HAL
open science

Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées

Samuel Thibault

To cite this version:

Samuel Thibault. Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées. Calcul parallèle, distribué et partagé [cs.DC]. 2004. hal-01962358

HAL Id: hal-01962358

https://inria.hal.science/hal-01962358

Submitted on 20 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées

Samuel Thibault

Mémoire de DEA

sous la direction de Raymond Namyst

25 Juin 2004

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : lip@ens-lyon.fr

Table des matières

1 Introduction **1**
1.1 Des environnements pour le calcul hautes performances . . . 1
1.2 Des machines de plus en plus difficiles à exploiter . . . 1
1.3 Un ordonnanceur flexible . . . 2

2 Exploitation de machines hiérarchiques à profondeur variable **3**
2.1 Machines multiprocesseurs de plus en plus complexes . . . 3
2.1.1 Nœuds . . . 3
2.1.2 Hyper-Threading . . . 4
2.1.3 Voire... . . . 6
2.2 Difficultés d'organisation des calculs . . . 6
2.2.1 Répartition des calculs . . . 6
2.2.2 Cohérence de la répartition des données et des threads . . . 7
2.3 Solutions existantes . . . 7
2.3.1 Décentralisation . . . 7
2.3.2 Allocation initiale des données . . . 8
2.3.3 Dynamicité des données . . . 8
2.3.4 Répartition des calculs par l'ordonnanceur . . . 9
2.4 Conclusion . . . 9

3 Un ordonnanceur universel guidé par l'application **11**
3.1 Modélisation et abstraction de l'architecture . . . 11
3.2 Modélisation et abstraction de la structuration des calculs . . . 13
3.3 Proposition : un ordonnanceur à bulles . . . 14
3.3.1 Évolution des bulles . . . 14
3.3.2 Notion de priorités . . . 16
3.3.3 Rééquilibrage . . . 16
3.4 Conclusion . . . 16

4 Détails d'implantation **19**
4.1 La bibliothèque de processus légers MARCEL . . . 19
4.2 Modularité . . . 19
4.3 Fonctionnement général . . . 20

5 Évaluation des performances **21**
5.1 Environnement de test . . . 21
5.2 Un exemple de test . . . 21

6 Conclusion **25**
6.1 Bilan 25
6.1.1 Un passage à l'échelle 25
6.1.2 Un ordonnanceur à bulle 25
6.1.3 Un dialogue avec l'application 25
6.1.4 Autres voies 26
6.2 Travaux futurs 26
6.2.1 Amélioration 26
6.2.2 Allocation mémoire 26
6.2.3 Une liberté encore plus grande pour l'application 26
6.2.4 Des «comportements» des tâches 26
6.2.5 Support Posix 26

A Algorithme d'ordonnancement **29**

Chapitre 1

Introduction

1.1 Des environnements pour le calcul hautes performances

Dès l'apparition des premiers ordinateurs, un des axes de recherche majeurs dans le domaine informatique a été de proposer à la communauté scientifique des moyens de traiter le plus rapidement possible des problèmes de complexité croissante, exigeant de longs calculs sur des données volumineuses, tout en exécutant en même temps quelques tâches systèmes comme la gestion des communications.

Pour réunir cette puissance de calcul demandée par de nombreux secteurs d'activité, le parallélisme s'est imposé comme la solution la plus effective, proposant des machines de calcul *hautes performances* qui réunissent la puissance de plusieurs processeurs sur une même station de travail (SMP : *Symmetrical Multi-Processing*). Ces stations servent alors de brique de base pour les architectures distribuées que sont les *grappes* — ou *clusters*.

Cependant, programmer directement de telles machines s'est rapidement révélé difficile. De nombreux environnements de programmation répartie ont donc été conçus pour encadrer l'exploitation de ces machines. Ils fournissent au programmeur d'application parallèle une plate-forme de programmation de haut niveau qui lui permet de s'abstraire des détails techniques tels que l'organisation des flots de calculs exécutés en parallèle. Certains (comme MPI) ne s'occupent que de l'aspect communication, d'autres se focalisent sur la gestion des flots d'exécutions (OPENMP [1]). Enfin, des environnements tels que PM^2 [2] proposent une intégration fine des deux aspects.

1.2 Des machines de plus en plus difficiles à exploiter

Pour pouvoir augmenter le nombre de processeurs sur une station, aligner simplement les processeurs sur une même carte mère est rapidement impossible, car l'accès à la mémoire devient un goulot d'étranglement, quelle que soit la technologie : un bus devient rapidement trop occupé, un commutateur réseau comporte rapidement trop d'étages. On est donc obligé de hiérarchiser l'organisation de ces processeurs, en regroupant par exemple quatre d'entre eux et une certaine quantité de mémoire sur une carte fille, et en enfichant quatre de ces cartes fille sur la carte mère d'une station, fournissant ainsi seize processeurs à l'utilisateur.

Cependant, le temps que le processeur met à lire une donnée en mémoire dépend alors de la localisation de cette mémoire. Si elle n'est pas située sur la même carte que le processeur, passer par le bus principal de la machine pour aller chercher la donnée sur une autre carte ajoute un coût non négligeable (effet NUMA : *Non Uniform Memory Access*). Il semble donc important de chercher à placer les données dans la mémoire des cartes où s'effectueront les

calculs qui en auront le plus besoin, tout en s'efforçant pour autant de répartir équitablement les calculs sur les différents processeurs pour gagner en parallélisme.

Il est certes parfois possible de déterminer statiquement la meilleure manière de répartir données et calculs pour une application donnée sur une machine donnée, mais lorsque l'on change de machine, cette répartition est alors souvent caduque. En outre, si les temps de calcul dépendent des données, la répartition ne peut être calculée avant l'exécution.

De nombreux environnements d'exécution établissent donc d'abord une répartition initiale des calculs, et utilisent ensuite une stratégie dynamique : si certains processeurs terminent leurs calculs avant d'autres, ils «volent» du travail à ceux-ci pour rééquilibrer la répartition. Il est cependant important que les données suivent également ce rééquilibrage, or les environnements ne savent en général pas à quelles données les calculs auront besoin d'accéder. Ils effectuent donc une mesure empirique des accès inter-cartes aux données, ce qui indique quels processeurs ont le plus utilisé telle ou telle donnée. Des heuristiques de prédiction permettent alors de décider de déplacer quelques données sur la carte où les données sont susceptibles d'être utilisées souvent à l'avenir. Cependant, le coût d'analyse et de migration (qui est parfois inutile, et souvent pénalisante) est en général assez important, pour un résultat décevant. Cette fonctionnalité est en fait bien souvent désactivée par les utilisateurs !

1.3 Un ordonnanceur flexible

Le problème principal qui apparaît est un manque de concertation entre l'application et l'environnement de programmation : on ne sait associer données et calculs qu'en *observant* le déroulement des calculs. Pourtant, le programmeur a en général une assez bonne idée de la répartition de l'utilisation des données ; il lui manque simplement un moyen pour l'exprimer. Les directives du langage OPENMP sont un exemple intéressant d'indications que le programmeur donne à l'environnement, mais ces directives s'adressent surtout à une programmation statique, qui permet difficilement une adaptation automatique à la machine.

Notre proposition est donc de permettre à l'application de décrire l'organisation de ses calculs, en précisant par exemple les calculs travaillant plutôt sur les mêmes données. L'ordonnanceur suit alors cette organisation pour répartir les calculs de la manière la plus appropriée selon la machine à disposition : un seul processeur, plusieurs processeurs, une hiérarchie de processeurs et de mémoire, etc. Et l'expérience montre qu'il y a effectivement un gain, même sur un seul processeur car c'est aussi une façon de localiser les calculs et de profiter du cache de ce processeur.

Nous verrons d'abord les problèmes rencontrés par les approches actuelles face à ces machines hiérarchisées. Nous détaillerons ensuite les nouveaux modèles utilisés pour organiser les calculs. L'implémentation sera alors présentée avant de donner quelques résultats et conclure.

Chapitre 2

Exploitation de machines hiérarchiques à profondeur variable

Toujours plus de puissance... Comme le montre le TOP 500 des ordinateurs les plus performants du monde, les machines hiérarchisées, bâties autour d'éléments standardisés, se sont imposées ces dernières années comme la solution privilégiée des grands centres de calculs. Cependant, l'exploitation de ces configurations gigantesques est difficile aussi bien au niveau macrocospique — dans [3], PETRINI, KERBYSON et PAKIN mettent en évidence qu'un phénomène de résonance entre différents services était à l'origine des contre-perfomances de cette machine — qu'au niveau microscopique comme nous allons le montrer dans ce chapitre.

Pour ce faire, après avoir brièvement décrit l'architecture des machines NUMA, briques de base de configurations de calculs plus larges, nous précisons les difficultés spécifiques à l'exploitation des machines hiérarchisées et présentons les solutions apportées par la communauté.

2.1 Machines multiprocesseurs de plus en plus complexes

Pour des raisons technologiques mais avant toutes économiques, les machines parallèles commercialisées de nos jours sont constituées de matériels standards hiérarchiquement organisés. Nous décrivons ici cette hiérarchie.

2.1.1 Nœuds

L'implantation symétrique des processeurs sur une même carte mère (SMP) est plutôt complexe. Il faut, entre autres, distribuer les signaux d'horloges de manière synchronisée, c'est-à-dire calculer les longueurs des pistes, ajouter parfois des portes de retardement pour allonger artificiellement le temps de propagation des signaux en évitant les problèmes de capacitance. Il faut également permettre à tous les processeurs d'accéder à la mémoire de manière concurrente. Pour éviter une contention, il est par exemple possible de découper cette mémoire en bancs auxquels les processeurs pourront, si le déroulement des calculs le permet, accéder de manière indépendante à l'aide d'un commutateur parfait (*cross-bar*).

On ne peut donc pas augmenter indéfiniment le nombre de processeurs ainsi synchronisés, il faut hiérarchiser cette implantation. On enfiche donc plusieurs processeurs sur une même carte fille (appelée **nœud**) comme on le fait en SMP sur une carte mère. La carte fille gère par exemple son propre signal d'horloge synchronisé et possède de la mémoire. Il suffit alors d'enficher plusieurs de ces cartes fille sur une même carte mère pour permettre aux processeurs

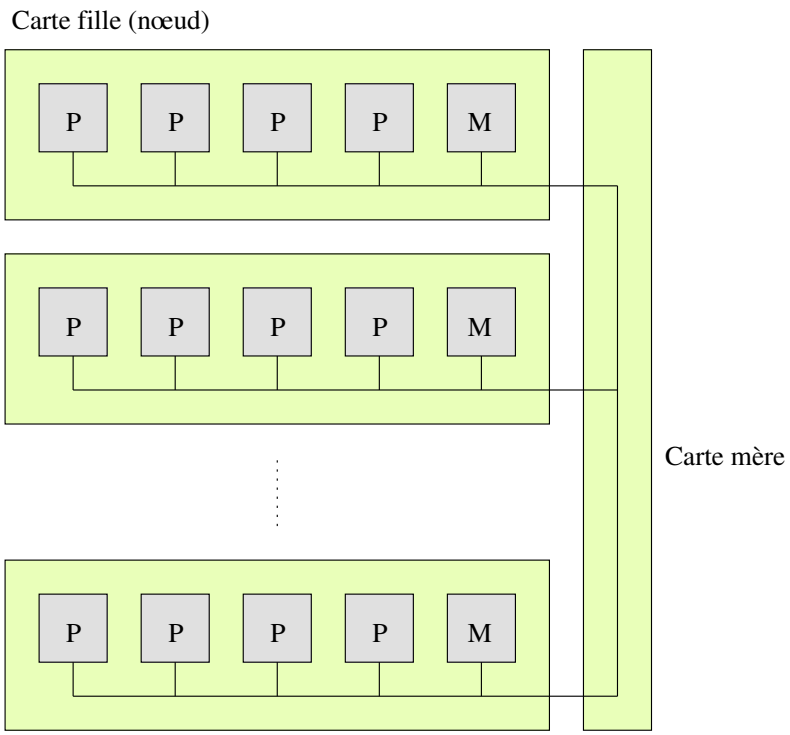


FIG. 2.1 – Machine NUMA.

d'accéder aux mémoires des autres cartes fille en passant par la carte mère, cf figure 2.1. On observe alors que le temps d'accès à la mémoire distante (inter-carte) est à peu près le double d'un accès à la mémoire locale : c'est le facteur NUMA (Non-Uniform Memory Access).

Enficher de nombreuses cartes fille sur une même carte mère étant finalement tout aussi complexe, certaines machines NUMA ont leurs nœuds organisés suivant des topologies particulières (tore, hypercube, *butterfly*), si bien que le temps d'accès distant dépend en plus de la distance entre les cartes.

Pour le programmeur une machine NUMA est similaire à une machine SMP. En effet, tous les processeurs peuvent accéder à toute la mémoire et toute modification effectuée par un processeur est « immédiatement » visible par tous les autres : la machine est dite à cohérence de cache (*Cache Coherent*). La seule différence perceptible par l'utilisateur un peu attentif aux performances, est que le temps d'accès à la mémoire n'est pas uniforme, ce temps dépendant des positions relatives du processeur et de la mémoire. Aussi, dans le cadre des hautes performances, il est important d'avoir des éléments statistiques sur l'accès distant à la mémoire, afin de déterminer si la répartition des données correspond bien à celle des calculs, par exemple. Ces machines proposent donc des compteurs d'accès mémoire inter-carte pouvant indiquer précisément quelles données sont utilisées par d'autres cartes.

2.1.2 Hyper-Threading

Le parallélisme est déjà présent au cœur même des machines. En effet, les processeurs modernes sont *pipelinés* et *superscalaires*. La technique du pipeline consiste à découper temporellement et spatialement l'exécution des instructions, autorisant le traitement à la chaîne du flot d'instructions, permettant ainsi d'augmenter plus facilement la fréquence du processeur. Un processeur est dit superscalaire s'il intègre plusieurs unités de calcul, c'est-à-dire qu'il est capable de traiter plusieurs instructions en même temps. Ces deux techniques ont largement

(a) Schématisation

(b) Aspect réel : la plupart de la puce est partagée entre les deux processeurs logiques

FIG. 2.2 – Processeur hyper-threadé.

contribué à respecter la loi de Moore dans les années 1990.

Cependant, l'exécution d'un programme séquentiel ne se prête pas toujours à l'utilisation optimale des ressources ainsi fournies. L'effet pipeline peut être anéanti par un défaut de cache par exemple, ce qui produite une «bulle» dans le pipeline : le retard occasionné par l'attente des données est propagé le long du pipeline, les unités doivent chacune à leur tour attendre ce retard.

L'effet superscalaire est également mis en défaut si le programme ne s'y prête pas. Si les calculs du programme dépendent les uns des autres, il n'est pas possible de les effectuer en parallèle, et les unités de calculs ne sont alors pas toutes occupées.

L'idée de l'*Hyper-Threading* — implémenté par INTEL™ dans ses processeurs XEON™ [4], aussi connue sous le nom de *Symmetrical Multi-Threading* (SMT) — est d'essayer de fournir encore plus d'instructions à la fois au processseur en lui demandant d'exécuter deux programmes en même temps. On espère ainsi que par exemple, si l'un des deux programmes doit attendre qu'une donnée soit chargée en mémoire, l'autre pourra s'exécuter.

Pour cela, tous les registres et états qui permettent d'exécuter un programme sont dupliqués, formant deux processeurs dits *logiques*. Par contre, tous les autres éléments sont partagés : les unités de calcul, le tampon de translation d'adresse (TLB) (pour les données seulement), les caches, etc. Il faut bien sûr ajouter cependant de nouveaux fils pour relier les registres aux unités de calcul, par exemple, ainsi qu'indiquer dans la TLB lequel des deux processeurs logique utilise telle ou telle entrée (il y a ici compétition). Notons que c'est en fait dommage dans le cas où les deux programmes exécutés utilisent en fait la même translation d'adresse, ce qui est le cas de deux threads d'un même processus : ils pourraient partager complètement la TLB. Un tel marquage n'est cependant pas nécessaire pour les caches : puisqu'ils sont adressés après translation par la TLB, toute ligne de cache chargée pour l'un des deux programmes est tout à fait valable pour l'autre.

Au final, l'opération n'est pas très coûteuse : la taille de la puce est augmentée d'à peine 5%.

Pour le système d'exploitation, ce processeur a l'air de se comporter comme s'il y avait deux processeurs SMP. Si deux programmes doivent s'exécuter, le système peut confier un programme à chaque processeur logique, comme s'ils étaient de véritables processeur. Il y a déjà un certain gain ici : sans cette technologie, le système ne «verrait» qu'un seul processeur, et serait obligé, pour faire croire à l'utilisateur que les deux programmes tournent «en parallèle», de passer son temps à exécuter le premier un certain temps, puis passer au deuxième pendant un certain temps, revenir au premier, etc jusqu'à la terminaison de l'un d'eux. Le coût de basculement d'un programme à l'autre n'est pas négligeable. Avec l'Hyper-Threading, les deux programmes tournent effectivement en parallèle, il n'y a plus besoin de basculement entre les deux.

Il n'y a donc a priori besoin d'aucune mise à jour logicielle pour pouvoir profiter de cette

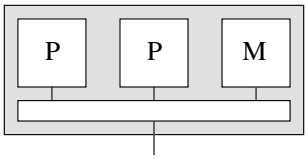


FIG. 2.3 – Multicore avec mémoire principale intégrée.

nouvelle technologie. Cependant, puisque les ressources sont partagées, même si exécuter deux programmes en même temps à l'aide de cette technique permet effectivement de gagner un peu de temps, la concurrence d'accès aux unités de calcul est telle que le gain n'est pas toujours concret. Quelques mesures empiriques montrent que sur un processeur XEON™, qui ne possède que quelques unités de calcul, exécuter deux programmes de calculs purement entiers en parallèle prend autant de temps que les exécuter séquentiellement. Par contre, exécuter deux programmes de calculs purement flottants prend effectivement deux fois moins de temps que leur exécution en séquence. Lorsque l'on mélange entiers et flottants, on obtient un gain intermédiaire, le calcul flottant étant en général privilégié. En exploitation réelle, le gain est souvent décevant. Les auteurs de l'ordonnanceur de FREEBSD 5 indiquent même : *At this time enabling Hyper-Threading logical cores leads to worse performance than having them disabled* [5].

Ainsi l'exploitation de la technologie Hyper-Threading dépend à la fois de la nature et de l'organisation des calculs. Aussi, si le programmeur ou si l'application sait qu'un sous-ensemble de ses calculs peut être avantageusement exécuté en parallèle sur un même processeur XEON™, il paraît intéressant de lui permettre de l'exprimer, ceci n'est pas le cas actuellement.

2.1.3 Voire...

D'autres organisations hiérarchiques sont bien sûr possibles. Il est possible de graver plusieurs processeurs sur une même puce, la mémoire cache de second niveau étant alors partagée entre ces processeurs (technologie *Multi-Core*). La mémoire principale pourrait même être gravée directement sur la puce [6], voir figure 2.3. L'accès à la mémoire intégrée à la puce serait ainsi très rapide. Cependant, pour accéder à la mémoire intégrée à une autre puce, il faudra tout de même passer par la carte sur laquelle sont enfichées ces puces. On retrouve ici un problème de localité respective des calculs et des données.

Ces puces pourraient même alors être enfichées sur une carte fille, de telles cartes filles elle-même enfichées sur une carte mère, ajoutant un niveau supplémentaire dans la localité respective des calculs et données.

2.2 Difficultés d'organisation des calculs

Les nouvelles génération de machines performantes sont donc hiérarchisées, cette structuration permet d'augmenter notablement le nombre de processeurs qu'elles peuvent accueillir. Cependant, savoir en profiter n'est pas toujours très simple car il faut savoir jouer avec cette hiérachie et composer avec les effets combinés des facteurs NUMA, de l'Hyper-Threading, et du partage des caches.

2.2.1 Répartition des calculs

Pour exploiter ces nombreux processeurs, il est tout d'abord nécessaire de *paralléliser* les calculs en les découpant en tâches.

Ce travail peut être fait *explicitement* en utilisant une bibliothèque de threads telle que celle proposée par la norme POSIX. C'est parfois assez fastidieux, et plus ou moins expressif : ce n'est que récemment que le support proposé par LINUX permet de choisir sur quel processeur on désire qu'un thread soit exécuté.

Le programmeur peut aussi se contenter d'*indications* : OPENMP [1] repère de simples directives ajoutées aux structures de contrôle «classiques». L'avantage ici est la très grande simplicité pour le programmeur. Il peut reprendre un vieux programme — même s'il n'était pas du tout pensé pour l'exploitation de machines multiprocesseurs au départ — en ajoutant simplement quelques lignes. Le compilateur s'occupe alors d'ajouter les instructions pour lancer au besoin les threads nécessaires pour effectuer les calculs en parallèle et récupérer leurs résultats.

Il est aussi parfois possible que le découpage soit *automatiquement* effectué par le compilateur : HPF (Hi-Performance Fortran, [7]) est une extension au langage Fortran qui permet au compilateur de découper automatiquement les calculs demandés.

Le calcul réside alors finalement en l'exécution de tâches par des threads. C'est à l'environnement d'exécution que revient la répartition des threads sur les processeurs. Une gestion centralisée n'est en général pas une bonne idée car le nombre de processeurs augmentant, un goulot d'étranglement apparaît : dans [8], il est montré que structurer de façon hiérarchique la gestion des threads permet d'éviter la contention d'une solution centralisée.

2.2.2 Cohérence de la répartition des données et des threads

Une fois les calculs répartis dans des threads exécutés sur différents processeurs, il faut être attentif au placement des données. Ainsi, sur une machine NUMA, les performances risquent d'être très dégradées si les données sont toutes stockées sur un même nœud, la plupart des threads étant désavantagés par le surcoût des accès distants. De même, un placement aléatoire obtiendra, en général, des performances plutôt mauvaises : en moyenne, en ignorant les effets de cache, la plupart des accès mémoire seront distants.

De fait, il est impératif que l'ordonnanceur prenne en compte l'affinité des threads et de leurs données.

2.3 Solutions existantes

De nombreuses solutions ont été proposées pour répondre à ces problèmes, avec plus ou moins de succès.

2.3.1 Décentralisation

C'est presque un détail, mais les efforts de décentralisation des structures système apportent un certain gain en performances.

Par exemple, tandis que le noyau LINUX 2.4 utilisait une liste globale de tâches, où tous les processeurs piochaient de manière concurrente pour trouver du travail, le noyau 2.6 utilise désormais une liste de tâches par processeur, chacun pouvant piocher dans sa liste sans gêner les autres [9]. Un équilibrage est ensuite réalisé entre ses listes, décrit à la section 2.3.4. Les résultats sont encore mitigés, le réglage des paramètres d'équilibrage n'étant pas encore très clair. Le nouvel ordonnanceur de FREEBSD 5, ULE [5], suit la même tendance.

D'autre part, les différentes informations nécessaires pour gérer les processeurs sont également bien séparées par processeur pour gagner en effet de cache et, sur NUMA, conserver ces données sur le nœud contenant le processeur.

Le code du noyau lui-même est également répliqué sur chaque nœud : cette partie si importante du système est recopiée dans la mémoire de chacun des nœuds, pour que les processeurs puissent disposer en permanence d'une copie locale. Il est également en projet de répliquer également le code des programmes, mais le faire de manière systématique peut risquer d'être coûteux en quantité de mémoire.

2.3.2 Allocation initiale des données

Les données utilisées par les calculs sont découpées en pages par le système. [10] compare plusieurs solutions d'allocation statique classiques, plus ou moins intéressantes selon les contextes.

Round Robin Les pages mémoire sont allouées tour à tour sur les différents nœuds, ce qui homogénéise les accès distants. Cependant, les données parculières à un thread risquent d'être allouées à distance, pénalisant ainsi l'exécution.

Prime Les pages sont également allouées tour à tour sur les différents nœuds, mais en faisant intervenir un nombre premier dans la périodicité. Cela permet d'éviter des situations pathologiques où la période de l'allocation *round robin* entre en résonance avec une certaine périodicité des allocations effectuées par l'application, si bien que tous les accès se retrouvent être distants.

First Touch Les pages sont allouées sur le nœud du premier processeur qui effectue un accès. Les données particulières à un thread sont donc correctement allouées sur le nœud où il s'exécute. Cependant, si un seul processeur initialise les données au début du programme, toutes les pages seront allouées sur le premier nœud, pénalisant les threads s'exécutant sur les autres nœuds.

L'allocation *round robin* est par exemple disponible sur SGI ORIGIN. LINUX 2.6 utilise actuellement l'allocation *first touch*, il est cependant possible pour l'application, depuis la très récente version 2.6.7, d'indiquer sur quel nœud devra se faire l'allocation.

2.3.3 Dynamicité des données

Ces allocations initiales ne répondent pas toujours bien aux besoins en données des threads. C'est d'autant plus le cas si l'ordonnanceur a déplacé des tâches d'un nœud à un autre : les données sur lesquelles elles travaillaient devraient suivre le mouvement, car ces tâches seraient alors pénalisées par l'effet NUMA.

Pour répondre aux défauts de répartition des pages, il existe quelques solutions dynamiques classiques, étudiées par exemple dans [11].

Réplication Bien souvent, tous les threads ont besoin d'accéder à certaines données telles que le programme lui-même et les données du problème. Si elles ne sont jamais modifiées, il peut être intéressant de dupliquer certaines pages sur chaque nœud, pour que l'accès à ces pages soit toujours local pour tous les threads. Si par contre ces pages sont modifiées sur un nœud, il faut invalider les autres réplicats.

Migration Pour compenser les défauts des allocations statiques décrites à la section précédente, il est possible d'utiliser les décomptes d'accès inter-cartes fournis par la machine. À l'aide d'heuristiques, il est alors possible d'essayer de prévoir l'utilisation future des pages à partir de leur utilisation passée. On peut ainsi décider de déplacer une page d'un nœud à un autre, pariant que ce sont les threads s'exécutant sur le nœud destination qui

en auront le plus besoin (car ils en avaient apparemment déjà beaucoup besoin par rapport à ceux des autres nœuds). Il est cependant difficile de trouver des heuristiques à la fois efficaces et rapides à calculer.

IRIX intègre depuis assez longtemps un mécanisme de migration automatique de pages indépendant des applications, mais il n'obtient en fait aucun réel gain de performances. [12] propose de le désactiver, et d'utiliser un algorithme plus proche à la fois de l'application et de l'ordonnanceur du système. Il obtient ainsi des performances beaucoup plus proches de l'optimal.

BULL développe actuellement un mécanisme générique de migration de pages sous LINUX pour ses machines NUMA, mais le coût d'analyse est encore élevé (14% du temps processeur !). Le gain en performances n'est pas pour autant très significatif. Les applications peuvent cependant effectuer elles-même des migrations qu'elles considèrent intéressantes.

À notre connaissance, il n'existe pas encore de moyen pour une application d'indiquer au noyau LINUX qu'elle désirerait voir des pages mémoire être répliquées de manière transparente.

2.3.4 Répartition des calculs par l'ordonnanceur

L'ordonnanceur du système n'a *a priori* aucune information sur l'affinité entre calculs et données car il n'a aucun dialogue avec l'application qui, elle, sait pourtant assez précisément en général quelles données elle utilisera, et quand. Faute de cette information, l'ordonnanceur est obligé d'utiliser des heuristiques qui ne sont pas toujours très adaptées.

Par exemple, le noyau LINUX 2.6 dispose d'une liste de tâches par processeur et doit équilibrer leurs charges respectives pour éviter que seul le processeur ayant lancé l'application exécute les calculs [9]. Un calcul de charge est effectué régulièrement pour les différents processeurs et pour les différents nœuds. Lorsqu'un processeur est inoccupé, il «vole» des tâches au processeur le plus chargé.

Ce vol n'est cependant pas systématique. Pour prendre un minimum en compte l'affinité qu'avaient ces tâches avec l'ancien processeur grâce aux effets de cache, l'ordonnanceur évitera de «voler» une tâche qui s'est exécutée assez longtemps sur ce nœud. Par ailleurs, les tâches qui n'effectuent que peu de calculs et attendent en fait souvent des évènements clavier ou réseau sont considérées comme «interactives», et seront aussi rarement déplacées, puisqu'elles ne chargent en fait que très peu le processeur.

Il est aussi prévu d'ajouter une notion de *homenode* aux tâches, c'est-à-dire d'éviter de voler une tâche au nœud où elle a été créée, pour ne pas perdre l'affinité qu'elle a avec ce nœud parce que ses données ont *a priori* été allouées dessus par la stratégie d'allocation initiale *first touch*. C'est cependant autant de risques que les calculs ne soient pas répartis sur toute la machine, lorsque toutes les tâches ont été lancées à partir d'un certain nœud.

2.4 Conclusion

Il existe donc de nombreuses solutions pour tenter de profiter au maximum de ces nouvelles machines hiérarchisées, par le placement des données ou leur migration, ainsi que par un ordonnancement plus ou moins approprié des calculs.

Cependant, aucune ne permet à l'application d'influencer la répartition des calculs et de la mémoire. Pourtant, dans le domaine du calcul hautes performances, le programmeur a en général une idée assez précise des répartitions adaptées.

Chapitre 3

Un ordonnanceur universel guidé par l'application

Notre proposition est d'impliquer l'application dans la répartition des calculs sur la machine. Nous proposons un modèle très générique de machine hiérarchisée, l'application fournit alors une organisation de ses calculs, et notre ordonnanceur peut enfin suivre ces indications pour répartir les calculs de manière adaptée.

3.1 Modélisation et abstraction de l'architecture

Pour être à même de gérer des machines aussi hiérarchisées que possible, nous les modélisons de manière hiérarchique à l'aide de listes de tâches. À chaque niveau de hiérarchie correspond une série de listes de tâches, une par élément de ce niveau. La figure 3.1 montre une machine très hiérarchisée et sa modélisation. Le niveau le plus haut (la machine) est modélisé par une liste générale de tâches. Le niveau suivant (les nœuds formés par les cartes fille) est modélisé par une liste de tâches par nœud. Chaque puce (**core**) est modélisée de la même manière par une liste de tâches, puis les processeurs et enfin les processeurs logiques.

Ce schéma n'est pas rigide : il est possible de l'étendre à volonté. Imaginons que l'on relie par exemple plusieurs de ces machines à l'aide d'un réseau SCI[13] qui permet de définir des zones de mémoire partagées de manière transparente. Il suffit simplement d'ajouter un niveau en haut de la hiérarchie.

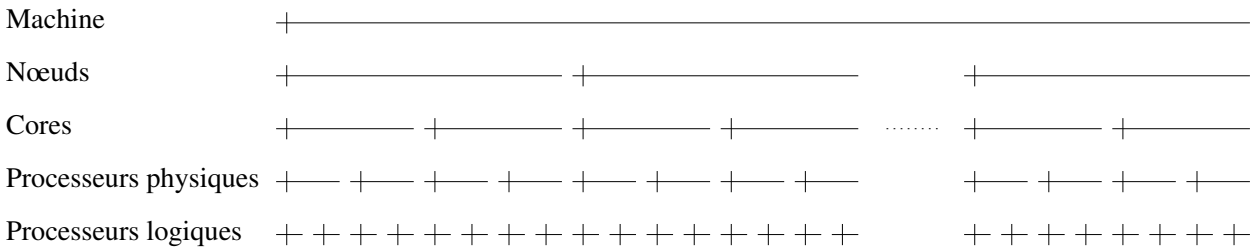
Chaque processeur a ainsi une série de listes de tâches qui le *couvrent*, depuis se liste propre jusqu'à la liste générale de la machine. Ce sont les listes qui modélisent les éléments architecturaux contenant ce processeur, dans l'ordre d'implantation physique.

A priori, l'architecture de la machine est suivie à la lettre, mais on peut par exemple vouloir ajouter des listes supplémentaires couvrant certaines listes.

Dans [14], WANG *et al.* proposent en effet de créer parfois des *clusters* artificiels pour limiter la contention. Dans notre situation, si un niveau i contient bien plus de listes que le niveau juste supérieur $i-1$, il peut être en effet intéressant, pour limiter la contention sur ces quelques listes du niveau $i-1$, d'insérer entre ces niveaux i et $i-1$ un niveau intermédiaire. Notons n_i le nombre de listes du niveau i et n_{i-1} le nombre de listes du niveau $i-1$, on choisira de constituer le niveau intermédiaire de l'ordre de $\sqrt{\frac{n_i}{n_{i-1}}}$ listes couvrant chacune $\sqrt{\frac{n_i}{n_{i-1}}}$ listes du niveau i. La figure 3.2 montre un exemple de tel niveau artificiel.

Dans le cas des machines où les nœuds sont reliés de manière torique sur la carte mère par exemple, on peut vouloir disposer des listes couvrant les voisinages de distance 1, de distance 2, etc pour profiter de la relative facilité d'accès mémoire entre ces nœuds. La position

(a) Deux processeurs hyperthreadés et un peu de mémoire sur une même puce avec un niveau de cache, cette puce est implantée en deux exemplaires sur une même carte fille avec un peu de mémoire, et l'on enfiche enfin de telles cartes sur une carte mère.



(b) Modélisation par des listes de tâches

FIG. 3.1 – Une machine très hiérarchisée et sa modélisation.

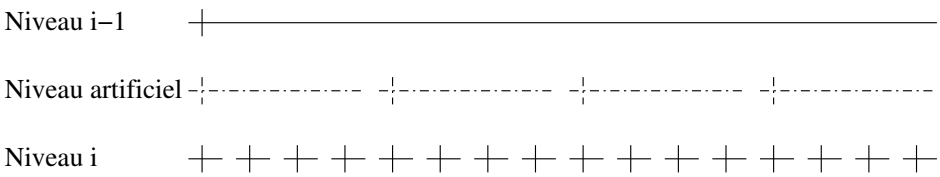


FIG. 3.2 – Exemple de niveau artificiel

relative de ces listes supplémentaires, par rapports aux listes correspondants à l'architecture de la machine, peut souvent être déterminé grâce à l'inclusion naturelle. Dans l'exemple des voisinages, il faudra cependant préciser un ordre, puisque ceux-ci s'intersectent mais ne sont pas inclus l'un dans l'autre.

Cette hiérarchisation des listes de tâches (inspirée de la décentralisation effectuée par le noyau, cf section 2.3.1) permet de limiter la contention au niveau de la liste générale de tâches. Elle permet également de lancer des tâches particulières à chaque processeur ou à chaque nœud par exemple. À l'inverse, il est possible de lancer des tâches générales qui peuvent être effectuées par n'importe quel processeur du système. Il serait même possible, quitte à modifier légèrement le mécanisme de réveil des tâches, de se passer complètement de verrouillage sur les listes de tâches de plus bas niveau.

3.2 Modélisation et abstraction de la structuration des calculs

Dès le début des années 80, avec l'émergence des réseaux de machines multiprocesseurs, OUSTERHOUT proposait déjà dans [15] de regrouper les threads et données par affinités sous forme de *gangs*. Ceux-ci devaient contenir un nombre fixe de threads, tout au plus égal au nombre de processeurs disponibles sur les machines du réseau considéré. Il suffisait alors, pour que les threads d'un *gang* s'exécute, de trouver sur le réseau une machine inoccupée, et d'y déplacer le *gang*, les threads pouvant tous être exécutés en parallèle puisqu'il y a suffisament de processeurs. Ainsi, calculs et données se retrouvaient très exactement sur la même machine. Il n'était cependant pas prévu que plusieurs *gangs* s'exécutent en même temps sur une même machine ; les processeurs n'étaient donc pas toujours tous occupés, si le *gang* était trop petit. [16] propose des algorithmes pour le cas d'un réseau hétérogène.

Nous proposons de reprendre quelque peu ce principe, appliqué ici à notre seule machine dont le caractère hiérarchique impose de s'inquiéter de la localisation des données et des calculs. Nous demandons donc à l'application de modéliser l'organisation de ses calculs.

Elle doit bien sûr, pour pouvoir profiter des différents processeurs de la machine, diviser ses calculs en tâches qui pourront être effectuées en parallèle. Mais pour indiquer les affinités respectives de ces tâches, elle les regroupe de manière hiérarchique dans des **bulles**.

Ces affinités peuvent être de plusieurs ordres, parfois complémentaires.

Partage de données Il se peut que des tâches travaillent en fait sur les mêmes données, s'ils travaillent sur un même bloc d'une matrice, par exemple. Il est alors intéressant d'essayer de rapprocher ces tâches pour profiter des effets de cache si elles parviennent à se retrouver sur la même puce, ou du moins éviter que certaines d'entre elles aient à passer par la carte mère pour aller chercher les données et soient alors pénalisées par le facteur NUMA.

Opérations collectives Il est aussi assez courant d'effectuer des opérations collectives. Une barrière de synchronisation permet par exemple de s'assurer que les tâches concernées ont toutes terminé la première partie d'un calcul avant d'entamer la seconde. Le calcul d'un `max` (ou tout autre opération associative et commutative) est également souvent utile. Dans les deux cas, s'assurer que ces tâches restent relativement proches permet une synchronisation plus locale donc plus efficace *a priori*.

Hyper-Threading On peut aussi vouloir exploiter l'*hyper-threading* (voir section 2.1.2) : si l'on sait que deux tâches sauront ne pas se gêner lorsqu'elles sont exécutées en parallèle sur les deux processeurs logiques d'un même processeur physique, il est intéressant de pouvoir les regrouper pour qu'elles soient *précisément* exécutées en parallèle.

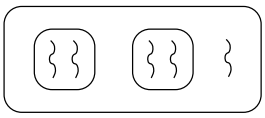


FIG. 3.3 – Regroupement de threads dans des bulles.

Supposons par exemple qu'une application a découpé le domaine de son calcul en deux morceaux, et que sur chaque morceaux, elle a découpé le calcul en deux tâches qui peuvent être effectuées en parallèle. Supposons qu'elle a également besoin d'une tâche à part, qui ne travaille pas sur des données particulière qui s'occupe d'afficher simplement les données d'entrée à l'écran, par exemple. La figure 3.3 montre une organisation adaptée à l'aide de bulles : les tâches de calcul travaillant sur les même données sont regroupées au sein d'un bulle. Ces deux bulles et la tâche d'affichage, puisqu'elles travaillent sur les mêmes données, sont elles-même regroupée dans une bulle plus grosse.

L'application n'a *a priori* pas à se soucier du tout de l'architecture de la machine, mais elle peut interroger l'environnement d'exécution. Elle peut par exemple demander le nombre de nœuds mémoire, c'est-à-dire le nombre de localisations possibles pour une donnée, pour savoir en combien de morceaux il sera utile de découper le domaine de calcul, par exemple. Elle peut également demander le nombre de processeurs disponibles sur ces nœuds pour avoir une idée du nombre de tâches distinctes qu'elle pourra lancer en parallèle sur chaque morceau du domaine.

3.3 Proposition : un ordonnanceur à bulles

Une fois les bulles créées par l'application, l'ordonnanceur de l'environnement d'exécution doit les fait évoluer pour répartir les tâches qu'elles contiennent.

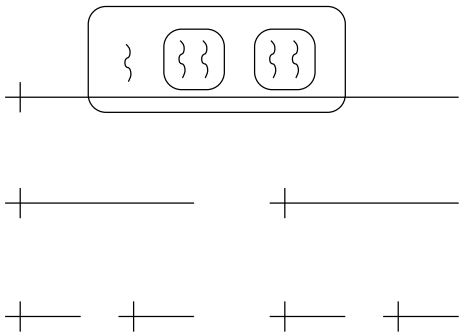
3.3.1 Évolution des bulles

L'objectif d'une bulle est d'«emmener» les tâches qu'elle contient jusqu'au niveau où il sera intéressant de les exécuter, parce qu'il englobe autant de processeurs qu'il y a de tâches à exécuter par exemple. Pour cela, la bulle «descend» dans la hiérarchie des listes de tâches jusqu'à arriver au niveau voulu. Là, elle «éclate», c'est-à-dire qu'elle libère les tâches et bulles qu'elle contient pour les laisser s'exécuter (ou descendre à leur tour).

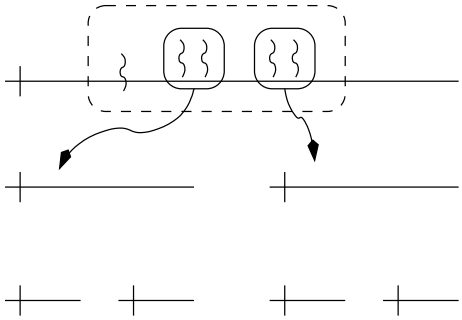
L'évolution de la bulle de la figure 3.3 est détaillée à la figure 3.4. La plus grande bulle éclate immédiatement sur la liste générale. La tâche d'affichage est alors libérée en même temps que les deux sous-bulles. Ces deux bulles peuvent alors descendre d'un niveau, où elles éclatent à leur tour, libérant leurs deux tâches de calcul. Celles-ci peuvent à leur tour descendre pour être exécutées sur des processeurs distincts.

Dans cet exemple, il se trouve que l'architecture correspond précisément à l'organisation des bulles. Si ce n'est pas le cas, il suffit que l'environnement d'exécution répartisse la profondeur de hiérarchies des bulle sur celle de la machine. Ainsi, une application peut se contenter de créer une hiérarchie de bulle de grande profondeur, elle pourra être correctement répartie sur toute machine ayant une profondeur moindre. L'application peut en tous cas *interroger* l'environnement pour déterminer le nombre de niveaux de hiérarchie et leur composition pour établir une hiérarchie de bulles de profondeur similaire.

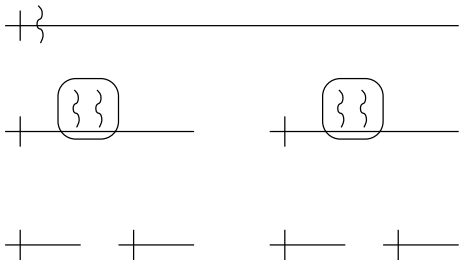
Les niveaux d'éclatement sont *a priori* précisés par l'application : c'est particulièrement utile dans le cas de l'Hyper-Threading, où l'application peut vouloir «envoyer» deux tâches



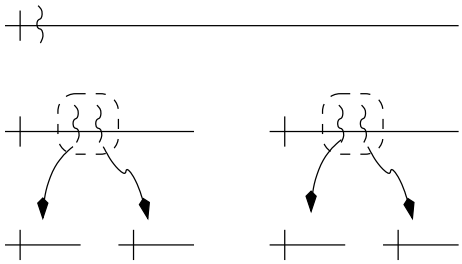
(a) La plus grande bulle a été placée sur la liste générale.



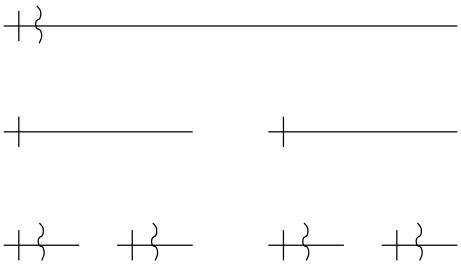
(b) Elle éclate, libérant la tâche de gestion qui peut déjà être exécutée et deux sous-bulles qui peuvent descendre dans la hiérarchie.



(c) Descente effectuée.



(d) Les deux sous-bulles éclatent en parallèle, libérant chacune deux tâches de calcul.



(e) Les tâches de calcul sont bien réparties et peuvent démarrer en parallèle.

FIG. 3.4 – Évolution des bulles.

sur les deux processeurs logiques d'un même processeur physique. Ils peuvent également correspondre à des niveaux où se situent les puce mémoire, pour indiquer où doivent aller une bulle travaillant sur une même zone de données. Si l'application n'a pas de telles particularités, elle peut se contenter d'indiquer des niveaux consécutifs que l'environnement s'occupera se ré-écheloner arbitrairement sur les niveaux physiques de la machines.

3.3.2 Notion de priorités

A priori, les tâches situées sur la liste particulière à un processeur sont plus prioritaires que celles situées sur la liste du niveau juste supérieur. Cependant, dans notre exemple de bulle (figure 3.3), il peut être intéressant que l'application puisse indiquer que la tâche d'affichage doit être exécutée dès que possible.

Pour cela, les bulles et les tâches ont chacune une priorité représentée par un entier. Lorsqu'un processeur cherche une tâche à exécuter, il devra donc parcourir, des plus *locales* (*i.e.* de niveau le plus bas) aux plus *globales*, les différentes listes qui le «couvrent», à la recherche de la tâche de priorité maximale. Il devra alors exécuter cette tâche même si d'autres tâches moins prioritaires résident sur des listes plus locales.

En fait, l'environnement d'exécution a bien souvent lui aussi besoin que certaines tâches soient même plus prioritaires que les tâches utilisateurs. Il lui suffit pour cela de donner à ces tâches des priorités que l'application n'a pas le droit d'indiquer.

Cette notion permet également à l'application de donner une certaine priorité aux tâches par rapport aux bulles En effet, en donnant aux tâches contenues dans les bulles une priorité plus grande que celle de ces dernières, l'application peut empêcher les bulles d'éclater avant que toutes les tâches des bulles précédemment éclatées soient effectuées. C'est par exemple utile si l'application a créé un grand nombre de bulles encapsulant des tâches qui devraient être effectuées une par une sans interruption.

3.3.3 Rééquilibrage

Comme tout système de répartition des tâches, il n'est pas dit que cette descente de bulle équilibrera correctement la charge de calcul sur les différents processeurs de la machine (sauf si les temps de terminaison des tâches sont très réguliers). Il est donc possible qu'un processeur ait effectué toutes les tâches disponibles sur les différentes listes qui le couvrent, tandis que d'autres processeurs continuent à exécuter des tâches de leurs propres listes. Un certain rééquilibrage est alors nécessaire.

Pour cela, il est possible d'organiser du vol de travail, tel qu'il est réalisé dans le noyau LINUX (voir section 2.3.4), en maintenant des compteurs de charge pour voler du travail aux listes les plus chargées. Cependant, pour éviter de trop perdre de possibilités d'affinités, il est peut-être préférable de prendre le temps de reformer la bulle qui contenait ce travail que l'on veut voler. Ainsi, on peut la faire remonter jusqu'à un niveau couvrant à la fois la liste où étaient situé ce travail et le processeur inoccupé, et l'y faire éclater. Elle fournit ainsi non seulement du travail pour ce processeur, mais aussi pour les processeurs qui lui sont proches dans la hiérarchie ; il y a de grandes chances que sans cela, ceux-ci auraient été bientôt eux aussi inoccupé à cause du même déséquilibre.

3.4 Conclusion

Ce mécanisme de bulles nous permet de répartir les calculs tout en prenant le plus possible en compte les affinités entre tâches que l'application nous indique.

Pour l'instant, nous comptons sur le système d'exploitation sous-jacent (le noyau LINUX) pour allouer les données aux bons endroits. Il se trouve que cela marche en général assez bien, car LINUX utilise une allocation *first touch* (décrite à la section 2.3.2). Si les tâches allouent elles-même les zones de données sur lesquelles elles vont travailler — c'est souvent le cas —, ces zones seront correctement allouées par LINUX sur le nœud où s'exécutent ces tâches. Si par contre toutes les tâches ont par exemple besoin d'accéder à des données d'entrée situées sur le premier nœud, cet accès sera lent pour la plupart, même si effets de cache limitent les pertes de performances.

Une perspective possible serait d'ajouter aux fonctions d'allocation mémoire des attributs permettant à l'application de préciser par exemple quelles tâches (ou tâches d'un bulle) utiliseront la zone allouée, et comment (lecture seulement ou lecture/écriture).

Chapitre 4

Détails d'implantation

Cette proposition a été implantée au sein de la bibliothèque de threads MARCEL. Après avoir présenté MARCEL et sa récente modularisation (pour des raisons de portabilité et de fonctionalités), nous schématisons le fonctionnement de l'implémentation.

4.1 La bibliothèque de processus légers MARCEL

MARCEL a été développé au sein du projet PM^2 [2]. Ce projet consiste en un environnement d'exécution destiné aux applications de calcul hautes performances à la fois multithreads et distribué, dont le paradigme est l'appel de fonctions à distance (*Remote Procedure Call* ou RPC). MARCEL est le composant dédié à la gestion des processus légers.

Les deux objectifs principaux de cette bibliothèques ont toujours été d'allier la performance à la capacité de gérer simultanément plusieurs milliers de processus légers. Au départ une simple bibliothèque de processus utilisateurs qui ne pouvait pas exploiter les machines SMP, V. DANJEAN l'a modifiée pour contenir un ordonnanceur de processus utilisateurs mixte [17] capable de profiter de ces machines. Il a ensuite introduit un mécanisme d'activations [18, 19] pour permettre aux applications d'effectuer des appels bloquants sans précautions particulières.

Ces versions de MARCEL ne se préoccupent cependant pas de la nature hiérarchique de la machine, obtenant ainsi dans certains cas de piètres performances. L'implantation de notre proposition devrait permettre de résoudre en partie ces problèmes de performances.

4.2 Modularité

Pendant le stage, MARCEL a vu sa structure complètement réorganisée de manière modulaire. Ainsi, l'ordonnanceur — qui a la charge de choisir laquelle des tâches prêtes à être exécutée doit remplacer la tâche en cours — a été extrait du reste du code. L'interface restant entre les deux se réduit à quelques fonctions et structures, elle est en fait inspirée du noyau LINUX. Nous avons également repris l'ordonnanceur même du noyau. Il a fallu l'adapter à un fonctionnement en mode utilisateur, et surtout à la hiérarchie de listes utilisée dans notre modèle (dans le noyau LINUX, il y a simplement une liste de tâches par processeur, sans liste commune).

Cette structure modulaire rend envisageable que l'auteur d'une application prenne le temps d'écrire lui-même un nouvel ordonnanceur en implémentant les quelques fonctions et structures nécessaires. Il suffira alors de le placer à la place du nôtre.

D'autres éléments ont été repris du noyau LINUX.

– Les verrous rotatifs (*spinlocks*) ont été repris tels quels, car leur efficacité a été particulièrement soignée par la communauté LINUX, ceci sur les différentes architectures supportées. Cela facilite grandement le portage de MARCEL sur différentes architectures, pourvu que LINUX lui-même ait été porté.
– La structure de liste de tâches a été reprise. Elle possède notament un mécanisme pour pouvoir rechercher une tâche de priorité maximale en O(1), ce qui est particulièrement précieux pour conserver un ordonnanceur très efficace.

Cela a permis entre autres de porter MARCEL pour un processeur qui nous était jusque maintenant inconnu : le processeur Itanium. Pour effectuer ce portage, le CEA nous a fourni des comptes distants sur la machine Fame construite, hébergée et entretenue par Bull.

4.3 Fonctionnement général

MARCEL permet à une application de lancer assez simplement une tâche à effectuer : il suffit d'appeler la fonction `marcel_create` en lui passant l'adresse de la fonction que la tâche devra exécuter et un éventuel paramètre qui sera fourni à la fonction. Cette tâche peut alors démarrer immédiatement. Une fois qu'elle est terminée, on peut récupérer sa valeur de retour à l'aide de `marcel_join`.

Notre proposition ajoute la possibilité de *créer* des tâches, mais sans les lancer. L'application peut ainsi, plutôt que de lancer ces tâches, les insérer dans des bulles de manière hiérarchique. Une fois l'organisation des tâches ainsi décrite, elle peut *lancer* les bulles les plus externes. L'ordonnanceur s'occupe alors de la descente dans les listes de tâches et l'éclatement hiérarchique, et les tâches ne sont réellement démarrées que lorsque les bulles les plus internes éclantent.

Par défaut, ces bulles seront insérées sur la liste de tâches générale, permettant d'exploiter toute la machine, mais l'application peut éventuellement choisir des listes de tâches à un niveau inférieur, pour ne consommer qu'une partie des resources de la machine, par exemple.

Pour obtenir le comportement décrit à la section 3.3, les processeurs — lorsqu'ils cherchent à exécuter une nouvelle tâche — exécutent l'algorithme d'ordonnancement décrit à l'annexe A. Cet algorithme essaie de verrouiller le moins possible de listes de tâches. Les quelques verrous sont également conservés le moins longtemps possible, afin de limiter la contention sur ces listes[1].

Une fonctionalité envisagée mais non encore implémentée est de confier une partie de l'ordonnancement à l'application. Elle n'aurait pas besoin de réécrire un ordonnanceur complet, mais seulement une fonction de choix indiquant s'il faut faire descendre ou éclater une bulle, ou exécuter une tâche.

[1] En fait, quitte à ralentir légèrement le réveil d'un tâche placée sur la liste de tâches particulière à un processeur, il serait possible de se passer complètement de verrou pour les listes de tâches particulières aux processeurs

Chapitre 5

Évaluation des performances

Bien qu'elle n'ait pas encore été optimisée et que l'interface de programmation fournie à l'application n'est pas encore stabilisée, une première implémentation de ce mécanisme de bulles est opérationnelle, il est donc intéressant de mesurer le gain de performances que l'on obtient en utilisant ses possibilités, pour vérifier que le coût de création des bulles n'est pas trop important.

5.1 Environnement de test

Les tests présentés ci-après ont été réalisés sur deux machines assez différentes :

Joe est une machine SMP équippée de deux processeurs Pentium 4 XEON™ cadencés à 2.7GHz, c'est-à-dire un total de quatre processeurs logiques grâce aux capacités de Multi-Threading, pour un total d'1Go de mémoire vive.

Fame est une machine NUMA construite par BULL qui possède quatre nœuds reliés par un réseau Quadrics. Chaque nœud est équippé de quatre processeurs Itanium II™ et de 16Go de mémoire vive. Le support de cette machine par MARCEL est en fait très récent et n'est pas encore complètement stabilisé.

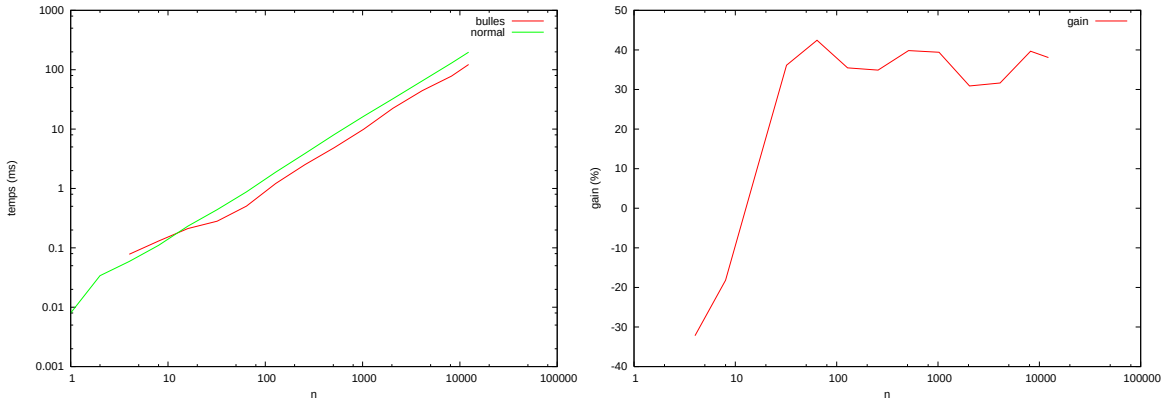
5.2 Un exemple de test

Pour vérifier que le surcoût apporté par les bulles n'est pas trop important, nous avons testé le programme *sumtime*. Ce programme calcule la somme des entiers de 1 à n (donné) de manière dichotomique récursive, ce qui est peu efficace, mais intéressant pour ses exigences en création et synchronisation de threads. Ce programme définit en effet une fonction $f(i, j)$ qui calcule la somme des entiers de i à j de manière récursive : si $i = j$, le résultat est i, sinon deux threads de calcul sont lancés : un calculant la somme des entiers de i à $\lfloor \frac{i+j}{2} \rfloor$ en appelant $f(i, \lfloor \frac{i+j}{2} \rfloor$, et l'autre la somme des entiers de $\lfloor \frac{i+j}{2} \rfloor + 1$ à j en appelant $f(\lfloor \frac{i+j}{2} \rfloor + 1, j)$. $f(i, j)$ doit attendre que ces deux threads terminent avant de récupérer leur résultat, et renvoyer la somme.

Finalement, $2n - 1$ threads auront été lancés, et $n - 1$ réductions auront été effectuées, tout ceci en parallèle autant que possible.

Dans la version originale, tous ces threads et synchronisations sont effectués assez aléatoirement sur les différents processeurs.

Dans la version «bulle» que nous proposons, nous regroupons les calculs sur les processeurs, selon le nombre d'entre eux disponibles sur la machine : sur une machine possédant



(a) (b)

FIG. 5.1 – Temps d'exécution de *sumtime* sur *Joe*

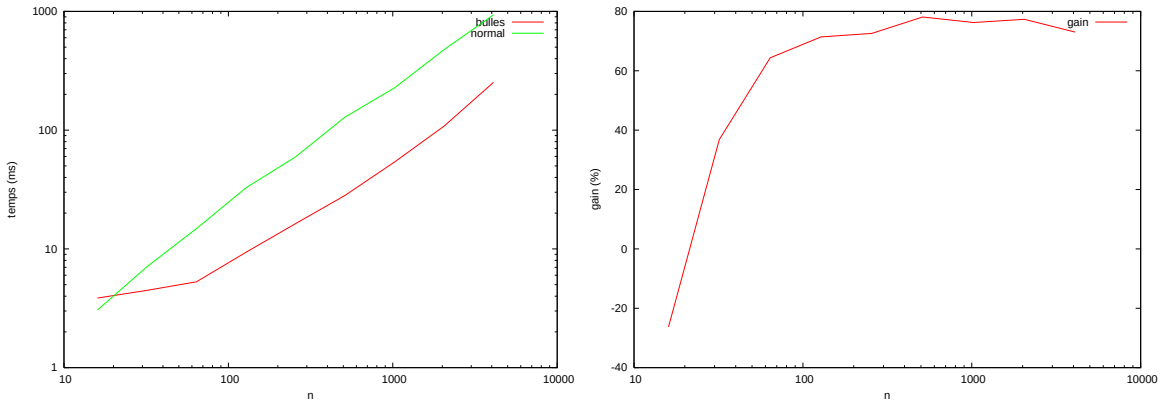
deux processeurs, nous regroupons chaque moitié du travail dans deux bulles séparées : nous regroupons le thread chargé de calculer la somme de 1 à $\lfloor \frac{n}{2} \rfloor$ et tous les threads qui seront lancés pour calculer cette somme dans une même bulle. De la même façon, le thread chargé de calculer la somme de $\lfloor \frac{n}{2} \rfloor + 1$ à n et sous les threads lancés pour calculer cette somme seront regroupés dans une autre bulle. En indiquant que l'on désire que ces bulles éclatent au niveau des processeurs, on localise ainsi la moitié du lancement des threads et synchronisations sur chaque processeur.

De manière récursive, sur une machine à quatre processeurs, chaque quart du travail est regroupé dans une même bulle, et ces bulles sont regroupées deux à deux dans deux plus grosses bulles. Et ainsi de suite : il n'y a nul besoin de réécrire le programme selon la machine à disposition, il n'y a en fait même pas besoin de le recompiler. En effet, ce programme est à même de calculer automatiquement *à l'exécution* une répartition des calculs selon le nombre de processeurs de la machine.

La figure 5.1 montre les temps d'exécution obtenus sur la machine *Joe*, possédant quatre processeurs. L'échelle est logarithmique pour pouvoir observer à la fois le comportement pour les petites et les grandes valeurs de n. Pour de petites valeurs de n, le coût de création des bulles ne permet pas d'obtenir de gain de performances. Par contre, dès $n = 16$, on obtient un gain entre 30 et 40% en temps.

La figure 5.2 indique les temps d'exécution obtenus sur la machine *Fame*, qui possède seize processeurs répartis quatre par quatre sur quatre nœuds possédant leur propre mémoire. On observe de nouveau que pour de petites valeurs de n, le coût de création des bulles empêche un gain intéressant. Mais dès $n = 32$, on atteint un gain en temps de 40%, celui-ci se stabilise même à près de 80%.

Évidemment, nous continuerons à effectuer de tels tests à la fois sur de petites applications telles que celle-ci, et sur de véritables applications, telles que celles du CEA.



(a) (b)

FIG. 5.2 – Temps d'exécution de *sumtime* sur *Fame*

Chapitre 6

Conclusion

6.1 Bilan

Au cours de ce mémoire de DEA, nous avons exposé une partie des travaux réalisés au sein de l'équipe RUNTIME du LaBRI, visant à permettre à MARCEL, bibliothèque de processus léger de l'environnement PM^2, de profiter pleinement des capacités croissantes des machines hiérarchisées.

6.1.1 Un passage à l'échelle

Les machines que les constructeurs proposent possédant de plus en plus de processeurs implantés de manière hiérarchisée, il était devenu nécessaire d'éliminer les points de contention de MARCEL. L'utilisation d'une hiérarchie de listes de tâches reproduisant fidèlement la structure physique de la machine permet par exemple de localiser le verrouillage nécessaire pour se prévenir des accès concurrents.

L'arrivée prochaine sur le marché des nouvelles machines ultra-hiérarchisées, avec l'apparition par exemple de la nouvelle technologie Multi-Core, est en fait déjà préparé.

6.1.2 Un ordonnanceur à bulle

Pour pouvoir à la fois répartir et regrouper par affinités les calculs d'une application, la notion de bulle a été définie. Elle permet à l'application de contrôler assez finement la manière dont sont répartis les calculs sur la machine en «envoyant» aux différents niveaux hiérarchiques de la machine des groupes de tâches qu'elle a définis.

Les gains sont à la fois en effet de cache et en adéquation des accès mémoire à la hiérarchie physique de la machine, ce qui évite d'avoir à recourir à un mécanisme de migration automatique des données.

6.1.3 Un dialogue avec l'application

Les programmeurs d'applications se mettraient volontiers aux commandes de l'ordonnanceur. Au-delà des simples bulles, c'est un véritable dialogue qui commence avec l'application pour que celle-ci contrôle la répartition des calculs et des données. Ce dialogue est plutôt simple, il n'y a nullement besoin de comprendre les mécanismes internes de MARCEL pour savoir comment obtenir les meilleurs performances de manière portable.

6.1.4 Autres voies

Un complément à l'idée de *bulle* — qui *regroupe* des tâches par affinités — est celle de *répulsion* : pour ne pas risquer de ne pas suffisament étaler les calculs et laisser des processeurs inoccupés, il serait possible que l'application indique que certaines bulles se «repoussent». Par effet ressort, elles seraient ainsi réparties correctement sur toute la machine. Cette solution n'a pas encore été approfondie car sa stabilité n'est pas forcément claire.

6.2 Travaux futurs

Notre proposition ouvre de nombreuses perspectives.

6.2.1 Amélioration

Non seulement l'implémentation doit désormais être optimisée, la proposition elle-même est améliorable. Il subsiste plusieurs points encore obscurs, tels que la reformation des bulles pour rééquilibrer la charge, la forme exacte de communication entre l'application et l'environnement. Il manque encore également d'autres tests de performances.

6.2.2 Allocation mémoire

L'application peut désormais indiquer l'affinité entre tâches. Il pourrait ensuite être intéressant pour elle d'indiquer aussi l'affinité entre les tâches contenues dans une bulle et les zones de mémoires. En effet, on conçoit aisément qu'elle peut tout à fait indiquer lors de l'allocation d'une zone de mémoire l'usage qui en sera fait : zone de données en lecture seule pour toute une bulle de tâches, zone strictement locale à un tâche, zone d'échange producteur / consommateur, ... autant de cas particuliers qu'il serait alors possible d'optimiser.

6.2.3 Une liberté encore plus grande pour l'application

Quel que soit le système, les programmeurs ont toujours du à un moment ou à un autre passer par des artefacts pour obtenir du système le comportement voulu. Plutôt que de se risquer à des opérations aussi peu portables, l'idéal serait de donner un certain contrôle des décisions du système au programmeur, en lui proposant d'implémenter les fonctions de choix critiques pour les performances.

6.2.4 Des «comportements» des tâches

Les tâches ont plusieurs natures : calculs, communications, réduction de résultats,... Le noyau LINUX distingue par exemple les processus dits *interactifs* des processus normaux, pour améliorer leur réactivité. Il est pour cela obligé de se baser sur des heuristiques observant leur comportement, ce qui a évidemment des effets de bords sur certaines applications.

L'application pourrait directement indiquer à l'environnement le type d'une tâche, permettant ainsi à l'ordonnanceur de savoir par exemple qu'une tâche ne s'exécutera que très peu de temps avant de redonner la main.

6.2.5 Support Posix

Il est en projet que MARCEL présente une interface Posix, permettant ainsi de l'utiliser facilement pour toutes les applications qui utilisent ce standard reconnu.

Il serait possible de créer des bulles en fonction de la manière dont sont créés les threads : si le thread principal crée deux fils, qui eux-mêmes en créent chacun deux autres, on peut supposer qu'il sera avantageux de reprendre cette hiérarchie en regroupant les quatre threads finaux deux par deux dans deux bulles, celles-ci étant alors regroupées avec les deux fils du thread principal dans deux autres bulles, et une bulle finale regrouperait le tout.

Annexe A

Algorithme d'ordonnancement

Cet algorithme est appelé par une tâche lorsqu'elle veut donner la main à une autre tâche.

- Les listes de tâches sont examinées une par une à partir de celle de plus bas niveau (particulière au processeur), à la recherche de celle où l'on peut trouver une tâche de priorité maximale. Cette recherche est effectuée *sans verrouillage*, pour éviter une contention sur les verrous des listes.
- Si cette recherche échoue, c'est qu'il n'y a plus de tâche à effectuer, la main est rendue à la tâche précédente si elle le permet, le processeur devient inoccupé sinon.
- Sinon,
- La liste trouvée est verrouillée en même temps que celle où était déposée la tâche précédente.
- On essaie d'enlever la première tâche (ou bulle) ayant la priorité maximale.
- S'il n'en existe plus (ce qui peut arriver si un autre processeur l'a enlevée entre la recherche faite sans verrouillage et le verrouillage proprement dit), on déverrouille les deux listes et l'on retourne à la première étape.
- Sinon,
- Si le niveau de la liste sur laquelle se trouve la tâche (ou la bulle) est plus haut que le niveau d'éxécution de la tâche (ou d'éclatement de la bulle), elle est descendue sur une liste de niveau suffisament bas, les listes sont déverrouillées et l'on retourne à la première étape.
- Sinon, si c'est une bulle, elle est éclatée : toutes les tâches ou bulles contenues sont alors insérées dans la liste. On déverrouille les listes et l'on retourne alors à la première étape.
- Sinon (c'est une tâche),
- On dépose la tâche précédente sur sa liste.
- On bascule sur la nouvelle tâche.
- On déverrouille finalement les deux listes.

Le schéma de verrouillage pourrait être encore amélioré. Il suffirait pour cela de ne déposer la tâche précédente sur sa liste (permettant éventuellement à d'autres processeurs de l'exécuter) qu'une fois le basculement effectué. Il serait alors possible de ne jamais avoir à verrouiller deux listes en même temps.

Bibliographie

[1] http://www.openmp.org/.

[2] NAMYST (R.), *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. Thèse de doctorat, Univ. de Lille 1, janvier 1997.

[3] PETRINI (F.), KERBYSON (D. J.) et PAKIN (S.), « The case of the missing supercomputer performance : Achieving optimal performance on the 8,192 processors of asci q », dans *Super Computing Conference*, p. 55, novembre 2003.

[4] MARR (D. T.), BINNS (F.), HILL (D. L.) *et al.*, « Hyper-threading technology architecture and microarchitecture », *Intel Technology*, vol. 6 issue 01, 2002. http://www.intel.com/technology/itj/2002/volume06issue01/.

[5] ROBERSON (J.), « ULE : A modern scheduler for FreeBSD ». Rapport technique, The FreeBSD Project, jeff@FreeBSD.org.

[6] OI (H.) et RANGANATHAN (N.), « Utilization of cache area in on-chip multiprocessor », *Microprocessors and Microsystems*, vol. 24, 2000, p. 429–436.

[7] KOELBEL (C.), LOVEMAN (D.), SCHREIBER (R.) *et al.*, « The high performance fortran handbook », 1994.

[8] DANDAMUDI (S.) et CHENG (S.), « Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems », *Systems Architecture*, vol. 43, 1997, p. 491–511.

[9] « Linux scalability effort ». http://lse.sourceforge.net.

[10] IYER (R.), WANG (H.) et BHUYAN (L. N.), « Design and analysis of static memory management policies for CC-NUMA multiprocessors », *Systems Architecture*, vol. 48, 2002, p. 59–80.

[11] WILSON (K. M.) et AGLIETTI (B. B.), « Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C », dans *Proceedings of the 2001 ACM/IEEE conference on SuperComputing*, p. 33–33, Denver, Colorado, novembre 2001. ACM.

[12] NIKOLOPOULOS (D. S.), POLYCHRONOPOULOS (C. D.), PAPATHEODOROU (T. S.) *et al.*, « Scheduler-activated dynamic page migration for multiprogrammed dsm multiprocessors », *Parallel and Distributed Computing*, vol. 62, décembre 2002, p. 1069–1103.

[13] SOLUTIONS (D. I.), « The dolphin sci interconnect », février 1996. http://www.dolphinics.com/pdf/whitepaper/T-WhitePaper.pdf.

[14] WANG (Y.-M.), WANG (H.-H.) et CHANG (R.-C.), « Clustered affinity scheduling on large-scale NUMA multiprocessors », *Systems Software*, vol. 39, 1997, p. 61–70.

[15] OUSTERHOUT (J. K.), « Scheduling techniques for concurrent systems », dans *Third International Conference on Distributed Computing Systems*, p. 22–30, octobre 1982.

[16] AL-SAQABI (K.), SARWAR (S.) et SALEH (K.), « Distributed gang scheduling in networks of heterogeneous workstations », *Computer Communications*, vol. 20, 1997, p. 338–348.

[17] DANJEAN (V.), « Introduction d'un ordonnanceur de processus légers mixte dans pm2 pour une exploitation efficace des architectures multiprocesseurs ». Rapport de stage mim1, ReMap - LIP - Lyon, 1998.

[18] DANJEAN (V.), NAMYST (R.) et RUSSELL (R.), « Integrating kernel activations in a multithreaded runtime system on Linux », dans *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, vol. 1800 (coll. *Lect. Notes in Comp. Science*), p. 1160–1167, Cancun, Mexico, mai 2000. En conjonction avec IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag. `http://www.ens-lyon.fr/\~bouge/Biblio/Danjean/DanNamRus00RTSPP.ps.gz`.

[19] DANJEAN (V.), NAMYST (R.) et RUSSELL (R.), « Linux kernel activations to support multithreading », dans *Proc. 18th IASTED International Conference on Applied Informatics (AI 2000)*, p. 718–723, Innsbruck, Austria, février 2000. IASTED. `http://www.ens-lyon.fr/\~bouge/Biblio/Danjean/DanNamRus00IASTED.ps.gz`.

[20] WANG (Y.-M.), WANG (H.-H.) et CHANG (R.-C.), « Hierarchical loop scheduling for clustered NUMA machines », *Systems and Software*, vol. 55, 2000, p. 33–44.

[21] FEITELSON (D. G.) et RUDOLPH (L.), « Evaluation of design choices for gang scheduling using distributed hierarchical control », *Parallel and Distributed Computing*, vol. 35, 1996, p. 18–34.