault). – *Régulation dynamique du partitionnement des données sur machines parallèles à mémoire distribuée.* – Thèse de PhD, École Normale Supérieure de Lyon, Janvier 1996.

[Dum97] Dumoulin (Cédric). – *Dream : une mémoire partagée répartie à cohérence programmable.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Janvier 1997.

[ERLA94] El-Rewine (Hesham), Lewis (Theodore G.) et Ali (Hesham H.). – *Task Scheduling in Parallel and Distributed Systems.* – Prentice Hall, 1994.

[Fac95] Facq (Laurent). – Implémentation d'un solveur creux de Cholesky par blocs sur Paragon. *In : Comptes rendus des septièmes rencontres Francophones du parallélisme*, pp. 100–103. – Mons, Belgique, 1995.

[FFS95] Folliot (Bertil), Foughali (Karim) et Sens (Pierre). – Placement d'applications parallèles dans les réseaux de stations de travail : l'expérience de gatostar. *In : Journées de recherche sur le placement et la réparition de charge : Application aux systèmes répartis et parallèles*, pp. 69–72. – paris, mai 1995.

[FKOT94] Foster (I.), Kesselman (C.), Olson (R.) et Tuecke (S.). – *Nexus: An interoperability toolkit for parallel and distributed computer systems.* – Rapport technique n° ANL/MCS-TM-189, Argonne National Laboratory, USA, 1994.

[FKT94] Foster (I.), Kesselman (C.) et Tuecke (S.). – The Nexus task-parallel runtime system. *In : Proc. 1st Intl Workshop on Parallel Processing.* – Tata Mc Graw Hill.

[FMP95] Feschet (Laurent), Miguet (Serge) et Perroton (Laurent). – *Parallélisme et Applications Irrégulières*, chap. ParList : Une structure de données parallèle pour l'équilibrage des charges. – Hermès, 1995.

[Fon95] Fonlupt (Cyrille). – *Distribution dynamique de données sur machine SIMD.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Janvier 1995.

[FR95] Facq (Laurent) et Roman (Jean). – *Parallélisme et Applications Irrégulières*, chap. Algèbre linéaire creuse. – Hermès, 1995.

[FS95] Ferrari (A.) et Sunderam (V.). – TPVM: Distributed Concurrent Computing with Lightweight Processes. *In : Proc. of IEEE High Performance Distributed Computing.* pp. 211–218. – Washington, 1995.

[Gal95] Galilée (François). – Régulation de rpc par des heuristiques de prévision de charge. – Mémoire de DEA, Juin 1995.

[GBD+95] Geist (Alan), Beguelin (Andre), Dongarra (Jacques), Jiang (Wang), , Mancheck (Richard) et Sunderam (Vitek). – *PVM3 User's Guide and Reference Manual.* – Oak Ridge National Laboratory, Mai 1995.

[Gin97] Ginzburg (Alan). – *Athapascan-0b : Intégration efficace et portable de multiprogrammation légère et de communications.* – Thèse de PhD, Institut National Polytechnique de Grenoble, Septembre 1997.

[GRV95] Gautier (Thierry), Roch (Jean-Louis) et Villard (Gilles). – Regular versus irregular problems and algorithms. *In : Parallel Algorithms for Irregularly Structured Problems.* pp. 1–25. – Springer. Irregular 95.

[Haf96] Hafidi (Zouhir). – Mars : Un ordonnanceur adaptatif d'applications parallèles dans un environnement multi-utilisateurs. *In : Actes de RenPar'8*, éd. par Castanet (Richard) et Roman (Jean), pp. 37–40. – Bordeaux, Mai 1996.

[Haf98] Hafidi (Zouhir). – *MARS : Un environnement de programmation parallele adaptative dans un milieu heterogene multi-utilisateur.* – Thèse de PhD, Université des Sciences et Technologies de Lille, 1998. À paraître.

[HCM94] Haines (M.), Cronk (D.) et Mehrotra (P.). – On the design of Chant: A talking threads package. *In : Proceedings of Supercomputing'94*, pp. 350–359. – Washington, November 1994.

[Hem94] Hemery (Frédéric). – *Étude de la répartition dynamique d'activités sur architectures décentralisées.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Juillet 1994.

[HHM+95] Haines (M.), Hess (B.), Mehrotra (P.), Rosendale (J.V.) et Zima (H.). – Runtime support for data parallel tasks. *In : Proceedings of The 5th Symposium on the Frontiers of Massively Parallel Computation.* pp. 432–439. – Mc Lean.

[IEE94] IEEE STANDARDS DEPARTMENT. – *10003.4d8 POSIX system application program interface : Threads extensions [C language].* – 1994.

[ISS96] Itzkovitz (A.), Schuster (A.) et Shalev (L.). – *Millipede: Towards Standard Interface for Virtual Parallel Machines on Top of Distributed Environments.* – Rapport technique n° LPCR-9607, Computer Science department of Technion, Haifa, octobre 1996. Disponible à l'adresse : `http://www.cs.technion.ac.il/Labs/Millipede/papers/mil-xface.ps.Z`.

[KB57] Koopmans (T.C.) et Beckmann (M.J.). – Assignment problems and the location of economic activities. *Econometrica*, no25, 1957, pp. 35–73.

[LB96] Lewis (B.) et Berg (D.J.). – *Threads Primer. A Guide to Multithreaded Programming.* – Prentice Hall, 1996.

[LC96] Le Cun (Bertrand). – *Des structures de données parallèles.* – Thèse de PhD, Université Paris VI - Pierre et Marie Curie, Janvier 1996.

[LM96] Lee (Y.N.) et McKeown (G.P.). – A distributed branch-and-bound kernel using PVM. *In : Proceedings of Parallel Optimization Colloquium*, pp. 25–31.

[LW86] Li (G.-J.) et Wah (B.W.). – Coping with anomalies in parallel branch-and-bound. *IEEE transactions on computers*, vol. 35, n° 6, June 1986, pp. 568–573.

[Man92] Mans (Bernard). – *Contribution à l'algorithmique non numérique parallèle : Parallélisation de méthodes de recherche arborescente.* – Thèse de PhD, Université Pierre et Marie Curie - Paris 6, Juin 1992.

[Mel97] Melab (Noureddime). – *Gestion de la granularité et régulation de charge dans le modèle P^3 d'évaluation parallèle des langages fonctionnels.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Janvier 1997.

[Mes] Message Passing Interface Forum. – *MPI2.* – Disponible à http://www.mpi-forum.org/docs/mpi-20.ps.

[Mes94] Message Passing Interface Forum. – *MPI: A Message-Passing Interface Standard.* – March 1994. Disponible à netlib2.cs.utk.edu.

[Mey] Meybodi (M.R.). – Concurrent data structures for hypercube machine. *In: Proceedings of the Conference on Parallel Architectures and languages, 1992.*

[MR] Miguet (Serge) et Robert (Yves). – Elastic load-balancing for image processing algorithms. *In: Proceedings of the 1rst International ACPC Conference, 1991.* – Zima.

[MR95] Mascarenhas (E.) et Rego (V.). – *Ariadne: Architecture of a portable threads system supporting mobile processes.* – Rapport technique n° CSD-TR-95-017, Purdue University, March 1995.

[MV] Mascarenhas (Edward) et Vernon (Rego). – Migrant threads on process farms: parallel programming with Ariadne. – Disponible à http://www.cs.purdue.edu/research/PaCS/ariadne.html.

[Nam97a] Namyst (Raymond). – Le clonage léger : un opérateur de décomposition parallèle pour les applications irrégulières massivement parallèles. *In: RenPar'9*, éd. par Schiper (André) et Trystram (Denis), pp. 167–170. – Lausanne, Mai 1997.

[Nam97b] Namyst (Raymond). – *PM^2 : un environnement pour une conception portable et efficace des applications parallèles irrégulières.* – Thèse de PhD, Université des Sciences et Technologies de Lille, Janvier 1997.

[NM97] Namyst (Raymond) et Méhaut (Jean-Fraçois). – *Efficient management of high speed networks in the PM^2 multithreaded environment.* – Rapport technique, Laboratoire d'Informatique du Parallélisme de Lyon, 1997. À paraître.

[Pla93] Plateau (Brigitte). – *Présentation d'*APACHE. – Rapport technique, Grenoble, IMAG, Octobre 1993.

[RK88] Rao (V.N.) et Kumar (V.). – Concurrent insertions and deletions in a priority queue. *In: IEEE proceedings of International Conference on Parallel Processing*, pp. 207–211.

[Rou95] Roucairol (Catherine). – STRATAGÈME *: Une méthodologie de programmation parallèle pour les problèmes non structurés.* – Rapport technique, Versailles, PRiSM, Décembre 1995.

[RVR95] Roch (Jean-Louis), Villard (Gilles) et Roucairol (Catherine). – *Parallélisme et Applications Irrégulières*, chap. Algorithmes irréguliers et ordonnancement. – Hermès, 1995.

[SG94] Silberschatz (Abraham) et Galvin (Peter B.). – *Principes des systèmes d'exploitation.* – Addison-Wesley, 1994.

05376269λ

[SJ90] Størner (Henrik) et Joensen (Daniel Eide). – *Introduction to Emerald*, septembre 1990. Disponible à `ftp://ftp.diku.dk/pub/diku/dists/emerald`.

[SJ95] Steensgaard (Bjarne) et Jul (Eric). – Object and native code thread mobility among heterogeneous computers. *Communications of the ACM*, 1995.

[SS97] Schuster (Assaf) et Shalev (Lea). – *Access histories: How to use the principle of locality in distributed shared memory systems.* – Rapport technique n° LPCR-9701, Computer Science department of Technion, Haifa, Janvier 1997. Disponible à `http://www.cs.technion.ac.il/Labs/Millipede`.

[Sun88] Sun Microsystems. – *Systems Services Overview Chapter 6: Lightweight Processes part number 800-1753*, 1988.

[Sun93] SunSoft. – *SunOs5.2 Guide to Multi-Thread Programming*, 1993.

[SV] Sinha (Amitabh B.) et V. (Kalé Laxmikant). – A load balancing strategy for prioritized execution of tasks. *In: ECOOP 1992.*

[Tan92] Tanenbaum (Andrew). – *Les systèmes d'exploitation.* – InterEditions, 1992.

[Zay87] Zayas (E.R.). – Attacking the process migration bottleneck. *In: Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 13–24.

UNIVERSITÉ DES SC. ET TECHN. BIBLIOTHÈQUE UNIVERSITAIRE LILLE