HAL
archives-ouvertes.fr

Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications

Samuel Thibault

▶ **To cite this version:**

Samuel Thibault. Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications. Réseaux et télécommunications [cs.NI]. Université Sciences et Technologies - Bordeaux I, 2007. Français. tel-00322881

HAL Id: tel-00322881
https://tel.archives-ouvertes.fr/tel-00322881

Submitted on 19 Sep 2008

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.

N° d'ordre : 3506

THÈSE

présentée à

L'Université Bordeaux 1

École Doctorale de Mathématiques et Informatique

par Monsieur Samuel THIBAULT

pour obtenir le grade de

Docteur

Spécialité : Informatique

Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications

Soutenue le : 6 Décembre 2007

Après avis de :

M. Jean-François	MÉHAUT	Professeur des Universités	Rapporteur
M. Thierry	PRIOL	Directeur de Recherche INRIA	Rapporteur

Devant la commission d'examen formée de :

M. Serge	DULUCQ	Directeur adjoint LaBRI	Président
M. Panagiotis	HADJIDOUKAS	Université d'Ioannina, Grèce	Examinateur
M. Jean-François	MÉHAUT	Professeur des Universités	Rapporteur
M. Raymond	NAMYST	Professeur des Universités	Directeur de thèse
M. Thierry	PRIOL	Directeur de Recherche INRIA	Rapporteur
M. François	ROBIN	Directeur délégation ANR "Calcul Intensif"	Examinateur
M. Pierre-André	WACRENIER	Maître de conférence	Directeur de thèse

À toutes celles et ceux qui nous ont passé le relai,

Remerciements

Je souhaite tout d'abord remercier en particulier Raymond Namyst et Pierre-André Wacrenier de m'avoir accueilli et guidé pendant ces trois années de thèse. Je remercie également les membres de mon jury, Serge Dulucq, Panagiotis Hadjidoukas (qui est venu de Grèce !), François Robin, et en particulier Jean-François Méhaut et Thierry Priol qui ont bien voulu prendre le temps de relire ce mémoire. Je remercie enfin les valeureux membres de l'équipe qui l'ont pré-relu, et Brigitte et Sylvie pour leur dévouement.

J'ai ensuite envie de remercier beaucoup de monde, il est donc impossible de citer tout le monde et pour toutes les raisons. Je vais donc essayer d'en oublier le moins possible, et surtout, je ne vais surtout pas chercher à évoquer forcément les raisons les plus sérieuses.

Je remercie tout d'abord Virginie et ses 3 —pardon, 4— petits monstres, qui égaient toujours autant les barbecues, ainsi que le robot de la piscine qui a la gentillesse de nettoyer après nos batifolages. Ensuite, par ordre alphabétique pour ne pas faire de jaloux, je remercie Alex et ses discussions à bâtons rompus, Babeth et ses animes, Brice et Émilie pour la soirée mémorable de dépendaison de crémaillère et vidage de Chartreuse et... je sais plus, Cécile pour avoir le courage de reprendre le flambeau, Christophe pour son téléphone portable et les conseils mangas, François 1, qui a écouté patiemment toutes les propositions débiles de noms pour PIOMan, François 2 qui a joué le rôle de Beta testeur 2, François 3 pour son joli record à battre, Cédric qui m'a initié aux méandres du Cell, Guillaume pour ses trouvailles cinématograhiques telles que la scène de combat la plus ridicule/kitsch/drôle/?? du monde, Jéjé et ses nouvelles sur les cartes graphiques, Marc et la vraie vie de la programmation parallèle, Marie-Christine pour son débroussaillage d'OpenMP, Mathieu pour avoir joué le rôle de Beta testeur 1, Nath pour les adaptateurs, Olivier, public fidèle, Pierre-André pour son flegme, Raymond pour Le Magnifique, Stéphanie et son gâteau à la citrouille, Sylvain, pour avoir été Beta testeur 3 et pour le sac à vélo, et enfin Vince qui m'a refourgué le bébé avec enthousiasme. Bien sûr, je remercie également les membres de l'équipe ScalApplix, toujours prêts à lever le verre, et plus généralement tout le LaBRI pour l'ambiance générale bien sympathique.

Que de chemin parcouru depuis les temps du minitel familial, et notamment le jour où l'on a ramené de chez ma grand-mère un Amstrad 6128 qui était sur le point de partir à la casse... C'est l'occasion pour moi ici de remercier ceux qui l'ont jalonné.

Pour les jalons académiques, je tiens à remercier Mme Janvier, Mme Termeau, Mr Chalon, Mr et Mme Labrousse, Mr Roland, Mr Devernay, et bien sûr tous les enseignants du MIM, notamment Dany, Xoff, Bob, F2D et surtout BigRay, Oli et Vince, ainsi que Bob Russell. Une mention spéciale également à mes professeurs de Français, de Philosophie et d'Anglais, je n'avais pas réellement mesuré à l'époque combien leurs enseignements me seraient utiles plus tard, pour rédiger articles, transparents et mémoire. Je remercie également Olivier et Nath, relecteurs assidus et exigeants.

Je tiens également à remercier ceux qui ont posé des jalons moins académiques mais non

moins enrichissants, à commencer par Gérard Laumonier, Paul Dubonnet, Jean Boucherie, ainsi que les professeurs de technologie et les documentalistes qui nous ont laissé bricoler allègrement les moyens informatiques du collège. Je remercie également mes compagnons de bricolages Gaby, Couhien, Jissé, Séb, Julio, et plus généralement la promo MIM 2001.

Il y a aussi des jalons qui ne sont pas académiques du tout, mais tout aussi enrichissants. Depuis les potes du bac à sable au DEA, je remercie donc Guillaume, Sylvain, Maxime, Yoann, Laurent, Bin, David et Pierre, ainsi que les geeks de #sos et les dinoïstes qui m'ont montré qu'on pouvait vraiment troller sur tout et surtout n'importe quoi. Je remercie également les étudiants de Bordeaux pour mettre autant d'animation dans les news. Je remercie par ailleurs mes profs de batterie, ainsi que les différents orchestres et groupes (notamment LA fanfare !) avec lesquels j'ai pu jouer histoire de respirer autre chose que l'air des ventilos, mon prof de trombone pour m'avoir initié aux secrets des bandas, et les voisins pour avoir supporté mes gammes. Je remercie la SNCF et sa carte 12-25, je ne remercie pas voyages-sncf.com (mais je remercie bahn.de). Enfin, je remercie Marianne pour m'avoir initié aux secrets de la vie anglaise, ce qui s'est révélé bien pratique pour pouvoir m'installer à Cambridge. Je remercie enfin la Rose des Sables pour son excellent couscous.

Enfin, je remercie toute ma famille, pour m'avoir laissé squatter l'ordinateur familial de manière honteuse pendant toutes ces années, et de ne s'être pas trop inquiétée de voir les cernes laissées par les longues soirées (nuits ?) passées à coder !

Résumé : La tendance des constructeurs pour le calcul scientifique est à l'imbrication de technologies permettant un degré de parallélisme toujours plus fort au sein d'une même machine : architecture NUMA, puces multicœurs, SMT. L'efficacité de l'exécution d'une application parallèle irrégulière sur de telles machines hiérarchiques repose alors sur la qualité de l'ordonnancement des tâches et du placement des données, pour éviter le plus possible les pénalités NUMA et les défauts de cache. Les systèmes d'exploitation actuels, pris au dépourvu car trop généralistes, laissent les concepteurs d'application contraints à « câbler » leurs programmes pour une machine donnée.
Dans cette thèse, pour garantir une certaine portabilité des performances, nous définissons la notion de *bulle* permettant d'exprimer la nature structurée du parallélisme du calcul, et nous modélisons l'architecture de la machine cible par une hiérarchie de listes de tâches. Une interface de programmation et des outils de débogage de haut niveau permettent alors de développer simplement des ordonnanceurs dédiés, efficaces et portables. Différents ordonnanceurs mettant en œuvre des approches variées ont été développés, en partie notamment par des stagiaires encadrés au sein de l'équipe, ce qui montre à la fois la puissance et la simplicité de l'interface. C'est ainsi une véritable plate-forme de développement et d'expérimentation d'ordonnanceurs à bulles qui a été intégrée au sein de la bibliothèque de threads utilisateur MARCEL. Le support OPENMP du compilateur GCC, GOMP, a été étendu pour utiliser cette bibliothèque et exprimer la nature structurée des sections parallèles imbriquées à l'aide de bulles. Avec la couche de compatibilité POSIX de MARCEL, ces supports ont permis de tester les différents ordonnanceurs à bulles développés, sur différentes applications. Les gains obtenus, de l'ordre de 20 à 40%, montrent l'intérêt de notre approche.
Mots-clés : Calcul intensif, parallélisme, supports d'exécution, threads, multiprocesseur, NUMA, multicore

Abstract: The current trend of constructors for scientific computation is towards an imbrication of technologies that permits an ever increased degree of parallelism within a single machine: NUMA architectures, multicore chips, SMT. The efficiency of the execution of an irregular parallel application on such hierarchical machines hence relies on the quality of thread scheduling and data placement, so as to avoid NUMA penalties and cache misses as much as possible. Current Operating Systems fail to achieve this because they are too generic, and thus application developers end up tuning their program for a given machine.
In this thesis, in order to guarantee some portability of performances, we define the notion of *bubble* that allows to express the parallel structure nature of the computation, and we model the architecture of the target machine by a hierarchy of runqueues. A high-level programming interface and several debugging tools then permit to easily develop dedicated, efficient, and portable schedulers. Several schedulers that implement various approaches have been developed, some of which by trainees, which shows both the powerfulness and the simplicity of the interface. To sum it up, this is a real platform for developing and experimenting with bubble schedulers which has been integrated in the user-level thread library MARCEL. The OPENMP support of the GCC compiler, GOMP, was extended to use that library and express the structured nature of nested parallel sections thanks to bubbles. Along with the POSIX compatibility layer of MARCEL, these ports permitted to test the bubble schedulers that were implemented on various applications. The achieved performance benefits, about 20 to 40%, show the interest of our approach.
Keywords: High-Performance Computing, parallelism, runtime systems, threads, multiprocessors, NUMA, multicore

Table des matières

Introduction **1**

1 Exploitation des architectures multiprocesseurs pour le H.P.C. **5**
1.1 Vers un raffinement des applications 6
1.1.1 Une structuration asymétrique 6
1.1.2 Un comportement irrégulier 7
1.1.3 Des besoins en ordonnancement particuliers 8
1.2 Vers une hiérarchisation des machines multiprocesseurs 8
1.2.1 Accès mémoire non uniformes 8
1.2.2 Puces multicœurs 10
1.2.3 Processeurs multithreadés 11
1.2.4 De véritables poupées russes 12
1.3 Du programmeur à l'environnement d'exécution 16
1.3.1 Une nécessité d'expliciter le parallélisme 16
1.3.2 Des programmeurs non spécialistes en parallélisme 16
1.4 De l'environnement d'exécution à l'ordonnancement et au placement 17
1.4.1 Approches opportunistes 18
1.4.2 Approches pré-calculées 20
1.4.3 Approches négociées 22
1.4.4 Paramètres et indications utiles à l'ordonnancement 22
1.5 Discussion 24

2 Structurer pour mieux ordonnancer **27**
2.1 Exprimer la structure parallèle d'une application 28
2.1.1 Structurer le parallélisme à l'aide de bulles 28
2.1.2 Qualifier le parallélisme à l'aide de bulles 30
2.1.3 Attacher des informations aux bulles 31
2.1.4 Travaux apparentés 32
2.2 Coller à la structure de la puissance de calcul 33
2.3 Combiner les modélisations 34

3 BubbleSched : une plate-forme pour l'élaboration d'ordonnanceurs portables **37**
3.1 Une interface pour programmer 38
3.1.1 Côté applicatif, construire une information hiérarchisée 38
3.1.2 Côté ordonnancement, gérer les hiérachies de listes et de bulles 38
3.1.3 Calculer et maintenir automatiquement des statistiques liées aux bulles 42

3.2 Quelques exemples pour s'inspirer . . . 44
3.2.1 *Burst*, un ordonnancement par niveau . . . 44
3.2.2 *Gang*, un ordonnancement par tourniquet . . . 46
3.2.3 *Spread*, un ordonnancement par équilibrage de charge . . . 48
3.2.4 *Steal*, un ordonnancement par vol de travail . . . 51
3.2.5 *Affinity*, un ordonnancement par affinités . . . 54
3.2.6 *MemAware*, un ordonnancement dirigé par la mémoire . . . 55
3.2.7 Discussion . . . 57
3.3 Un environnement pour expérimenter . . . 58
3.3.1 Combiner des ordonnanceurs . . . 58
3.3.2 Comprendre les performances . . . 58
3.3.3 Multiplier les tests . . . 60

4 Éléments d'implémentation **63**
4.1 Sur les épaules de MARCEL . . . 63
4.2 Ordonnancement . . . 66
4.2.1 Ordonnanceur de base . . . 66
4.2.2 Mémoriser le placement . . . 67
4.2.3 Verrouillage . . . 68
4.3 Portabilité . . . 68
4.3.1 Systèmes d'exploitation : découverte de topologie . . . 68
4.3.2 Architectures : opérations de synchronisation . . . 71
4.4 Optimisations . . . 72
4.4.1 Parcours des bulles à la recherche du prochain thread à exécuter . . . 73
4.4.2 Plus léger que les processus légers . . . 73
4.4.3 Distribution des allocations . . . 75

5 Diffusion **77**
5.1 Compatibilité POSIX et construction automatisée de hiérarchies de bulles . . . 77
5.2 Support OPENMP : FORESTGOMP . . . 79
5.2.1 Parallélisme imbriqué en OPENMP . . . 80
5.2.2 Vers l'utilisation de MARCEL . . . 81
5.2.3 Intégration au sein de GCC : FORESTGOMP . . . 81
5.2.4 Discussion . . . 83

6 Validation **87**
6.1 Tests synthétiques . . . 87
6.1.1 Opérations de base . . . 87
6.1.2 Stress-test . . . 90
6.2 Applications . . . 90
6.2.1 ADVECTION / CONDUCTION, un parallélisme régulier . . . 92
6.2.2 SUPERLU, un parallélisme de tâches . . . 93
6.2.3 NPB BT-MZ, un parallélisme imbriqué irrégulier dans l'espace . . . 94
6.2.4 MPU, un parallélisme imbriqué irrégulier dans le temps . . . 97

7 Conclusion et perspectives **105**
7.1 Contributions . . . 106

7.2 Perspectives 107
7.2.1 Toujours plus d'informations 107
7.2.2 Des ordonnanceurs variés 107
7.2.3 Au-delà du modèle hiérarchique de processeurs 108
7.2.4 Diffusion du concept 110

A Interface de programmation des bulles **111**
A.1 Interface de programmation utilisateur des bulles 112
A.2 Interface de programmation d'ordonnancement des bulles 114
A.3 Interface d'accès aux statistiques 115
A.4 Conteneurs et entités 117

Bibliographie **119**

Liste des publications **127**

Table des figures

1.1 Exemple d'évolution de maillage A.M.R. 7
1.2 Architecture SMP. 9
1.3 Architecture NUMA. 9
1.4 Puce multicœur. 10
1.5 Processeur multithreadé 2 voies. 12
1.6 Architecture très hiérarchisée. 13
1.7 Hagrid, octo-bicœur Opteron. 14
1.8 Aragog, bi-quadricœur E5345 Xeon. 15
1.9 Ordonnancement opportuniste dans un système d'exploitation. 19
1.10 Ordonnancement pré-calculé, court-circuitant le système d'exploitation. . . . 21
1.11 Ordonnancement négocié, utilisant des informations de l'application et s'adaptant à la machine sous-jacente. 22

2.1 Expression des relations entre threads. 29
2.2 Synthèse de l'estimation d'utilisation du temps processeur. 31
2.3 Modélisation d'une machine très hiérarchisée par une hiérarchie de listes. . . 33
2.4 Distributions possibles des threads et bulles sur la machine. 36

3.1 Exemple pratique de construction imbriquée de bulles. 39
3.2 Différents types de parcours des listes. 41
3.3 Exemple de définition et d'utilisation d'une statistique. 43
3.4 Code source de l'ordonnanceur *Burst*. 45
3.5 Exécution de l'ordonnanceur *Burst*. 47
3.6 Code source du *Gang Scheduler*. 49
3.7 Statistiques d'exécution fournies par notre outil `top`. 50
3.8 Exécution de deux *gang schedulers* en parallèle. 50
3.9 Pseudo-code source de l'ordonnanceur *Spread*. 52
3.10 Déroulement de l'ordonnancement *Spread*. 53
3.11 Déroulement de l'ordonnancement *Affinity*. 56
3.12 *Action Replay* de l'ordonnancement *Spread*. 59
3.13 Interface graphique de génération de bulles : *BubbleGum*. 61

4.1 Mode de fonctionnement de base de Marcel. 64
4.2 Mode de fonctionnement de Marcel avec bulles. 65
4.3 Distribution de bulles et threads avec priorités. 67
4.4 Exemple de machine contenant deux nœuds NUMA comportant chacun un cœur hyperthreadé. 69

4.5 Déroulement de l'analyse de topologie. 70
4.6 Implémentations des opérations atomiques et verrous rotatifs. 72
4.7 Illustration du cache de threads. 73
4.8 Temps de création et de destruction parallèle de 100 000 threads. 76

5.1 Modèle hybride de gestion des signaux. 78
5.2 Construction automatique de bulles à partir des relations de filiation POSIX. 79
5.3 Construction automatique de bulles à partir des relations de filiation OPENMP. 82
5.4 Code de construction des bulles adaptées à OPENMP. 83

6.1 Temps de création de threads et de graines de threads. 88
6.2 Temps d'ordonnancement et de changement de contexte. 89
6.3 Coût de migration d'un thread. 89
6.4 Performances du programme *sumtime*. 91
6.5 Performances de la bibliothèque SUPERLU. 95
6.6 Parallélisme externe et parallélisme interne de l'application BT-MZ. 96
6.7 Parallélisme imbriqué. 98
6.8 Principe de la partition de l'unité multi-niveau (MPU). 99
6.9 Boucle principale de l'application MPU. 100
6.10 Principe de fonctionnement d'un *deque* du protocole THE simplifié. 101
6.11 Accélération de l'application MPU selon l'environnement d'exécution utilisé. 102
6.12 Construction d'un ensemble de polygones à l'aide de l'algorithme *Marching-Cube*. 103

Liste des tableaux

1.1 Temps d'exécution d'un solveur PDE. 10
1.2 Confrontation de deux types de calculs sur processeur hyperthreadé. 12

4.1 Efficacité du programme *sumtime* selon les différents efforts de portage. . . . 72
4.2 Coût de création et d'exécution d'un thread et d'une graine de thread. 74

6.1 Performances des applications ADVECTION et CONDUCTION selon les approches. 93

Introduction

Le calcul hautes performances

La simulation numérique est maintenant reconnue comme l'un des piliers de la démarche scientifique. En effet, que ce soit en climatologie, en astrophysique, en nanosciences, en chimie des matériaux, en biologie moléculaire, etc. elle est devenue un outil complémentaire à la théorie et à l'expérimentation, permettant par exemple de présélectionner les expériences à réaliser voire de les remplacer. Disposer de grands moyens de calcul est même devenu une des vitrines de certains instituts de recherche. Les besoins en terme de puissance de calcul sont ainsi en constante progression quels que soient les domaines, nécessitant alors des évolutions techniques pour pouvoir subvenir à une telle demande. Une réponse durant les années 70 avait été de développer des super-calculateurs, qui ont connu leur âge d'or dans les années 80. Les années 90 ont cependant montré que des grappes de simples PCs se révèlent bien moins coûteuses et plus évolutives. Le top 500 [top] des machines de calcul les plus puissantes du monde a ainsi progressivement basculé vers une prédominance de telles machines, à l'origine pensées pourtant pour une utilisation personnelle (*Personal Computer*).

Dans le contexte du marché de l'ordinateur personnel, les besoins en terme de performances sont tout autant présents pour offrir toujours plus de fonctionnalités ou effets visuels réalistes, par exemple. La réponse a alors été pendant longtemps de rendre les processeurs toujours plus rapides, que ce soit par l'augmentation de la fréquence ou de la complexité du processeur, allant même jusqu'à y introduire un parallélisme implicite ! Les applications pouvaient alors profiter de ces améliorations sans aucune modification. Cependant, les fréquences utilisées de nos jours sont telles que ne serait-ce que la dissipation thermique pose d'énormes problèmes ; l'abandon par INTEL de l'architecture qui avait permis de construire le PENTIUM 4 montre par ailleurs qu'une certaine limite de complexité des processeurs a été atteinte. La *course à la fréquence* a ainsi fait place à ce que beaucoup appellent l'*ère multi-cœur* : plutôt que de dépenser des transistors pour construire des processeurs de plus en plus difficiles à mettre au point, il vaut mieux désormais graver plusieurs processeurs (appelés cœurs) sur une même puce. Reste alors aux éditeurs de logiciels grand public à s'orienter vers la programmation parallèle...

Cette tendance du domaine de la grande distribution vers des machines multiprocesseurs n'est, en fait, que la partie émergée d'une tendance profonde des machines de calcul parallèle vers des architectures complexes embarquant de nombreux processeurs, telles que les machines SUN WILDFIRE, IBM P550Q, BULL NOVASCALE, ou les SGI ORIGIN et ALTIX,

certains prototypes de ces dernières pouvant embarquer 4096 processeurs tout en assurant une cohérence de cache sur toute la machine ! Bien sûr, de telles machines ne sont pas de simples alignements de processeurs, mais de véritables poupées russes, combinant de manière hiérarchique des réseaux d'interconnexion de processeurs (introduisant des facteurs NUMA variés), des puces multicœurs avec certains niveaux de cache partagés, ou encore la technologie SMT. Au final, un scientifique qui veut programmer une telle machine est plutôt perplexe : comment prendre en compte la complexité de ces machines pour pouvoir en tirer la quintessence ? Cette question est d'autant plus ardue que les applications elles-mêmes se sont complexifiées, utilisant désormais des techniques telles que le raffinement adaptatif de maillage (AMR) qui introduisent des irrégularités de charge de calcul qu'il n'est pas trivial de projeter sur de telles machines. Hennessy et Paterson relèvent [HP03] à propos des systèmes proposés pour SGI ORIGIN et SUN WILDFIRE : *« There is a long history of software lagging behind on massively parallel processors, possibly because the software problems are much harder. »*

Il apparaît ainsi un élément qui se révèle de plus en plus essentiel au sein de la chaîne de fonctionnement du calcul hautes performances : l'environnement d'exécution et son ordonnanceur. Avec la complexité des applications irrégulières, le compilateur ne peut en effet pas prévoir *a priori* un ordonnancement, même s'il dispose d'informations sur la nature du parallélisme. Le système d'exploitation, qui offre une interface assez pauvre pour traverser la barrière entre espace utilisateur et espace noyau, n'a, à l'inverse, quasiment aucune information sur l'application en cours d'exécution. Il est ainsi naturel d'embarquer un environnement d'exécution en espace utilisateur, qui lui, pourra profiter d'informations données par le programmeur ou le compilateur pour effectuer un ordonnancement approprié. C'est là que s'inscrivent les développements de cette thèse.

Objectifs de la thèse et contributions

Il s'agit donc d'abord d'étudier les moyens dont disposent les programmeurs scientifiques pour exprimer le parallélisme de leurs applications, et comment les environnements d'exécution existants exécutent celles-ci sur ces machines complexes. Nous verrons alors un manque général de prise en compte, par ces environnements d'exécution, d'informations pourtant essentielles pour obtenir une exécution efficace, notamment les relations d'affinités entre tâches. De fait, il est devenu courant que les programmeurs construisent à la main des ordonnancements, bien souvent pour une machine donnée seulement, alors que ce devrait plutôt être à l'*environnement d'exécution* d'effectuer automatiquement un ordonnancement adapté à la machine cible grâce à des informations sur l'application. Loin d'aller jusqu'à exiger une description complète permettant un ordonnancement *a priori*, ou de ré-implémenter pour chacune un ordonnanceur complet, nous proposons un concept au départ simple, la notion de *bulle*. Cet objet permet de regrouper des tâches par affinités et aboutit finalement à une structuration riche, par imbrication de ces bulles et ajout d'informations utiles telles que la charge de calcul estimée. Il est alors possible pour des experts en ordonnancement de développer aisément et de manière *portable* des ordonnanceurs qui peuvent s'adapter automatiquement à différentes machines et être réutilisés pour différentes applications. Les axes significatifs qui ont été suivis sont donc :

- Une programmation aisée pour *tout scientifique* : il n'est pas question d'exiger d'un programmeur qu'il fournisse une grande quantité de détails sur le comportement de son programme. La construction des bulles et leur enrichissement doivent pouvoir être effectués de manière plus ou moins automatisée selon l'environnement de programmation utilisé.
- Une interface de programmation d'ordonnanceurs pour *spécialistes en ordonnancement* : pour pouvoir se concentrer sur l'algorithmie de l'ordonnancement plutôt que les détails techniques d'implémentation, il est utile de disposer d'une interface de haut niveau pour implémenter des ordonnanceurs à bulles.
- Une *efficacité* finale satisfaisante : puisqu'il est ici question d'obtenir de bonnes performances d'exécution, il est bien sûr essentiel que les concepts mis en jeu soient implémentés d'une manière efficace pour limiter les surcoûts résultants.
- Une intégration en une *plate-forme complète* : pour qu'elle puisse être utilisable, une plate-forme se doit de disposer d'outils adéquats tels qu'une aide au débogage.

Les contributions majeures de cette thèse sont ainsi d'une part la proposition de la notion originale de *bulle* et d'interfaces de programmation pour les programmeurs d'applications et pour les spécialistes en ordonnancement, et d'autre part le développement d'une plate-forme complète de développement mettant en œuvre ces concepts. Ceci comprend l'intégration proprement dite et autres développements utiles au sein de la bibliothèque de threads utilisateur MARCEL, faisant passer son nombre de lignes de code d'environ 30 000 à 71 000, auquel s'ajoute des modules de débogage graphique totalisant un peu plus de 15 000 lignes de code, et l'intégration au sein du compilateur OPENMP de GCC, GOMP, qui a nécessité quelques centaines de lignes. D'autre part, deux stagiaires encadrés au sein de l'équipe ont produit des ordonnanceurs à bulles pour un total d'environ 3 000 lignes. Ces travaux ont fait l'objet de plusieurs publications citées à la fin de ce document et de nombreux séminaires.

Organisation du document

Le chapitre 1 nous permet de présenter plus en détails le contexte dans lequel s'inscrit cette thèse. Nous y présentons les évolutions du calcul hautes performances, à la fois d'un point de vue applicatif, d'un point de vue architectural, et de l'interaction entre programmeurs, environnement d'exécution et architectures parallèles. Le chapitre 2 introduira alors de manière théorique le concept de *bulle* et d'ordonnancement à bulles basé sur les informations qui y sont attachées. Le chapitre 3 présente ensuite, de manière plus concrète, la plate-forme développée au cours de cette thèse : les interfaces fournies aux programmeurs d'application et aux programmeurs d'ordonnanceurs, ainsi que plusieurs exemples d'ordonnanceurs et quelques outils d'aide au développement. Le chapitre 4 fournit des détails sur l'implémentation de cette plate-forme : l'intégration au sein de MARCEL, la mise en œuvre efficace du concept de bulle, une simplification du portage sur des systèmes et processeurs variés, et quelques optimisations notables qu'il a été utile d'effectuer. Le chapitre 5 explique comment cette plate-forme a pu s'intégrer au sein des environnements de programmation parallèle classiques existants : l'interface de threads POSIX et le standard OPENMP. Le chapitre 6 permet de valider les développements effectués, à la fois sur des benchmarks clés et sur des applications concrètes. Nous pourrons alors tirer les conclusions de nos travaux et discuter des principales perspectives de recherches qui en résultent.

Chapitre 1

Exploitation des architectures multiprocesseurs pour le Calcul Hautes Performances

Sommaire

1.1 Vers un raffinement des applications 6
1.1.1 Une structuration asymétrique 6
1.1.2 Un comportement irrégulier 7
1.1.3 Des besoins en ordonnancement particuliers 8
1.2 Vers une hiérarchisation des machines multiprocesseurs 8
1.2.1 Accès mémoire non uniformes 8
1.2.2 Puces multicœurs 10
1.2.3 Processeurs multithreadés 11
1.2.4 De véritables poupées russes 12
1.3 Du programmeur à l'environnement d'exécution 16
1.3.1 Une nécessité d'expliciter le parallélisme 16
1.3.2 Des programmeurs non spécialistes en parallélisme 16
1.4 De l'environnement d'exécution à l'ordonnancement et au placement . . 17
1.4.1 Approches opportunistes 18
1.4.2 Approches pré-calculées 20
1.4.3 Approches négociées 22
1.4.4 Paramètres et indications utiles à l'ordonnancement 22
1.5 Discussion 24

Dans ce premier chapitre, nous exposons le contexte dans lequel se situe notre travail, ainsi que l'état de l'art à la fois du côté applicatif, matériel et programmation. Les évolutions du Calcul Hautes Performances se produisent en effet à tous les niveaux. Les scientifiques sont bien sûr toujours plus demandeurs de ressources de calcul pour leurs applications, mais celles-ci exhibent par ailleurs de plus en plus souvent un comportement irrégulier, non prévisible. D'autre part, la tendance architecturale des ordinateurs contemporains, toutes catégories confondues, est d'augmenter significativement le niveau de parallélisme des unités de calcul. Le problème qui se pose est alors de parvenir à exécuter ces applications le plus

efficacement possible sur ces machines toujours plus complexes, ce qui mène assez naturellement à la notion d'*environnement d'exécution*, qui se charge d'apporter des fonctionnalités de haut niveau aux programmeurs d'applications, en masquant certains aspects des architectures sous-jacentes.

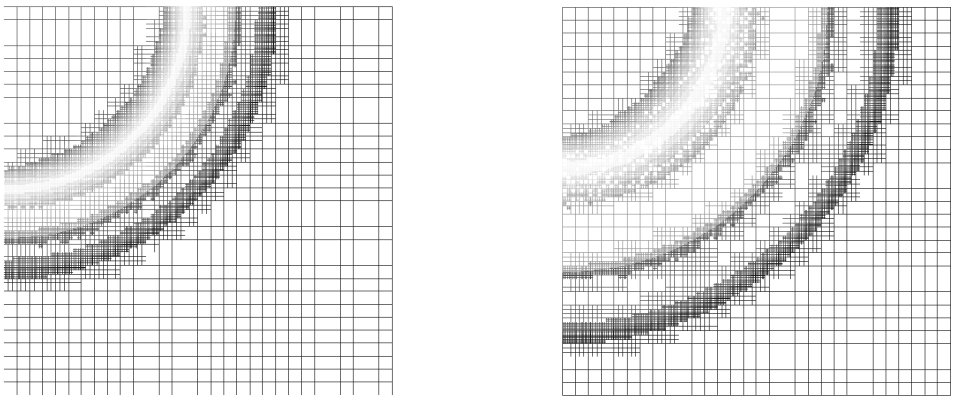
Ce chapitre est donc organisé en cinq parties. Nous voyons dans un premier temps les caractéristiques particulières que peuvent exhiber les applications de calcul scientifique, notamment leur comportement irrégulier. Ensuite, nous faisons un tour d'horizon des technologies de parallélisme intégrées dans les processeurs actuels (y compris ceux du grand public !), technologies qui les rendent si complexes à utiliser : accès mémoire non uniforme, multicœur et multithreading, ainsi que leurs combinaisons. Nous nous intéressons alors aux environnements d'exécution, par deux approches. Nous étudions d'abord leurs liens avec les programmeurs d'applications, par le biais des langages de programmation principalement. Nous explorons ensuite les principales approches existantes pour exécuter effectivement les applications sur les machines actuelles. Nous discutons enfin des atouts et inconvénients des différentes solutions existantes, ce qui mène aux contributions de cette thèse, exposées dans les chapitres suivants.

1.1 Vers un raffinement des applications

Quel que soit le domaine (physique, chimie, biologie, etc.), les scientifiques sont toujours demandeurs d'une précision de calcul accrue et d'un temps d'exécution réduit. Cela induit une tendance vers une structuration asymétrique irrégulière, ainsi que des besoins en ordonnancement particuliers.

1.1.1 Une structuration asymétrique

Pour augmenter toujours plus la précision de calcul sans pour autant que l'occupation mémoire et le temps de calcul explosent, les applications ont de plus en plus tendance à utiliser des structures creuses ou localement raffinées. En effet, avec par exemple une approche basique qui est d'utiliser un simple maillage 3D, l'occupation mémoire et le temps de calcul sont *a priori* directement proportionnels à l'occupation du maillage, qui elle est proportionnelle au cube du côté du maillage. Ne serait-ce que doubler la finesse d'une simulation dans les trois dimensions multiplie ainsi par 8 l'occupation mémoire. Le temps de calcul est *a priori* aussi multiplié par 8, mais peut éventuellement augmenter bien plus si la taille du cache des processeurs devient alors insuffisante. Pour éviter une telle explosion, les applications de calcul scientifique exploitent de plus en plus souvent les caractéristiques du calcul pour alléger leurs structures de données. Il arrive par exemple que de grands blocs d'une matrice soient nuls, auquel cas la matrice est stockée par blocs, et les blocs nuls ne sont pas mémorisés. On économise ainsi à la fois l'occupation mémoire et les calculs concernant ces blocs (puisqu'ils deviennent alors le plus souvent triviaux). Dans le cas d'une simulation physique à l'aide d'un maillage, il arrive souvent que seules certaines portions de maillage ont réellement besoin d'une bonne précision de calcul, le reste ayant un comportement homogène et régulier. On utilise alors un maillage raffiné localement aux endroits intéressants : sur la figure 1.1(a) par exemple, le maillage n'est raffiné qu'au niveau de l'onde de choc, le reste du domaine



(a) Cycle 171. (b) Cycle 337.

FIG. 1.1 – Exemple d'évolution de maillage A.M.R.

est simulé par un maillage grossier. Cependant, il arrive que ces portions changent au cours du temps, lorsqu'une onde de choc se propage par exemple. Il est donc courant d'utiliser un *maillage adaptatif* (ou A.M.R.), c'est-à-dire de faire évoluer le raffinement du maillage au cours de la simulation, pour « suivre » le phénomène physique simulé. On voit ainsi sur la figure 1.1(b) que le maillage a été raffiné à la nouvelle position de l'onde de choc, tandis qu'il a été regrossi à son ancienne position. La précision de calcul au niveau du phénomène intéressant peut ainsi être très grande sans que le coût explose : l'occupation mémoire et le temps de calcul sont dépensés essentiellement aux endroits utiles.

Par ailleurs, il est courant d'utiliser un *couplage de codes* : lorsqu'une simulation met en jeu des éléments de natures très différentes (liquide / solide par exemple), il est préférable d'utiliser pour chaque élément un code de simulation qui y est adapté, et de coupler les codes entre eux au niveau des interfaces entre éléments. On se retrouve alors avec plusieurs codes de natures éventuellement très différentes, à faire exécuter de concert sur une même machine.

1.1.2 Un comportement irrégulier

La conséquence de tels raffinements est que le comportement des applications de calcul scientifique devient irrégulier. Il l'est d'abord au sens où la charge de calcul et d'occupation mémoire n'est pas homogène (c'est d'ailleurs l'objectif visé !). On ne peut donc pas se contenter d'utiliser des solutions de répartition de travail triviales. Le comportement est de plus irrégulier au sens où il évolue au cours du temps, selon un schéma qui n'est souvent *pas prévisible a priori* : il dépend des résultats intermédiaires obtenus lors de l'exécution. Dans le cas de calculs matriciels creux, il peut être possible de prévoir l'évolution des blocs nuls, mais

dans le cas d'un maillage A.M.R., le raffinement effectué est très vite difficilement prévisible. Les numériciens parviennent dans une certaine mesure à prévoir dynamiquement comment le raffinement évolue, mais cette prévision ne peut évidemment pas être parfaite (cela voudrait dire que l'on connaît déjà le resultat de la simulation !). Les solutions de répartition de travail doivent donc en plus être *dynamiques*.

1.1.3 Des besoins en ordonnancement particuliers

Enfin, les besoins en ordonnancement de telles applications sont souvent particuliers. En effet, elles utilisent souvent pour leurs calculs des routines BLAS. De telles routines sont « calées » pour exploiter au maximum un processeur et son cache. Il est donc préférable de ne pas interrompre de telles routines pendant leur exécution, pour que la réutilisation du cache soit maximale. Or en général, pour préserver l'*équité* entre les tâches, les systèmes d'exploitation *préemptent* régulièrement la tâche en cours pour en laisser exécuter une autre, afin que toutes les tâches progressent globalement de manière équitable. Dans le cas d'une application de calcul scientifique, c'est inutile : l'objectif est surtout de donner le résultat le plus vite possible. C'est pourquoi les numériciens ont tendance à faire désactiver les mécanismes de préemption. Par contre, il est courant de décomposer les calculs pour réduire la taille des données manipulées par chaque opération à la taille du cache. Les nombreuses tâches résultant de cette décomposition peuvent alors très bien être exécutées (chacune entièrement) dans un ordre quelconque et en parallèle, tant que les dépendances de données sont respectées.

Ainsi donc les applications de calcul scientifique ont des besoins particuliers. Elles doivent pouvoir exprimer la finesse asymétrique de leur structure (pour permettre une distribution de charge fine), et garder un certain contrôle sur l'ordonnancement effectué (pouvoir désactiver toute préemption par exemple).

1.2 Vers une hiérarchisation des machines multiprocesseurs

Le grand public a récemment découvert de nouveaux termes tels que *HyperThreading* et *multicore*, et il lui devient de plus en plus difficile d'acheter une machine qui ne soit pas parallèle. Ceci n'est que la partie émergée d'une tendance profonde du calcul scientifique vers des machines toujours complexes, où le parallélisme s'instaure à tous les niveaux. Dans cette section, nous présentons les caractéristiques des technologies principales de parallélisme actuelles, dont la combinaison produit des machines très hiérarchisées.

1.2.1 Accès mémoire non uniformes

La forme la plus simple de machine parallèle est l'architecture *Symmetric MultiProcessing* (**SMP**), telle que représentée à la figure 1.2 : plusieurs processeurs sont connectés à une mémoire commune via un bus ou un commutateur parfait (*crossbar*). Cependant, avec un nombre croissant de processeurs, une telle implantation devient rapidement problématique :

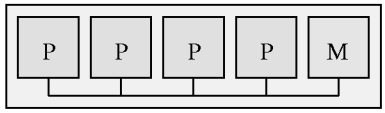


FIG. 1.2 – Architecture SMP.

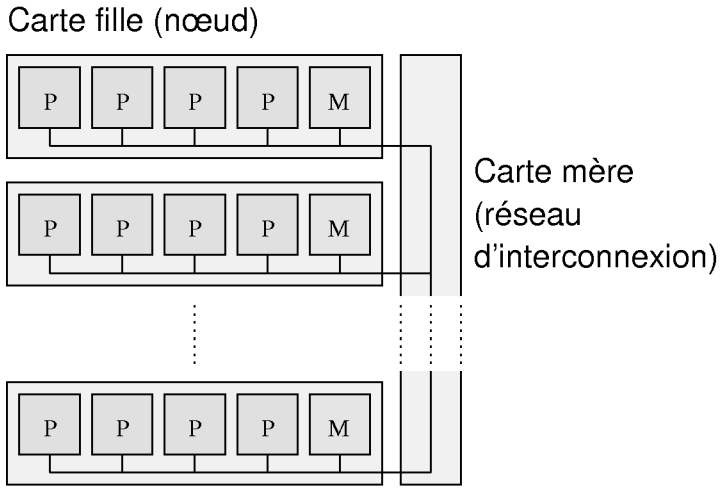


FIG. 1.3 – Architecture NUMA.

un bus est limité par sa bande passante, et le coût d'un commutateur devient très vite prohibitif. Une solution courante est alors d'utiliser une architecture à accès mémoire non uniforme (*Non-Uniform Memory Access*, **NUMA**), comme représentée sur la figure 1.3. La mémoire y est distribuée sur différents **nœuds**, qui sont reliés par un réseau d'interconnexion pour que tous les processeurs puissent tout de même accéder de manière transparente à toute la mémoire. Typiquement, les nœuds sont des cartes filles enfichées sur une carte mère. Dans une telle machine, les accès mémoire dépendent alors de la position relative du processeur qui effectue l'accès et de l'emplacement mémoire accédé. La latence d'accès à une donnée *distante* peut être par exemple 2 fois plus grande que celle pour une donnée *locale*, car l'accès doit transiter par le réseau d'interconnexion. C'est le **facteur NUMA**, qui dépend fortement de la machine et peut typiquement varier de $1,1$ à 10 ! En pratique, il est généralement de l'ordre de 2, et est légèrement plus grand pour une lecture que pour une écriture.

Ce type d'architecture était traditionnellement plutôt réservé aux super-calculateurs, or comme le montre le classement Top500 [top], ceux-ci tendent depuis quelque temps à laisser la place aux grappes de simples P.C., si bien que l'architecture NUMA était quelque peu tombée dans l'oubli. Cependant, dans ses puces OPTERON le fondeur A.M.D. intègre désormais un contrôleur mémoire directement dans le processeur pour pouvoir y connecter directement la mémoire et ainsi augmenter fortement la bande passante disponible. Cela induit en fait une architecture NUMA, puisque la mémoire se retrouve alors répartie sur les différents processeurs. Ces puces étant devenues à la mode dans le grand public, l'architecture NUMA s'est en fait fortement démocratisée, et on la retrouve donc naturellement dans les grappes de P.C.

Le problème posé par ces machines est bien illustré par Henrik Löf *et al.* au sein de leurs

	Séquentiel	Parallèle	*Speedup*
SUN ENTERPRISE 10 000	220 s	8 s	27,5
SUN FIRE 15 000	45 s	9,5 s	4,7

TAB. 1.1 – Temps d'exécution d'un solveur PDE.

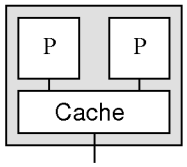


FIG. 1.4 – Puce multicœur.

travaux sur un solveur PDE [LH05]. Ils disposaient d'une machine SMP SUN ENTERPRISE 10 000 composée de 32 processeurs cadencés à 400MHz avec une latence d'accès mémoire de 550ns, et ont acquis une machine NUMA SUN FIRE 15 000 composée de 32 processeurs également, cadencés à 900MHz avec une latence d'accès mémoire local de 200ns et une latence d'accès mémoire distant de 400ns (soit un facteur NUMA égal à 2). Comme le montre la première colonne du tableau 1.1, l'augmentation de la fréquence des processeurs et la diminution du temps de latence d'accès mémoire permet au temps d'exécution séquentiel d'être divisé par presque 5. Cependant, le temps d'exécution parallèle est plus que décevant : la nouvelle machine se révèle *« plus lente »* que l'ancienne ! La raison à cela est que leur solveur initialise toutes les données séquentiellement sur le premier processeur, et le système SOLARIS les alloue alors dans la mémoire du nœud correspondant, le premier. Lorsque l'exécution parallèle commence réellement, tous les processeurs tentent alors d'accéder à la mémoire du premier nœud, qui se retrouve engorgée. Il apparaît donc essentiel, sur de telles machines, de bien contrôler l'emplacement effectif des allocations mémoire par rapport aux processeurs.

1.2.2 Puces multicœurs

Avec une finesse de gravure toujours plus poussée, les fondeurs disposent de plus en plus de place sur les puces. Pendant longtemps, cela a permis de fabriquer des processeurs de plus en plus sophistiqués avec des opérations flottantes avancées, une prédiction de branchement, etc. De nos jours cependant, la sophistication est devenue telle qu'il est devenu plus simple, pour gagner toujours plus en performances, de graver plusieurs processeurs sur une même puce : ce sont les puces multicœurs, telle qu'illustrée figure 1.4. Tous les grands fondeurs proposent désormais des déclinaisons bicœurs, quadricœurs, voire octocœurs de leurs processeurs, tels le NIAGARA 2 de SUN, ou le CELL d'IBM popularisé par la PLAYSTATION 3 de SONY. Les puces avec 16 cœurs sont déjà sur les agendas, et Intel prévoit même de fabriquer un jour des puces embarquant des centaines de cœurs [Rat] !

Une caractéristique intéressante de ces puces est que, puisque les processeurs sont gravés sur une même puce, il est courant de leur donner un accès en commun à un seul et même cache. En effet, lorsque les deux processeurs exécutent une même application ou une même bibliothèque par exemple, il serait dommage de garder une copie du programme au sein de chacun des processeurs. Par ailleurs, si les deux processeurs ont besoin de communiquer

entre eux, ils peuvent le faire très rapidement par mémoire partagée *via* ce cache. Au-delà de deux cœurs, il est fréquent que plusieurs niveaux de caches soient partagés de manière hiérarchique, tel que le conseillent Hsu *et al.* [HIM+05].

Cependant, la bande passante d'accès vers l'extérieur de la puce est elle aussi partagée. On le constate effectivement expérimentalement : nous avons mesuré sur une puce bicœur Opteron que si un seul processeur écrit en mémoire, il dispose d'une bande passante d'environ $4,4\,\mathrm{Go/s}$, alors que lorsque les deux processeurs écrivent en même temps, ils disposent chacun d'environ $2,1\,\mathrm{Go/s}$.

On est donc déjà ici face à un dilemme : selon les applications, il vaudra mieux placer des tâches sur une même puce multicœur pour qu'elles bénéficient du partage du cache, ou bien au contraire les répartir sur différentes puces pour que chaque tâche puisse bénéficier de la bande passante mémoire maximale et qu'elles évitent d'empiéter l'une sur l'autre dans le cache. Il faudra donc que l'ordonnanceur puisse prendre en compte les contraintes applicatives pour effectuer des choix appropriés.

1.2.3 Processeurs multithreadés

Le parallélisme est en fait au cœur même des processeurs grand public depuis longtemps. Les premiers processeurs Pentium par exemple disposaient déjà de plusieurs unités de calcul pour exécuter en parallèle deux instructions successives d'un programme séquentiel, quand les contraintes de données le permettent. Ce niveau de parallélisme n'est cependant pas vraiment « visible » pour le programmeur : le programme fourni reste séquentiel. Les compilateurs s'efforcent en général, dans le code généré, de disperser les instructions dépendant les unes des autres pour permettre une exécution parallèle, mais cela reste limité. Des instructions spéciales (du type MMX ou SSE du côté des Pentiums) permettent de fournir explicitement une série d'instructions flottantes à effectuer, que le processeur peut traiter facilement en parallèle. Cependant, cela nécessite soit d'écrire le code explicitement en assembleur[1], soit d'inclure un module de vectorisation dans le compilateur. Un tel module n'est cependant bien souvent pas à même de parvenir à extraire de grandes portions de code de calcul vectoriel. Par ailleurs, les unités de calcul sont de plus en plus fortement pipelinées. Lorsqu'une donnée n'est pas disponible immédiatement dans le cache, ces longs pipelines deviennent progressivement inactifs, en attente de l'accès mémoire (ce qu'on appelle *bulle*). Pour mieux profiter de ce parallélisme et tenter de combler les bulles dans les pipelines, certains processeurs contiennent de quoi exécuter plusieurs programmes en parallèle, c'est le *Simultaneous MultiThreading* (**SMT**, aussi appelé *HyperThreading* par INTEL [MBH+02], représenté figure 1.5). Les différents programmes (en général un nombre fixe pour un processeur donné) progressent selon l'occupation courante du processeur. Les détails d'ordonnancement varient d'un processeur à un autre, mais le processeur peut par exemple à chaque cycle prendre une instruction d'un ou plusieurs programmes (tour à tour) selon les disponibilités des unités de calcul et du type de l'instruction en cours des programmes. Cela permet ainsi en général d'alimenter toutes les unités de calcul, et lorsqu'un programme est bloqué en attente d'une lecture mémoire, les autres peuvent progresser plus « souvent ».

[1] On peut citer le cas du codec Xvid qui exploite au maximum cette possibilité, si bien que la version parallèle, lancée sur un processeur hyperthreadé, fonctionne exactement à la même vitesse : la version séquentielle profite en fait *déjà* au maximum de toutes les unités de calcul.

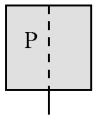


FIG. 1.5 – Processeur multithreadé 2 voies.

	Flottant	Entier	Inactif
Flottant	280 / 280	120 / 281	- / 282
Entier	281 / 120	110 / 110	- / 210

TAB. 1.2 – Confrontation de deux types de calculs sur processeur hyperthreadé, en itérations par seconde.

Le gain obtenu n'est cependant pas forcément évident. En effet, nous avons essayé de combiner des boucles de calcul entier et flottant très simples sur les puces hyperthreadées d'INTEL qui fournissent deux *processeurs virtuels*. Les résultats sont représentés dans le tableau 1.2, où l'on peut lire la vitesse d'exécution des deux boucles, pour chaque combinaison possible. On constate que la boucle de calcul flottant n'est que très peu ralentie lorsqu'on l'exécute en parallèle sur les deux processeurs virtuels. La boucle de calcul entier, par contre, voit sa vitesse presque divisée par deux. On peut en déduire que ce processeur possède au moins deux unités de calcul flottant, permettant vraiment d'exécuter nos deux boucles en parallèle, mais qu'il ne possède qu'une unité de calcul entier, que les deux processeurs virtuels doivent se partager. La combinaison Entier/Flottant est beaucoup moins évidente à interpréter : la boucle de calcul flottant semble fortement ralentir la boucle de calcul entier.

De manière générale, le comportement des combinaisons de programmes non triviaux est assez difficile à prévoir. Intel, pour sa part, annonce un gain de performances qui peut atteindre 30%. Bulpin et Pratt [BP04] ont essayé de combiner les différents programmes de la suite SPEC CPU2000, et obtiennent effectivement parfois un gain de 30%, corrélé avec un taux de défauts de cache élevé. À l'inverse, lorsque le taux de défauts de cache est faible, ils observent parfois une perte de performances ! C'est pourquoi bien souvent, pour le calcul scientifique, l'*HyperThreading* est désactivé...

1.2.4 De véritables poupées russes

Nous avons jusqu'ici étudié différentes technologies indépendamment les unes des autres. En pratique, elles sont très souvent combinées, ce qui produit des machines très hiérarchisées, telles que les machines SUN WILDFIRE [HK99], SGI ALTIX [WRRF03] et ORIGIN [LL97], BULL NOVASCALE [Bul] ou l'IBM P550Q [AFK+06]. La figure 1.6 montre un exemple d'une machine combinant une architecture NUMA avec des puces multicœurs dont chaque cœur est lui-même hyperthreadé. On peut remarquer que le réseau d'interconnexion est ici lui-même hiérarchisé. Sur d'autres machines cependant, il a parfois une topologie moins symétrique. La figure 1.7 montre par exemple l'architecture d'une des machines de test utilisées pour le chapitre 6 : les 8 nœuds NUMA y sont reliés d'une manière assez particulière. Le

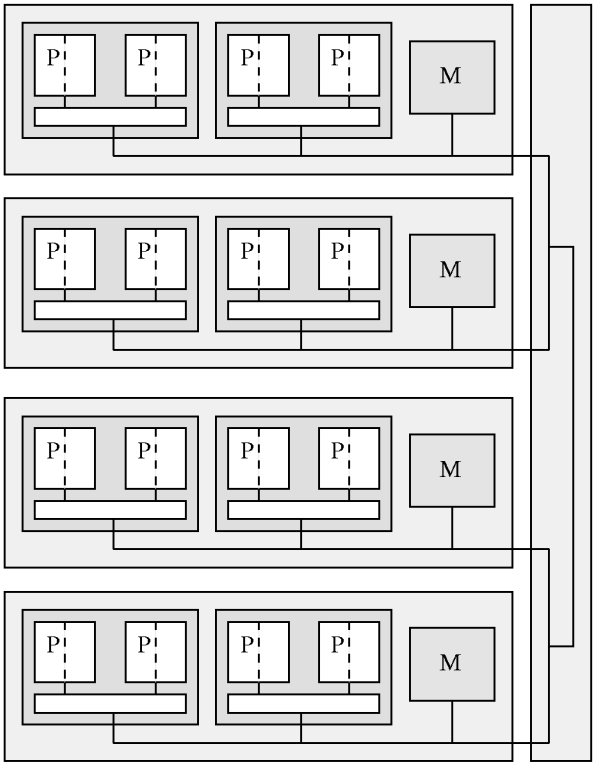


FIG. 1.6 – Architecture très hiérarchisée.

facteur NUMA varie ainsi beaucoup selon les positions respectives du processeur effectuant l'accès et l'emplacement mémoire accédé. En pratique, nous avons pu mesurer des facteurs variant d'environ $1,1$ pour des nœuds voisins à environ $1,4$ pour des nœuds extrêmes. Il faut noter que ce n'est pas forcément seulement ce facteur qui est pénalisant, mais aussi les bandes passantes qui sont limitées sur chaque lien.

Il devient ainsi bien difficile d'arriver à exploiter de telles machines. L'écart se creuse de plus en plus entre les performances de crête censées être disponibles et les performances réellement obtenues. Par exemple, la machine TERA10, acquise par le CEA pour obtenir une puissance de calcul de 10 TeraFlops pour ses simulations, est en réalité une machine « Tera65 » : sa puissance de crête est de 65 TeraFlops, que cependant aucun code de calcul réel n'est à même d'atteindre...

La variété de ces machines étant par ailleurs très grande, exploiter ces machines de manière *portable* est d'autant plus un défi. Certains constructeurs tentent de palier ces problèmes avec certains artefacts. Par exemple, il est souvent possible de configurer la carte mère pour fonctionner en mode *entrelacé*. En principe sur une machine NUMA, l'espace d'adressage mémoire physique est divisé en autant de fois qu'il y a de nœuds, le système d'exploitation pouvant ainsi allouer une page sur un nœud donné en choisissant simplement son adresse à l'intérieur de l'espace d'adressage correspondant au nœud. En mode entrelacé, les pages mémoire des différents nœuds sont distribuées dans l'espace d'adressage mémoire physique de manière cyclique : avec 2 nœuds par exemple, les pages de numéro pair proviennent du nœud 0 tandis que les pages de numéro impair proviennent du nœud 1. Ainsi, en moyenne

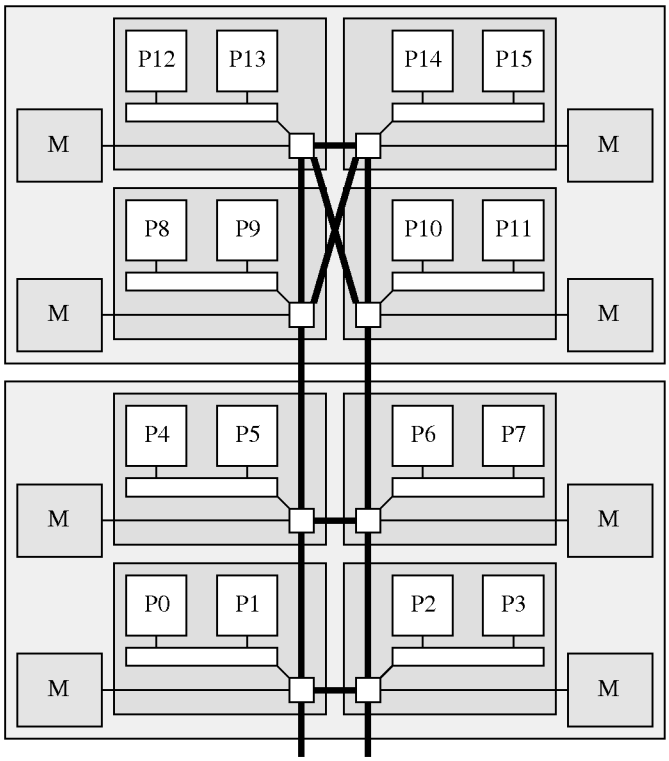


FIG. 1.7 – Hagrid, octo-bicœur Opteron.
Chaque nœud NUMA est composé d'une puce bicœur Opteron cadencée à $1,8$ GHz et de 8 Go de mémoire. Ces nœuds sont reliés par un réseau de liens HyperTransport. Le facteur NUMA varie selon la position relative de l'emplacement mémoire accédée et du processeur effectuant l'accès. En pratique, nous avons observé grossièrement trois valeurs typiques de facteurs NUMA pour cette machine : environ $1,1$, $1,25$ et $1,4$. Nous en avons alors déduit la topologie du réseau montrée ci-dessus, puisque les facteurs minimum correspondent à un seul saut dans le réseau. Cette topologie apparemment étrange s'expliquer par le fait que la machine est en fait physiquement composée de deux cartes mères superposées. En pratique cette partition en deux n'a pas d'impact sur les latences d'accès mémoire.

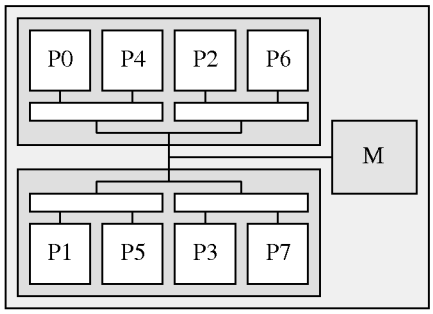


FIG. 1.8 – Aragog, bi-quadricœur E5345 Xeon.
Cette machine est composée de deux puces cadencées à 2, 33 GHz, chacune embarquant quatre cœurs. L'organisation hiérarchique des caches peut s'observer en mesurant la vitesse d'une boucle intensive de calcul sur des variables *volatiles* situées dans une même ligne de cache (en situation de faux partage, donc), par exemple une simple décrémentation. Lorsqu'un seul thread est lancé, il peut effectuer environ 330 millions d'itérations à la seconde. Lorsqu'un autre thread est lancé de telle sorte que le cache L2 est partagé, tous deux peuvent effectuer environ 280 millions d'itérations à la seconde, soit environ 1, 2 fois moins. Lorsqu'ils ne partagent pas le cache L2 mais sont placés sur la même puce, ils ne peuvent effectuer qu'environ 220 millions d'itérations à la seconde, soit 1, 5 fois moins. Lorsqu'ils sont placés sur des puces différentes, ils ne peuvent effectuer que seulement 63 millions d'itérations à la seconde, soit environ 5, 2 fois moins !

sur plusieurs pages, le temps d'accès paraît assez homogène, égal à la moyenne entre le temps d'accès local et le temps d'accès distant. C'est une solution simple pour éviter les cas pathologiques qui surviennent très souvent avec les programmes qui ne sont pas pensés pour machines NUMA ; c'est cependant du gâchis de performances, puisque l'on n'obtient qu'un temps d'accès moyen. De plus, un agent commercial peut alors faire croire à ses clients que « ce n'est pas une machine NUMA »... À titre anecdotique, signalons qu'à l'arrivée d'une commande de deux machines, nous avons constaté que l'entrelacement mémoire était activé sur une machine mais pas sur l'autre !

Un autre artefact courant repose sur la numérotation des processeurs. La figure 1.8 montre une machine dont la numérotation matérielle des processeurs est à première vue étrange. Lorsque l'on exécute une simulation occupant tous les processeurs et que les communications sont localisées entre tâches de numéros consécutifs, la stratégie usuelle (et d'habitude plutôt efficace), qui est d'attribuer les tâches dans l'ordre des processeurs, devient une des pires idées que l'on puisse imaginer ! Cette numérotation particulière est en fait prévue pour le cas où l'on dispose de moins de tâches à effectuer qu'il n'y a de processeurs, et que l'on attribue bêtement ces tâches dans l'ordre des processeurs. Les tâches disposent alors ici de la plus grande taille de cache pour elles seules, et du maximum de bande passante possible. Ce scénario se reproduit de la même façon dans le cas des processeurs hyperthreadés, et de même que pour l'entrelacement mémoire, si cette numérotation est souvent configurable, la configuration effectivement activée est parfois aléatoire...

1.3 Du programmeur à l'environnement d'exécution

Face à de telles machines, on peut comprendre que les scientifiques non informaticiens soient quelque peu perplexes. Comment peut-on programmer ces machines de manière portable ? Il apparaît donc nécessaire de disposer d'un *environnement d'exécution* qui masque tous ces détails techniques et permette aux scientifiques de se concentrer sur leurs propres problématiques. Nous présentons ici pourquoi il reste tout de même nécessaire d'exprimer d'une manière ou d'une autre le parallélisme, et dans quelle mesure il est possible de le faire sans être spécialiste en la matière.

1.3.1 Une nécessité d'expliciter le parallélisme

La parallélisation automatique permet dans les cas simples d'exécuter un programme sur une machine parallèle sans modifications (ou très peu). Ainsi, HPF (*High-Performance Fortran* [KLS+94, Sch96]) est un ensemble de pragmas pour le langage Fortran 90 pour la parallélisation automatique. Fortran 90 fournit au programmeur des opérations qui agissent directement sur des matrices et vecteurs. HPF parallélise automatiquement ces opérations en découpant ces opérations en bandes de matrices et éléments de vecteurs, et en les distribuant aux différents processeurs. Cependant, comme nous l'avons vu à la section 1.1 (page 6), les applications de calcul scientifique deviennent de plus en plus complexes, loin d'un schéma aussi basique, et les parallélisations complètement automatiques sont alors incapables d'extraire du parallélisme. De plus, les parallélisations mises en jeu, nécessairement simplistes, ne peuvent pas passer à l'échelle des machines décrites dans la section précédente.

Ainsi, il n'y a pas de solution miracle, et il est nécessaire de modifier les programmes, même légèrement, pour exprimer du parallélisme de manière appropriée. La question qui se pose est la finesse de parallélisme qu'il faut exprimer. Des approches comme Cilk [FLR98] ont montré que pour obtenir les meilleures performances, il faut exprimer le plus de parallélisme possible. Il faut bien sûr savoir rester raisonnable pour éviter que l'expression même du parallélisme prenne plus de temps que les tâches ainsi isolées, mais plus l'environnement d'exécution dispose d'un parallélisme raffiné, plus il sera à même d'effectuer un équilibrage dynamique de charge adapté à la machine cible. C'est d'autant plus vrai dans le cas d'une application très irrégulière pour laquelle on ne sait pas *a priori* comment pourra se répartir la charge.

1.3.2 Des programmeurs non spécialistes en parallélisme

Il est certes nécessaire d'exprimer du parallélisme, mais les programmeurs de codes de calcul scientifique ne sont en général pas des informaticiens, et donc encore moins des spécialistes du parallélisme. De ce point de vue, l'interface de parallélisme la plus standard, les threads POSIX (PTHREAD), est en pratique d'un niveau de programmation bien trop technique pour être réellement utilisable telle quelle. Par ailleurs, en ce qui concerne le placement des données sur machine NUMA, aucun outil n'est fourni.

Pour ces raisons, des interfaces de plus haut niveau ont été conçues. OPENMP [ope, ME99] par exemple, va plus loin qu'HPF en proposant un ensemble d'annotations (sous forme de

`pragmas`) qui permettent d'indiquer au compilateur comment il est possible de paralléliser le code : on indique par exemple quelles boucles sont parallélisables, quels tableaux sont privés ou partagés, etc. Les *Threads Building Blocks* (TBB [tbb]) d'INTEL vont encore plus loin en s'appuyant sur les abstractions de C++. Ainsi, plutôt que de paralléliser de simples boucles indicées par des entiers, il est possible d'exprimer la nature parallèle des objets que l'on manipule, qui peuvent alors être bien plus complexes que de simples matrices. L'environnement d'exécution peut alors créer des threads et répartir les données à l'insu du programmeur pour effectuer les opérations qu'il demande. Il existe par ailleurs des interfaces de programmation parallèle plus explicites telles que le langage UPC [CDC+99], extension au langage C où, comme dans le cas de MPI, le même programme est exécuté plusieurs fois en parallèle, mais cette fois au sein d'un même processus. Des éléments de syntaxe supplémentaires permettent de distinguer variables privées et partagées. La bibliothèque MPC [Pér06] fonctionne sur le même principe, sauf que pour communiquer des valeurs entre les flots parallèles, le programmeur doit utiliser explicitement des fonctions de passage de messages ou des opérations collectives. Le même programme peut dans ce cas être indifféremment exécuté sur une grappe de machines monoprocesseurs, une machine multiprocesseur, ou une grappe de machines multiprocesseurs. C'est l'environnement d'exécution qui s'adapte aux différentes situations. Cela n'empêche pas bien sûr l'environnement d'exécution de décider d'effectuer aussi certaines opérations en parallèle. Ceci introduit en fait assez naturellement la notion de *parallélisme récursif* : une tâche se décompose en sous-tâches, qui peuvent à leur tour se décomposer en sous-tâches, etc. Comme le disent Martorell *et al.* [MAN+99], cela permet d'exprimer d'autant plus de parallélisme pour un équilibrage de charge adapté.

1.4 De l'environnement d'exécution à l'ordonnancement et au placement

L'environnement d'exécution se retrouve ainsi avec un certain nombre de tâches à exécuter et de données à placer sur la machine. Selon les environnements, les tâches sont exprimées concrètement à l'aide de processus, de tâches applicatives, de flots d'exécution,... disons de manière générale, de *threads*. L'objectif est alors de les exécuter sur une machine cible, c'est-à-dire les *ordonnancer*. La subtilité est que cette action ne peut techniquement pas être mise en application de manière *globale* par le *système* puisque ce sont en réalité les *processeurs* qui, lorsqu'ils deviennent inactifs, réaffectent leur compteur ordinal vers un nouveau thread à exécuter. L'ordonnancement consiste donc en une série asynchrone d'actions purement *locales* aux *processeurs*. Nous reviendrons sur ce point important au chapitre suivant. Les données, par contre, peuvent en général être simplement placées explicitement. La problématique est alors de combiner ordonnancement de threads et placement de données : il sera par exemple courant qu'une prévision d'ordonnancement de threads sur les processeurs conditionne le placement des données, et qu'à l'inverse le placement courant des données intervienne dans la décision de l'ordonnancement, ou bien l'on peut encore décider de migrer des données pour pouvoir mieux répartir la charge de calcul, etc. L'objectif est bien sûr d'obtenir les meilleures *performances* possibles, mais aussi d'avoir une certaine *flexibilité* pour pouvoir disposer d'un certain contrôle sur l'ordonnancement. Un autre objectif non moins important est la *portabilité des performances*, c'est-à-dire d'obtenir les meilleures performances quelle que soit la machine sous-jacente sans avoir à modifier l'application ou

l'environnement d'exécution.

Dans cette section, nous faisons un tour d'horizon des approches existantes en les regroupant en trois tendances : les approches opportunistes, plutôt orientées vers un support générique sans information de la part de l'application ; les approches pré-calculées, où les décisions sont essentiellement prises au niveau applicatif ; enfin les approches négociées où l'environnement d'exécution interagit avec l'application.

1.4.1 Approches opportunistes

Les approches opportunistes sont les plus courantes : elles sont utilisées notamment par les systèmes d'exploitation, car elles ne nécessitent pas d'information de la part des applications. Il est à noter que l'apparition progressive des machines décrites à la section 1.2 (page 8) a obligé à améliorer sans cesse les algorithmes pour pouvoir exploiter ces machines.

Stratégies de base

Pour l'ordonnancement de threads, le principe de base est en général le *Self-Scheduling* [TY86, DMKJ96] : chaque processeur, lorsqu'il devient inactif, « pioche » dans une liste globale le prochain thread à exécuter, permettant ainsi de répartir très facilement la charge sur la machine. Le dernier placement de chaque thread est mémorisé pour influencer le choix du thread pioché : on préfèrera ainsi ré-exécuter les threads qui s'étaient exécutés sur ce processeur il y a peu de temps, privilégiant ainsi les affinités que le cache du processeur peut encore avoir avec ce thread. C'est cette technique qui est utilisée dans les systèmes d'exploitation LINUX 2.4 et WINDOWS [Rus97] par exemple. Cependant, utiliser une unique liste de threads pour toute une machine constitue un goulet d'étranglement d'autant plus important lorsque la machine possède de nombreux processeurs. C'est ainsi que l'on utilise les algorithmes d'*AFfinity Scheduling* (AFS [ML94]) ou de *Locality-based Dynamic Scheduling* (LDS [LTSS93]), où l'on utilise plutôt des listes de threads locales à chaque processeur. Les threads sont initialement répartis sur ces listes, et si un processeur n'en trouve plus sur sa liste, il en « vole » à un autre processeur, le plus chargé par exemple. Ces algorithmes sont utilisés par les systèmes d'exploitation plus récents, par exemple LINUX 2.6 [Lin], CELLULAR IRIX [WMB+97], FREEBSD 5.0 [Rob03], UnixWare [CK01]. Ils y ajoutent en général des temps de rééquilibrage : lorsque l'on crée un thread, on le place sur le processeur le moins chargé, et de temps en temps, le processeur le plus chargé confie une partie de son fardeau au processeur le moins chargé. Cependant, à partir d'un certain nombre de processeurs, et surtout pour les machines NUMA, ce n'est plus suffisant. Les dernières versions des systèmes tels que LINUX et FREEBSD suivent l'algorithme *Hierarchical AFfinity Scheduling* (HAFS [WWC00]) en utilisant une hiérarchie de listes de threads avec des stratégies de rééquilibrage local, évitant ainsi des verrouillages globaux intempestifs. Dans le cas d'une machine NUMA, la notion de *homenode*, le nœud où un thread a démarré (et où donc se situent *a priori* toutes ses données) permet par ailleurs lors d'un rééquilibrage de tendre à placer les threads sur le nœud où se trouvent ses données.

Pour l'allocation des pages mémoire, il existe trois stratégies courantes. La plus fréquente (et en général celle activée par défaut par les systèmes d'exploitation) est le *first touch* : c'est seulement lorsqu'elle est accédée la première fois qu'une donnée est réellement allouée, et

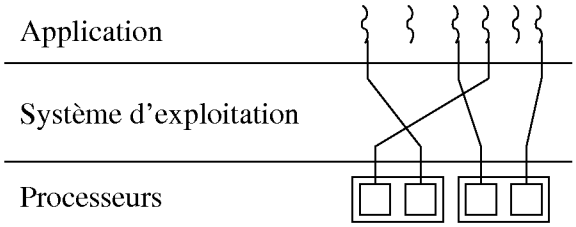


FIG. 1.9 – Ordonnancement opportuniste dans un système d'exploitation.

on le fait sur le nœud du processeur qui a effectué l'accès, en espérant que les accès futurs seront aussi depuis ce processeur. Une autre approche est le *round robin* : de manière similaire au mécanisme décrit à la section 1.2.4 (page 13) les données sont réparties page par page sur les différents nœuds, de manière cyclique, afin de répartir la charge d'allocation et de bande passante. Cela permet de compenser le cas pathologique où toutes les données se retrouvent allouées sur le nœud 0 (voir l'explication du problème qu'ont rencontré Löf *et al.* section 1.2.1, page 9). Un raffinement de cette approche est d'utiliser une période de répartition faisant intervenir un nombre premier. Cela permet d'éviter des situations de résonance entre la période de l'allocation et la période propre à l'application, où toutes les données se retrouveraient allouées sur le mauvais nœud. Ces différentes stratégies sont donc plus ou moins heureuses selon les cas et il n'est pas forcément facile de savoir laquelle utiliser selon l'application et la machine cible.

Raffinements à l'aide de mesures

Pour raffiner ces stratégies, les environnements d'exécution effectuent parfois des mesures et agissent en conséquence durant le déroulement de l'exécution.

Ainsi, Steckermeier et Bellosa [SB95] s'intéressent aux défauts de cache : ils les observent pour modéliser assez précisément le vieillissement de l'affinité qu'a encore le cache d'un processeur avec les threads qui s'y sont exécutés. Ils peuvent ainsi effectuer un ordonnancement influencé par cette affinité.

Fedorova [Fed06] s'intéresse quant à elle au partage de cache sur puce multicœur. En observant les défauts de cache et les compteurs de performances, elle peut détecter les contentions de cache, lorsque trop de threads l'utilisent et donc se gênent mutuellement, et qu'il vaut donc mieux les répartir sur d'autres puces. Elle peut également ajuster les quantums de temps accordés aux threads pour rendre l'exécution plus équitable.

Pour ordonnancer des threads sur puce multithreadée de manière appropriée, de nombreux travaux ont été développés autour de l'*appariement* de threads : McGregor *et al.* [MAN05], Jain *et al.* [JHA02], Bulpin [Bul04], El-Moursy *et al.* [EMGAD06] utilisent tous des compteurs de performances pour évaluer si une paire donnée de threads« s'exécute bien ensemble ». En effet, si certaines paires peuvent être intéressantes (thread de calcul brut avec thread effectuant beaucoup de défauts de cache) et peuvent travailler en *symbiose*, d'autres paires peuvent ne pas l'être (voir section 1.2.3, page 11) et être constituées au contraire de thread *interférant* l'un avec l'autre. Mesurer si la paire de threads actuellement ordonnancée « s'exécute bien » n'est cependant pas évident. Selon les auteurs, les compteurs de per-

formances utilisés pour cela sont très variables : le taux de références externes, le taux de défauts de cache, le nombre d'instructions par seconde, le nombre d'instructions en cours d'exécution, ... Les heuristiques pour combiner ces compteurs sont d'autant plus variées. Il semble donc qu'il n'y ait pas de solution universelle.

Enfin, sur machine NUMA, pour compenser les stratégies d'allocation mémoire simplistes citées précédemment, certains systèmes proposent un mécanisme de migration mémoire automatique. Le principe est d'interroger les compteurs de performances intégrés aux contrôleurs mémoire pour détecter lorsqu'une page est plus souvent accédée de manière distante que locale. Il est alors *a priori* intéressant d'effectuer une migration de cette page sur le nœud qui y accède le plus, pour optimiser les accès de ce nœud-là au détriment des accès moindres du nœud d'origine. C'est ce qui est proposé dans le système IRIX [NPP+02]. Cependant, l'analyse même est difficile, car les informations fournies par les contrôleurs mémoire ne sont pas forcément précises (il est coûteux de mémoriser des statistiques précises pour *toutes* les pages de la machine), la consultation même des informations prend d'autant plus de temps que les statistiques sont précises, et le processus de décision est délicat. BULL, qui tente actuellement de développer un mécanisme similaire pour ses machines NOVASCALE, dépense de l'ordre de 15 %[2] de temps processeur pour l'ensemble de son mécanisme ! De plus, Nikolopoulos *et al.* [NPP+00] indiquent que sans aucune autre information sur le déroulement de l'application (notamment les phases d'initialisation et les itérations de boucles), il est bien difficile de ne pas se tromper en migrant une page. Wilson et Aglietti [WA01] conseillent de disposer de seuils que l'on règle selon l'application. En pratique pour le calcul scientifique, une telle migration automatique est souvent désactivée et l'on utilise plutôt des migrations explicites depuis l'application [NPP+00], plus adaptées aux différentes étapes de son déroulement.

On le voit, il existe ainsi de nombreux travaux sur les ordonnancements et placements opportunistes, d'autant plus avec les nouveaux problèmes que posent les machines contemporaines. L'enjeu est en effet important, car les systèmes d'exploitation n'ont en général que très peu d'interaction avec les applications. Les résultats obtenus sont relativement intéressants, et les mises en œuvre sont en général assez portables. Cependant, les résultats sont la plupart du temps plutôt loin des performances de crêtes. Une remarque que l'on peut faire est que bien souvent l'ordonnancement des threads et le placement des données sont effectués de manière indépendante, même si l'ordonnanceur de threads s'efforce de les placer à côté de leurs données, et que les algorithmes de migration s'efforcent de placer les données là où elles sont le plus utilisées. En fait, le problème principal est que ces approches tentent de se passer d'une information cruciale que les applications pourraient fournir : les affinités entre threads et entre les threads et les données. La figure 1.9 montre comment un système d'exploitation risque ainsi d'attribuer des processeurs aux threads de manière désordonnée.

1.4.2 Approches pré-calculées

À l'inverse des approches opportunistes qui ne font que réagir aux threads que les utilisateurs lancent, les approches pré-calculées effectuent l'exécution en plusieurs étapes. Une première phase éventuelle permet d'étalonner le comportement de la machine. On peut en-

[2] D'après discussion avec les ingénieurs en charge du projet.

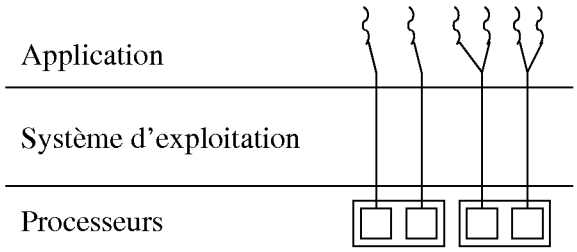


FIG. 1.10 – Ordonnancement pré-calculé, court-circuitant le système d'exploitation.

suite déterminer un ordonnancement des threads et un placement des données adaptés à la complexité de la machine cible (voire optimaux). Enfin, on exécute simplement l'application en exigeant du système d'exploitation l'ordonnancement et placement voulus[3], tel qu'illustré figure 1.10, en supposant que la machine est dédiée à l'application. Les performances obtenues sont alors excellentes.

C'est ainsi qu'en connaissant à l'avance le graphe des tâches à effectuer (y compris leur temps de calcul), Lai et Chen [LC96a] et Narlikar [Nar02] sont à même de les exécuter d'une manière la plus appropriée possible, en privilégiant notamment un parcours local du graphe pour privilégier les affinités et ainsi bénéficier des effets de cache. Acar *et al.* [ABB02] y ajoutent une stratégie de vol local pour compenser les déséquilibres éventuels lorsque l'estimation du temps de calcul n'est pas assez fiable.

Le solveur PASTIX [HRR00] illustre également particulièrement bien cette approche : il résout de (très) grands systèmes linéaires creux par une méthode directe en calculant d'abord un ordonnancement statique des calculs par blocs et des communications, par l'intermédiaire d'une simulation utilisant une modélisation des opérateurs BLAS et de la communication sur l'architecture cible.

Enfin, pour distribuer de manière appropriée les données d'une application sur une machine NUMA sans autre connaissance approfondie de l'application que sa boucle principale, Marathe et Mueller [MM06] effectuent une première exécution de la première itération seulement de l'application, pendant laquelle ils utilisent les compteurs de performances de l'ITANIUM pour obtenir une trace approximative des accès effectués par chaque processeur pendant cette itération. Cette trace permet alors de décider du placement effectif des pages sur les différents nœuds NUMA. L'application est alors relancée avec ce placement mémoire, ce qui améliore grandement les performances par rapport aux approches opportunistes. Par rapport au mécanisme de migration automatique expliqué dans la section précédente, le point clé est que la mesure est effectuée pendant exactement *une itération*. Elle n'est perturbée ni par la phase d'initialisation, ni par le biais qui survient en faveur du début ou de la fin de la boucle principale lorsqu'une mesure dure pendant par exemple une itération et demie. L'estimation qui en résulte est ainsi très caractéristique de l'exécution de l'application.

Les performances obtenues par des méthodes de pré-calcul sont ainsi excellentes. Leur problème est qu'elles ne peuvent être utilisées que pour des applications dont le comportement

[3] Tous les systèmes actuels proposent une interface pour fixer les threads noyau sur les processeurs et les pages mémoire sur les nœuds.

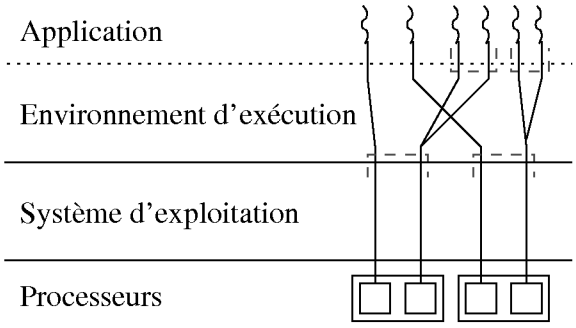


FIG. 1.11 – Ordonnancement négocié, utilisant des informations de l'application et s'adaptant à la machine sous-jacente.

est régulier. Lorsque le comportement de l'application dépend des résultats intermédiaires obtenus, il est impossible de pré-calculer un ordonnancement ou un placement.

1.4.3 Approches négociées

Entre ces deux approches extrêmes, il existe des approches qui, tout en restant plutôt génériques, font intervenir des formes de négociation entre l'environnement d'exécution et la machine sous-jacente.

C'est ainsi que les compilateurs pour langages parallèles (OPENMP, HPF, UPC, etc. décrits à la section 1.3.2, page 16) compilent les programmes de manière suffisamment générique pour pouvoir s'adapter aux différentes architectures parallèles, et ajoutent au code généré une portion qui détermine *à l'exécution* l'architecture de la machine (le nombre de processeurs par exemple) et adapte le démarrage de threads et le placement des données à cette architecture (autant de threads que de processeurs par exemple).

Certains de ces compilateurs (Omni/ST pour OPENMP [TTSY00] par exemple) vont plus loin en embarquant un environnement d'exécution complet au sein de l'application. Celui-ci court-circuite complètement l'ordonnancement effectué par le système d'exploitation en supposant que la machine est dédiée à l'application et en effectuant explicitement en espace utilisateur les changements de contexte entre threads. Certains systèmes d'exploitation (tels que les anciennes versions de SOLARIS ou les versions récentes de FREEBSD) fournissent même le mécanisme de *Scheduler Activation* pour traiter le problème des appels systèmes bloquants que ce genre d'approche rencontre. L'environnement d'exécution embarqué peut alors contrôler complètement l'ordonnancement des threads, ce qui permet de prendre en compte les informations que le compilateur a pu collecter tout autant que l'architecture de la machine cible, tel qu'illustré figure 1.11. Un tel environnement est cependant plutôt difficile à mettre au point, et se limite donc assez souvent à un ordonnancement simple de tâches.

1.4.4 Paramètres et indications utiles à l'ordonnancement

Les approches négociées sont intéressantes, car elles sont à même de prendre en compte toute information utile tout en restant assez génériques. Or des informations peuvent venir

de nombreuses sources.

À l'*exécution*, le système d'exploitation peut fournir des informations sur la machine cible : non seulement le nombre de processeurs mais aussi la structure détaillée, depuis l'organisation des bancs mémoire pour les machines NUMA jusqu'à l'organisation des caches sur les puces multicœurs. Nous avons également vu à la section 1.4.1 (page 19) que les compteurs de performances des processeurs actuels peuvent apporter des informations précieuses sur le déroulement de l'application.

Le *compilateur* peut aussi fournir des indications sur le comportement de l'application. En analysant les schémas d'accès aux données [ZOSD04, SGDA05], il est possible d'estimer comment répartir threads et données pour favoriser les accès locaux et s'assurer que l'occupation mémoire sera possible sur la machine cible. Un compilateur OPENMP peut aussi estimer assez précisément la quantité de données partagées entre threads d'une même section parallèle par exemple. Le projet CAPRICCIO [BCZ+03] utilise également une analyse statique à la compilation pour connaître à l'avance l'utilisation des piles. Le type des instructions émises par le compilateur peut également permettre d'estimer si deux threads pourront être efficacement appariés sur un processeur hyperthreadé.

Enfin, les *programmeurs* eux-mêmes détiennent beaucoup d'informations. Ils connaissent *a priori* assez précisément l'information essentielle citée à la section 1.4.1 (page 18) : les relations entre threads et entre threads et données, c'est-à-dire selon quel schéma les threads communiquent entre eux, et comment ils accèdent aux données. Cela permet par exemple de catégoriser ces dernières pour leur appliquer des stratégies d'allocation différentes [BFS89], ou bien de migration des données lors des migrations de threads. Ils savent si le schéma d'accès aux données est plutôt orienté producteur/consommateur ou bien complètement irrégulier par exemple. Il est ainsi possible de choisir entre des stratégies typiques telles que « allocations localisées + ordonnancement hiérarchisé », ou bien « allocations entrelacées + ordonnancement centralisé ». Les programmeurs ont aussi une assez bonne idée du comportement des threads vis-à-vis de l'ordonnancement : s'ils utilisent plutôt des communications ou bien consomment beaucoup de temps processeur par exemple. Il se peut également que l'on sache *a priori* la charge que représente un thread en temps de calcul et en quantité de mémoire allouée, voire la *charge restante* : le nombre d'itérations qu'il faut encore exécuter par exemple. Enfin, une information a priori anodine mais en fait essentielle est le déroulement du programme : des évènements tels que la terminaison de la phase d'initialisation, le début des itérations d'une boucle principale, les phases de raffinement, etc. sont utiles pour pouvoir appliquer des heuristiques adaptées. En effet, dans l'exemple de la section 1.2.1 (page 9), la phase d'initialisation allouait toutes les données sur le nœud 0. La solution que Löf a trouvée [LH05] est en quelque sorte de notifier au système la fin de cette phase à l'aide de la fonction *Affinity on next touch* du système Solaris. Cette fonction indique en effet que le prochain accès à une page devra activer une migration de la page vers le processeur ayant effectué l'accès. Le meilleur moment pour utiliser cette fonction est précisément à la fin de l'initialisation. De même, nous avons expliqué à la section 1.4.2 (page 21) que connaître avec précision la portée des itérations de boucles permettait d'effectuer des mesures précises.

Ainsi donc quantités d'informations peuvent être disponibles pour prendre des décisions d'ordonnancement et de placement. Les systèmes d'exploitation classiques ont une barrière entre espace noyau et espace utilisateur qui réduit fortement la possibilité de transfert d'information entre l'environnement d'exécution (ici, le noyau) et l'application. Cette

limitation lourde empêche d'obtenir les meilleures performances possibles : Roper et Olsson l'indiquent à propos de leur langage CATAPULTS [RO05] permettant de développer des ordonnanceurs intégrés aux applications : utiliser des informations provenant de l'application permet d'obtenir des gains de performances significatifs. Les approches micro-noyau permettent bien souvent de déporter l'ordonnancement et la gestion mémoire dans l'espace utilisateur, mais en pratique ils sont très peu déployés sur les machines de calcul scientifique.

1.5 Discussion

Traditionnellement, paralléliser une application commence par l'hypothèse « Soient p processeurs,... », et le programmeur découpe et distribue alors son problème sur ces p processeurs. Il a ainsi la sensation de maîtriser l'exécution de son algorithme sur la machine cible. Cependant, nous avons vu que pour pouvoir raffiner les simulations, les algorithmes sont devenus de plus en plus irréguliers voire couplés entre eux, si bien qu'une distribution sur p processeurs ne peut plus se faire *a priori*. De plus, avec les machines contemporaines, l'hypothèse « soient p processeurs » est de plus en plus éloignée de la réalité : les processeurs ne sont pas simplement alignés les uns à côté des autres ! Pour obtenir les meilleures performances, il faut profiter des partages de cache potentiels et il est essentiel de prendre en compte l'architecture NUMA de la machine. On ne peut donc plus se contenter d'un modèle aussi simple, ni même espérer pouvoir prévoir le comportement de l'exécution d'une application scientifique. On est donc obligé d'utiliser un *environnement d'exécution* qui se charge d'exécuter au mieux les tâches parallèles que le programmeur scientifique aura exprimées le plus finement possible.

Nous avons vu que d'un côté les approches opportunistes, par leur généricité, étaient plutôt limitées dans la recherche des meilleures performances, par manque d'informations ; de l'autre, les approches pré-calculées ne sont pas utilisables pour des applications irrégulières. Les approches négociées ont paru être un compromis intéressant, d'autant plus que les informations qu'elles permettent de prendre en compte peuvent être précieuses. D'autre part, il n'existe pas d'ordonnanceur qui serait efficace quelle que soit l'application visée, il paraît donc normal de développer différents environnements d'exécution, adaptés aux différentes grandes classes d'applications de calcul scientifique. Le problème est que pour pouvoir contrôler effectivement l'exécution, il est nécessaire *a priori* de développer un environnement d'exécution complet, ce qui est techniquement difficile et rebute donc *a priori* tout algorithmicien.

Dans le cadre de cette thèse, nous nous intéressons donc aux approches négociées, mais en découpant l'environnement d'exécution en deux parties. Une partie « basse » regroupe des mécanismes d'ordonnancement techniques de base, et fournit une interface de programmation de l'ordonnancement de haut niveau. Une partie « haute », en contact avec le programmeur final d'application *via* un langage de haut niveau, peut alors utiliser l'interface de programmation de l'ordonnancement pour ordonnancer les threads de l'application d'une manière qui est appropriée à cette dernière. Nous nous retrouvons ainsi avec **trois** acteurs. Le *programmeur d'application*, expert dans son domaine scientifique qui n'a bien souvent rien à voir avec l'informatique, peut continuer à programmer dans un langage du plus haut niveau possible pour lui éviter tout problème technique, et le laisser ainsi concentré sur son propre

domaine et ses algorithmes de calcul. Le langage utilisé devra cependant lui permettre d'exprimer, directement ou implicitement, toutes informations utiles dont il dispose, telles que celles citées à la section 1.4.4 (page 22). L'*expert en ordonnancement* implémente alors la partie haute de l'environnement d'exécution à l'aide de l'interface de programmation d'ordonnancement de haut niveau : de manière plutôt algorithmique, il utilise les informations fournies par le programmeur d'application et toute autre information fournie par le compilateur, ou par le matériel lors de l'exécution, pour établir un ordonnancement adapté à l'application. Il peut également implémenter différents algorithmes relativement génériques, le programmeur d'application pouvant alors les tester et sélectionner celui qui a l'air de fonctionner le mieux, avec quelques réglages éventuels. L'*expert technique* enfin, implémente la partie basse de l'environnement d'exécution : il se charge de tous les détails techniques nécessaires à l'implémentation de l'interface de programmation d'ordonnancement. Tout cela est en fait une manière de réaliser l'architecture que le visionnaire Gao *et al.* imaginent pour l'avenir de la programmation parallèle [GSS+06] : les trois acteurs peuvent chacun exprimer son expertise dans un langage approprié, et *expérimenter* différents algorithmes indépendamment les un des autres.

Il existe déjà quelques environnements similaires. Bossa [BM02] fournit par exemple des abstractions d'ordonnancement de haut niveau et un langage pour développer et prouver des ordonnanceurs. L'implémentation actuelle est un ajout au noyau LINUX. Cependant, l'objectif est surtout de pouvoir *prouver* la correction de l'ordonnanceur. Par conséquent, le langage proposé, bien que suffisamment puissant pour implémenter l'ordonnanceur monoprocesseur de LINUX 2.4, est assez restrictif, et limite ainsi beaucoup les programmeurs. Catapults [RO05] propose par ailleurs une approche orientée objet pour écrire un ordonnanceur adapté à l'application cible. Les machines ciblées sont cependant les systèmes embarqués, et donc bien qu'il traite effectivement des applications multithreadées, il ne traite pas des machines multiprocesseurs.

Se pose alors le problème de savoir quelle interface de programmation d'ordonnancement de haut niveau fournir à l'expert en ordonnancement pour qu'il puisse véritablement se concentrer sur une algorithmie de l'ordonnancement, pendant que l'expert technique s'occupe de tous les détails sordides. Comme à l'accoutumée, nous commençons par développer les modélisations des objets mis en jeu : elles doivent être suffisamment simples pour l'expert en ordonnancement tout en étant suffisamment puissantes pour exprimer raisonnablement à la fois la complexité des applications et des machines contemporaines. Elles doivent bien sûr pouvoir être implémentées d'une manière la plus efficace possible. C'est ce que nous détaillons au chapitre suivant.

Chapitre 2

Structurer pour mieux ordonnancer

Sommaire

2.1 Exprimer la structure parallèle d'une application 28

2.1.1 Structurer le parallélisme à l'aide de bulles 28

2.1.2 Qualifier le parallélisme à l'aide de bulles 30

2.1.3 Attacher des informations aux bulles 31

2.1.4 Travaux apparentés 32

2.2 Coller à la structure de la puissance de calcul 33

2.3 Combiner les modélisations 34

Le chapitre précédent nous a amenés à introduire la notion d'une véritable plate-forme de développement comportant une interface d'ordonnancement de haut niveau qu'un expert en ordonnancement pourra utiliser pour se concentrer sur des problèmes algorithmiques plutôt que les détails techniques d'implémentation. Ce chapitre est le premier d'une série de trois chapitres consacrés à la contribution proprement dite de la thèse. Avant de décrire précisément notre plate-forme d'élaboration d'ordonnanceurs ainsi que certains points clés de sa mise en œuvre efficace, nous commençons par présenter l'idée centrale de notre modèle d'ordonnancement, c'est-à-dire une modélisation récursive à la fois de la structure de l'application et de l'architecture des machines multiprocesseurs contemporaines. Du côté applicatif, un mécanisme de regroupement récursif appelé *bulle* permet d'exprimer des informations sur la structure (notamment les affinités entre fils d'exécution) voire le comportement des calculs scientifiques, fournis explicitement par le programmeur de l'application, ou automatiquement collectés par le compilateur ou à l'exécution. Du côté architecture, la complexité des machines actuelles est rationalisée de manière automatique. La problématique de l'ordonnancement de l'exécution de l'application est ainsi ramenée à la combinaison de ces deux modélisations.

Nous commençons donc par exposer les modélisations de l'application et de la machine et leur richesse d'expression avant de donner un aperçu de l'éventail des combinaisons possibles.

2.1 Exprimer la structure parallèle d'une application

Au chapitre précédent, nous avons vu que pour espérer obtenir de bonnes performances lors de l'exécution d'une application scientifique, il était important de prendre en compte des informations sur son comportement. Les relations entre threads et entre threads et données étaient apparues comme l'une des informations essentielles que le programmeur pourrait indiquer, mais pour laquelle il ne dispose pas d'outils d'expression. Nous présentons donc ici un moyen d'exprimer ces relations, c'est-à-dire la structure parallèle même de l'application. Cette structuration permet en fait par ailleurs d'organiser les autres informations décrites à la section 1.4.4 (page 22).

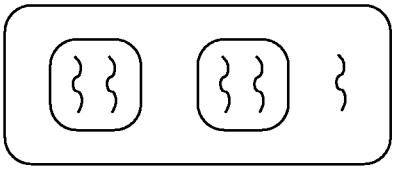
2.1.1 Structurer le parallélisme à l'aide de bulles

L'expression de la structure de l'application est basée sur la notion d'ensembles récursifs imbriqués appelés *bulles*. Une bulle regroupe les objets qui « travaillent ensemble », tels que deux threads qui utilisent les mêmes données par exemple. La notion de bulle ressemble ainsi à la notion de *gang* de tâches du *gang scheduling*, la différence principale est que les bulles peuvent s'imbriquer récursivement. On peut aussi comparer la notion de bulle à celle de communicateur MPI (qui permet d'indiquer à l'environnement d'exécution un ensemble particulier de processus qui communiquent de concert).

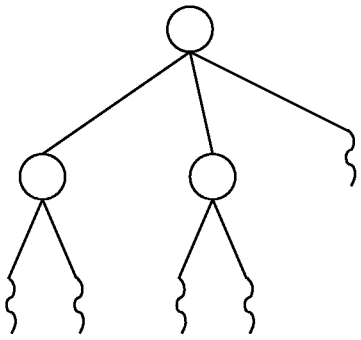
Supposons par exemple qu'une application a découpé le domaine de son calcul en deux morceaux et que, sur chaque morceau, elle a découpé le calcul en deux threads qui peuvent être exécutés en parallèle. Supposons qu'elle a également besoin d'un thread à part, qui ne travaille pas sur des données particulières mais qui s'occupe d'afficher simplement les données d'entrée à l'écran par exemple. La figure 2.1(a) montre une modélisation à l'aide de bulles : les threads de calcul travaillant sur les mêmes données sont regroupées au sein d'une bulle. Ces deux bulles et la tâche d'affichage, puisqu'elles travaillent sur un même domaine, sont elles-mêmes regroupées dans une bulle plus grosse. La figure 2.1(b) montre la même hiérarchie de bulles, mais représentée pour plus de commodités sous forme d'arbre. La figure 2.1(c) donne l'extrait de code permettant d'établir une telle hiérarchie de bulles.

La notion de bulle peut donc être interprétée comme une *classe d'équivalence* par rapport à une relation d'affinité donnée, l'imbrication des bulles signifiant le *raffinement* de la relation par une autre relation : par exemple le partage de données, une synchronisation, des communications, des opérations collectives, etc. Maintenue et adaptée dynamiquement tout au long et selon l'exécution, cette imbrication de bulles permet à toute application d'exposer de façon structurée sa nature parallèle à un ordonnanceur adapté.

Bien sûr, le choix d'une structuration en arbre ne permet pas de modéliser directement n'importe quelle structure d'application : il n'est pas possible d'exprimer une intersection entre deux bulles par exemple (un thread ne peut pas directement appartenir à deux bulles différentes). Dans l'idéal, il faudrait utiliser une structure de graphe (voire d'hypergraphe) qui, elle, est complètement générique. Cependant, comme l'indique Foster [Fos95], une structure de graphe amène très rapidement des algorithmes qui sont NP-complets. La structuration en arbre permet de conserver une complexité polynomiale tout en permettant une expressivité suffisamment intéressante. Dans certains cas (un maillage cartésien régulier par exemple),



(a) Représentation ensembliste.



(b) Représentation arborescente.

```
marcel_t comp_thread[4], comm_thread;
marcel_bubble_t bubbles[2], bubble;

marcel_bubble_init(&bubbles[0]);
marcel_bubble_insertthread(&bubbles[0],&comp_thread[0]);
marcel_bubble_insertthread(&bubbles[0],&comp_thread[1]);

marcel_bubble_init(&bubbles[1]);
marcel_bubble_insertthread(&bubbles[1],&comp_thread[2]);
marcel_bubble_insertthread(&bubbles[1],&comp_thread[3]);

marcel_bubble_init(&bubble);
marcel_bubble_insertbubble(&bubble,&bubbles[0]);
marcel_bubble_insertbubble(&bubble,&bubbles[1]);
marcel_bubble_insertthread(&bubble,&comm_thread);
```

(c) Code source correspondant.

FIG. 2.1 – Expression des relations entre threads.

il pourra être utile d'attacher des informations de topologie (les coordonnées au sein du maillage par exemple) que l'ordonnanceur pourra utiliser de manière particulière. Au pire, il est possible d'approximer la structure de l'application (un maillage plus ou moins régulier par exemple) à l'aide d'un arbre, par décomposition hiérarchique par exemple. En pratique, la structure d'arbre se prête souvent bien à modéliser les applications, comme on le voit dans la section suivante.

2.1.2 Qualifier le parallélisme à l'aide de bulles

Une telle structuration en arbre est en fait assez naturelle : les applications sont le plus souvent déjà structurées en arbre, que ce soient les approches « diviser pour régner », le raffinement récursif d'un code AMR, les sections parallèles imbriquées d'OpenMP, etc., de manière générale tout parallélisme récursif. Ainsi, une application peut simplement exprimer la structure naturelle de son parallélisme. Cette expression peut être explicite, telle que donnée en exemple figure 2.1(c), ou bien implicitement construite par la partie « haute » de l'environnement d'exécution, comme détaillé dans le chapitre 5 (page 77). On pourrait même imaginer un mécanisme de découverte automatique d'affinités basée sur la détection de prise des mêmes mutexes ou l'accès à un même tableau par exemple. À l'inverse, l'application peut interroger l'environnement d'exécution sur la composition de la machine pour savoir par exemple quel degré de parallélisme il peut être intéressant d'exhiber ou le nombre de morceaux qu'un découpage de données devrait produire pour qu'elles puissent être réparties sur les différents nœuds NUMA.

Les relations entre threads qui sont ainsi exprimées peuvent être de natures assez variées, le plus souvent combinées. Ces relations ont alors des conséquences sur les bonnes propriétés que peut avoir un ordonnancement :

Partage de données Il arrive souvent que des threads travaillent en fait sur les mêmes données, lorsqu'ils travaillent sur un même bloc d'une matrice, par exemple. Il est alors intéressant d'essayer de rapprocher ces threads sur une même puce pour profiter des effets de cache, ou du moins éviter que certains d'entre eux se retrouvent sur un autre nœud NUMA, et soient alors pénalisés par le facteur NUMA.

Communications Sans aller jusqu'à un partage de données, des threads peuvent avoir à communiquer des résultats intermédiaires. Il est donc important de tendre à les placer de manière proche, ne serait-ce que sur des nœuds NUMA voisins pour limiter le facteur NUMA mais aussi limiter l'utilisation en bande passante du réseau d'interconnexion 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 threads concernés ont tous 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. Selon les applications, il pourra alors être intéressant de s'assurer que ces threads restent relativement proches pour permettre une synchronisation plus locale donc plus efficace *a priori*, ou bien il vaudra mieux au contraire étaler les threads le plus possible pour assurer une exécution parallèle et obtenir alors une terminaison plus rapide.

HyperThreading On peut aussi vouloir exploiter de manière fine l'*HyperThreading* : si l'on

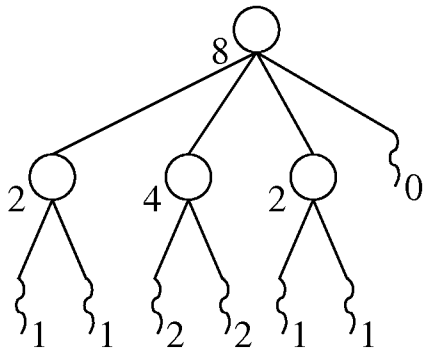


FIG. 2.2 – Synthèse de l'estimation d'utilisation du temps processeur.

sait que deux threads sauront s'exécuter en *symbiose*, c'est-à-dire de ne pas se gêner lorsqu'ils sont exécutés en parallèle sur les deux processeurs logiques d'un même processeur physique, il est intéressant de pouvoir les regrouper pour qu'ils soient *vraiment* exécutés ensemble sur un même processeur physique. Un tel mécanisme n'est pas fourni par les interfaces des systèmes d'exploitation actuels.

Ordonnancement Lorsque l'on couple des codes parallèles de natures différentes, il peut être utile d'appliquer des stratégies d'ordonnancement différentes aux threads des différents codes. Ces stratégies peuvent de plus elles-mêmes nécessiter un contexte propre à chaque code de simulation parallèle exécuté. Utiliser une bulle par code de simulation permet d'exprimer ceci d'une manière très naturelle.

2.1.3 Attacher des informations aux bulles

Au-delà d'exprimer simplement la structure de l'application, une hiérarchie de bulles peut aussi servir de *support* aux informations détaillées à la section 1.4.4 (page 22).

Dans les différents cas de regroupement énumérés à la section précédente, on peut ainsi *quantifier* le degré de regroupement : la quantité de données partagées ou communiquées par exemple, ou bien de manière plus abstraite un simple *coefficient d'élasticité* qui exprime de manière relative combien un ordonnanceur pourra se permettre d'éloigner les threads et sous-bulles contenus dans une bulle. On peut également exprimer des *contraintes* : l'occupation mémoire des threads peut être importante par exemple, auquel cas il faudra faire attention de la répartir correctement en regard de la quantité de mémoire disponible sur les différents nœuds NUMA de la machine. L'utilisation de bande passante mémoire peut également être suffisamment importante pour qu'il faille la répartir correctement. Une estimation de l'utilisation en temps processeur permet par ailleurs d'effectuer une bonne répartition de charge *a priori*. On peut éventuellement faire la synthèse de ces contraintes en les sommant en remontant dans la hiérarchie de bulles comme illustré figure 2.4(f) : chaque thread indique ici une estimation de sa charge en temps processeur, et par somme on peut connaître de la charge de tout une bulle, ce qui permet alors de prendre des décisions à très haut niveau. Toutes ces informations pourraient être directement fournies ou du moins estimées par le programmeur d'application, mais comme nous l'avons vu à la section 1.4.4, le compilateur peut souvent les estimer lui-même à partir de l'analyse du programme.

Les informations sur l'exécution proprement dite de l'application peuvent également être synthétisées le long de la hiérarchie de bulles. Dans le cas des informations d'ordonnancement, le fait qu'un thread est vivant ou endormi par exemple peut être résumé sous la forme du nombre de threads vivants et endormis à l'intérieur d'une bulle et de ses sous-bulles. Pour les informations fournies par le programmeur ou le compilateur, l'estimation du temps de calcul restant pour un thread par exemple peut être accumulée pour obtenir une estimation de la charge totale en temps processeur de toute une bulle et son contenu. Enfin, les informations fournies par les compteurs de performances peuvent être synthétisées : on peut faire calculer le long de la hiérarchie de bulles la moyenne des taux de défauts de cache, du nombre d'instructions par seconde, etc. Ceci permet d'avoir un résumé hiérarchisé du comportement actuel de l'application en cours d'exécution.

2.1.4 Travaux apparentés

Il existe souvent dans les systèmes d'exploitation des mécanismes de regroupement ressemblant aux bulles décrites ci-dessus. On peut citer notamment les *Process AGGregates* (PAGG [sgi]) de SGI, la `liblgroup` [Sun] de SUN, les *NUMA Scheduling Groups* (NSG [Com]) de COMPAQ, la `libnuma` et les *cpuset* de LINUX. Une première différence, importante, est que ces mécanismes ne sont pas récursifs : il n'est pas possible de construire des groupes de groupes, etc. Ils ne permettent donc pas d'exprimer la nature récursive des applications parallèles, mais au contraire posent surtout la question « combien de threads mettre par groupe ? », alors que la construction récursive de bulle apporte la question plus naturelle « à quel niveau de parallélisme se situe ce thread ? ». D'ailleurs, au sein de l'interface des *Thread Building Blocks*, INTEL permet justement au programmeur de préciser ce niveau de parallélisme, pour que les stratégies de vol implémentées dans l'environnement d'exécution puissent privilégier la localité et minimiser les vols. De plus, l'objectif de tels mécanismes concerne d'avantage l'administration (partage de machine, quotas, comptabilité, etc.) que les performances. Ainsi, bien que l'on puisse effectivement indiquer pour un groupe l'ensemble des processeurs sur lequel on veut restreindre son exécution, rien n'est fourni pour attacher des informations aux groupes ou pour avoir un réel contrôle flexible sur l'ordonnancement au sein des groupes. Il est à noter des travaux très récents au sein du noyau LINUX qui généralisent encore la notion de PAGG en des *containers* [Men07] qui pourront, eux, être récursifs. L'objectif reste toujours principalement l'administration, mais il est déjà question de faire intervenir la notion de *container* pour l'ordonnancement, pour l'équité seulement cependant. Le nouvel ordonnanceur de LINUX 2.6.23, CFS (*Completely Fair Scheduling*), intègre en effet dans la version 2.6.24 la notion de *Group Scheduling* où la distribution de temps CPU est équitable par rapport aux groupes plutôt qu'aux processus individuels, empêchant ainsi par exemple un utilisateur de monopoliser les processeurs en lançant de nombreux processus. Au moment de l'écriture de ces lignes, l'implémentation ne supporte pas encore la récursivité, mais on conçoit déjà que cette dernière sera naturelle : on peut trier les threads par processus, puis par groupe de processus et par session (au sens UNIX), et enfin par utilisateur, voire par espace de *pid* !

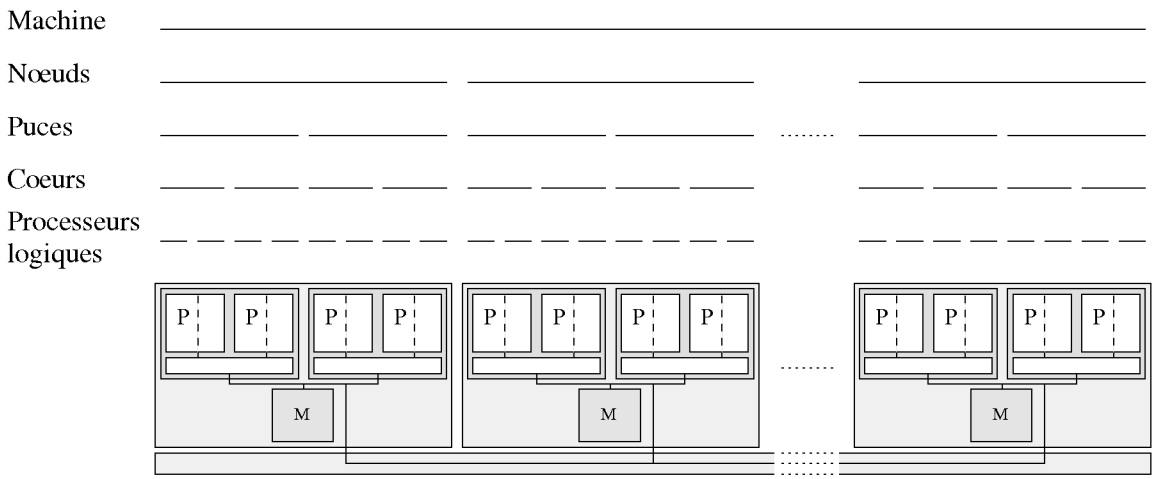


FIG. 2.3 – Modélisation d'une machine très hiérarchisée par une hiérarchie de listes.

2.2 Coller à la structure de la puissance de calcul

Une des difficultés lorsque l'on écrit un ordonnanceur est due au modèle de VON NEUMANN : ce sont les processeurs eux-mêmes qui exécutent les instructions d'un thread et basculent éventuellement sur un autre lorsque le précédent se termine ou se bloque en attente d'une ressource. Ainsi, *a priori*, on ne *confie* pas un thread à un processeur, c'est plutôt le processeur qui doit de lui-même déterminer le prochain thread à exécuter. Toute la difficulté est alors que les différents processeurs doivent parvenir à faire ce choix *de concert* et le plus rapidement possible. Nous avons vu à la section 1.4.1 (page 18) que la solution généralement adoptée par les algorithmes opportunistes était de disposer de listes de threads dans lesquelles les processeurs viennent piocher. Par ailleurs, Dandamudi et Cheng [DC97], Oguma et Nakayama [ON01] et Nikolopoulos *et al.* [NPP98] s'accordent à dire qu'il vaut mieux utiliser au moins deux niveaux de listes de threads pour à la fois éviter des contentions et pouvoir facilement répartir la charge.

Nous poussons donc la logique jusqu'au bout en modélisant l'architecture de la machine cible par une hiérarchie de listes de threads. À chaque élément de chaque étage de la hiérarchie de la machine est associé (*bijectivement*) une liste de threads. La figure 2.3 montre une machine très hiérarchique et sa modélisation. La machine en entier, chaque nœud NUMA, chaque puce physique, chaque cœur, et chaque processeur logique d'un même cœur (SMT) possède ainsi une liste de threads prêts. On utilise alors, comme pour les stratégies opportunistes, un ordonnancement de base de type *Self Scheduling* : chaque processeur, lorsqu'il doit choisir le prochain thread à exécuter, examine les listes qui le « couvrent » à la recherche du thread le plus prioritaire. L'identification entre élément physique et liste de threads permet ainsi de déterminer le domaine d'exécution d'un thread donné : placé sur une liste associée à une puce physique, ce thread pourra être exécuté par tout processeur de cette puce ; placé sur la liste globale il pourra être exécuté par tout processeur de la machine. En outre, une notion de priorité permet de s'assurer de la réactivité d'éventuels threads de communication par exemple.

Il serait même aisé de modéliser un ensemble de machines NUMA reliées par un réseau

à capacité d'adressage (tel que SCI[1] [Sol96]), ou des machines exécutant un système SSI (*Single System Image*) tel que Kerrighed [MLV+03]. Il suffit simplement d'ajouter un niveau « réseau ».

Bien entendu, suivre l'organisation physique de la machine n'est pas obligatoire. Par exemple, lorsque l'*arité* entre les éléments d'un niveau et du niveau inférieur est grande et risque d'entraîner des problèmes de contention, des groupes artificiels de processeurs peuvent eux aussi être modélisés, de la même façon que Wang *et al.* le font pour leur algorithme CAFS (*Clustered AFinity Scheduling* [WWC97]).

La construction de cette modélisation s'effectue bien sûr de manière automatique lors du démarrage de l'application, en interrogeant le système d'exploitation sur le nombre de nœuds NUMA et la topologie du réseau d'interconnexion, le détail du partage des caches, etc. Tous les systèmes contemporains fournissent une interface donnant ce genre de renseignements. Il est bien sûr nécessaire pour chaque système de réécrire les fonctions d'interrogation, mais les différentes interfaces se ressemblant beaucoup, cela est plutôt aisé. Il est à noter que WINDOWS VISTA, dernier système à ce jour à avoir ajouté une telle interface, fournit une très grande quantité de détails tels que l'associativité des caches !

Certaines machines n'ont pas une hiérarchie régulière : sur la figure 1.7 (page 14), nous avons vu un exemple de machine dont le réseau d'interconnexion était asymétrique. D'autres réseaux d'interconnexion sont organisés en *tore 2D*, dans les machines X1 de CRAY ou les ALPHASERVER d'HP par exemple. La modélisation sous la forme d'une hiérarchie de listes n'est alors pas le reflet exact de la réalité, et il faudra à l'avenir utiliser des modèles plus complexes. En pratique, notre modèle hiérarchique reste cependant une assez bonne approximation de telles machines.

Dans le cas où la machine cible permet de débrancher et rebrancher des processeurs à chaud, ou si elle est partitionnée pour plusieurs utilisateurs, la structure de la machine reste tout de même la même, et l'on désactive et réactive simplement dynamiquement les niveaux devenus inutiles.

2.3 Combiner les modélisations

Une fois que l'on dispose de la structure de l'application sous forme d'une hiérarchie de bulles et de threads d'une part, et de la structure de la machine sous la forme d'une hiérarchie de listes d'autre part, le principe est de distribuer la hiérarchie de bulles et de threads sur les listes. En reprenant l'exemple de l'application qui avait été modélisée avec des bulles à la figure 2.1, exécutée par exemple sur une machine double-bicœur, deux distributions extrêmes sont représentées figure 2.4. Sur la figure 2.4(a), toute la hiérarchie de bulles (et donc l'ensemble des threads) a été placée tout en haut de la hiérarchie de listes. Une telle distribution permet certes d'utiliser tous les processeurs de la machine, puisque tous ont l'opportunité d'exécuter n'importe quel thread. Cela ne prend cependant pas en compte les affinités entre threads et processeurs, puisqu'aucune relation entre eux n'est réellement utilisée. À l'inverse sur la figure 2.4(b), toute la hiérarchie de bulles (et donc tous les threads)

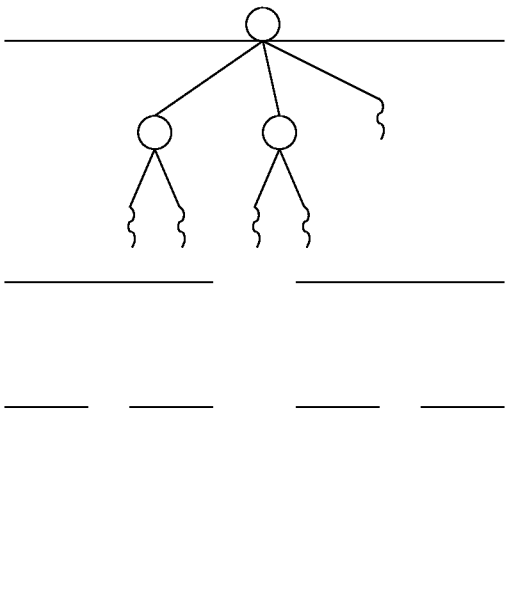
[1] Un réseau SCI permet de définir des segments de mémoire partagés par plusieurs machines de façon transparente, la machine SEQUENT NUMA-Q [LCS96] utilise cette technique.

a été placée sur la liste d'un seul des processeurs. La prise en compte des affinités est ici maximale : tous les threads seront exécutés par le même processeur ! Cependant, puisque l'application n'a pas d'autres threads à exécuter, tous les autres processeurs restent inactifs. La figure 2.4(c) montre comment les threads peuvent être distribués de manière spatiale en prenant fortement les affinités en compte, puisque les threads sont en fait ici correctement répartis sur les processeurs selon leurs affinités. Cependant, si certains threads s'endorment, les processeurs correspondants deviennent inactifs, et l'utilisation des processeurs est donc potentiellement partielle. La figure 2.4(d) montre enfin comment une distribution de threads sur les processeurs peut être incorrecte du point de vue des affinités : on voit des relations entre bulles et threads se croiser.

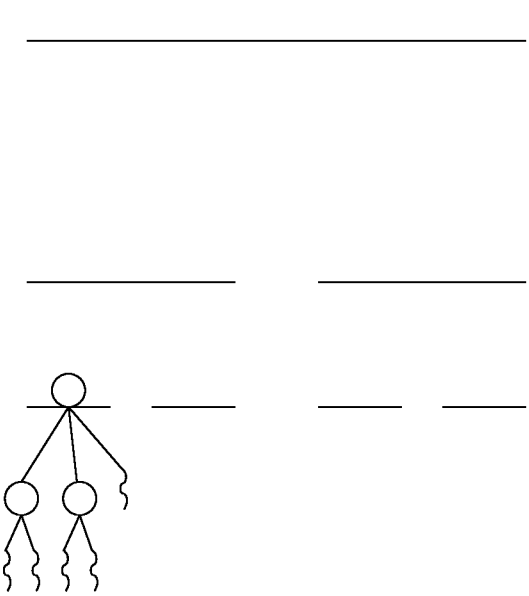
Toute la problématique se ramène alors à évoluer entre les différentes distributions possibles, en évitant les écueils tels que les placements extrêmes 2.4(a) et 2.4(b), au profit de placements intermédiaires tels que 2.4(c) qui établissent un compromis entre distribution de charge et prise en compte des affinités. Il faudra bien sûr autant que possible éviter des situations telles que 2.4(d). Les informations attachées aux bulles détaillées à la section 2.1.3 permettent de raffiner les choix de placement. Dans l'exemple de la figure 2.2, un ordonnancement simpliste pourrait aboutir au placement représenté figure 2.4(e). L'ordonnanceur *Spread*, que l'on présente plus loin à la section 3.2.3 (page 48), utilise l'information synthétique de charge pour obtenir une répartition plus équilibrée représentée figure 2.4(f) : il a commencé par distribuer la bulle de poids le plus fort (4) sur la première moitié de machine, et les deux autres (2+2=4) sur l'autre moitié.

Une fois une certaine distribution choisie, les processeurs peuvent, grâce à leur algorithme d'ordonnancement de base, déterminer rapidement quels threads ils devraient exécuter. On peut alors laisser l'application s'exécuter un certain temps : si la distribution est appropriée, l'exécution devrait être efficace et il n'y a pas de raison de redistribuer. Cependant, lorsque des évènements surviennent, la création, l'endormissement ou la terminaison de threads par exemple, il sera sans doute nécessaire de redistribuer. Ainsi, les décisions prises pour la distribution de threads et de bulles sont plutôt à *moyen terme*, en suivant les grandes étapes du déroulement de l'application.

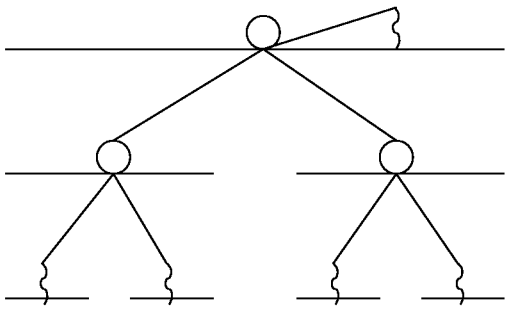
Pour réaliser de telles distributions, nous proposons une véritable plate-forme permettant de développer des *ordonnanceurs à bulles* adaptés aux différents types d'application. Cette plate-forme est décrite au chapitre suivant.



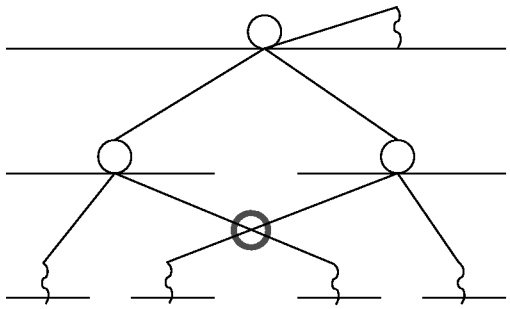
(a) Bonne utilisation processeur, affinités non prises en compte.



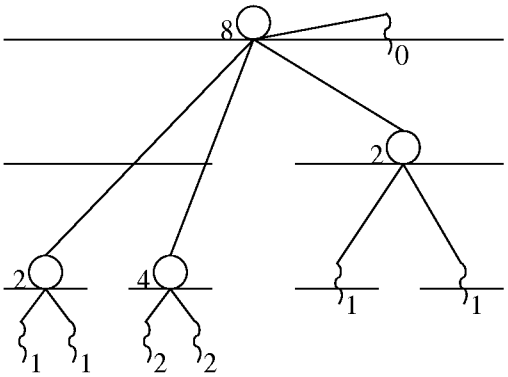
(b) Mauvaise utilisation processeur, affinités extrêmement prises en compte.



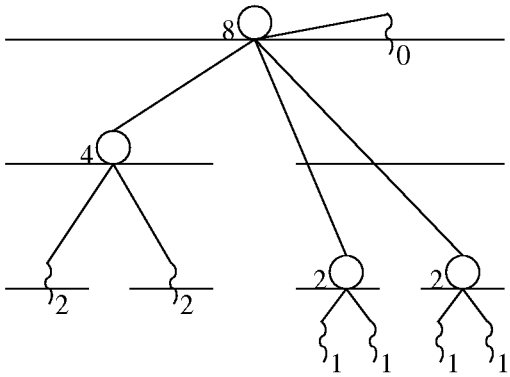
(c) Utilisation distribuée des processeurs, affinités correctement prises en compte.



(d) Utilisation distribuée des processeurs, affinités mal prises en compte.



(e) Répartition simpliste, non équilibrée.



(f) Répartition selon la charge estimée.

FIG. 2.4 – Distributions possibles des threads et bulles sur la machine.

Chapitre 3

BubbleSched : une plate-forme pour l'élaboration d'ordonnanceurs portables

Sommaire

3.1 Une interface pour programmer 38
3.1.1 Côté applicatif, construire une information hiérarchisée 38
3.1.2 Côté ordonnancement, gérer les hiérachies de listes et de bulles . . . 38
3.1.3 Calculer et maintenir automatiquement des statistiques liées aux bulles 42
3.2 Quelques exemples pour s'inspirer 44
3.2.1 *Burst*, un ordonnancement par niveau 44
3.2.2 *Gang*, un ordonnancement par tourniquet 46
3.2.3 *Spread*, un ordonnancement par équilibrage de charge 48
3.2.4 *Steal*, un ordonnancement par vol de travail 51
3.2.5 *Affinity*, un ordonnancement par affinités 54
3.2.6 *MemAware*, un ordonnancement dirigé par la mémoire 55
3.2.7 Discussion 57
3.3 Un environnement pour expérimenter 58
3.3.1 Combiner des ordonnanceurs 58
3.3.2 Comprendre les performances 58
3.3.3 Multiplier les tests 60

Ce chapitre présente le côté concret de la contribution de la thèse : une plate-forme appelée BubbleSched mettant en œuvre les concepts théoriques qui ont été introduits au chapitre précédent. Elle fournit aux experts en ordonnancement de quoi développer des ordonnanceurs établissant des combinaisons de modélisations adaptées à l'application et la machine mises en jeu, et ce de manière portable.

L'organisation de ce chapitre s'articule comme suit. L'interface de programmation des bulles est d'abord exposée. Des exemples d'ordonnanceurs typiques sont ensuite détaillés pour montrer comment l'interface peut être utilisée de manière variée. Enfin, nous proposons des outils permettant d'expérimenter de manière poussée différents algorithmes d'ordonnancement.

3.1 Une interface pour programmer

L'interface de programmation des bulles se décompose en deux parties. Du côté applicatif, des outils sont fournis pour construire les bulles et y attacher des informations. Du côté ordonnancement, nous proposons une infrastructure pour développer des ordonnanceurs sous forme d'une série de fonctions à implémenter pour réagir à différents évènements.

3.1.1 Côté applicatif, construire une information hiérarchisée

La construction des bulles peut être effectuée soit explicitement par le programmeur applicatif, soit implicitement par le compilateur à partir d'informations fournies par le programmeur à l'aide du langage. Le principe est de commencer par initialiser une bulle à l'aide d'une fonction `init()`, puis d'y insérer des threads ou d'autres bulles à l'aide de fonctions `insertthread()` et `insertbubble()`. Pour pouvoir effectuer l'insertion d'un thread en même temps que sa création, il est également possible de positionner un attribut de création à l'aide de la fonction `marcel_attr_setinitbubble()`. On peut alors « lancer » la bulle à l'aide d'une fonction `wake_up_bubble()`, ce qui prévient l'ordonnanceur à bulles de l'arrivée d'une nouvelle bulle. Durant l'exécution, il est possible de réutiliser `insertthread()` et `insertbubble()`, ainsi que `removethread()` et `removebubble()` pour modifier dynamiquement la hiérarchie de bulles. Pour attendre la terminaison de tous les threads et sous-bulles, on peut utiliser la fonction `join()`. L'ensemble des fonctions fournies est énuméré à l'annexe A.1 (page 112). La figure 3.1 montre en gras les appels de fonction qu'il suffit d'ajouter pour construire automatiquement un exemple de hiérarchie de bulles adaptée au calcul naïf de la suite de FIBONACCI

3.1.2 Côté ordonnancement, gérer les hiérachies de listes et de bulles

Le programmeur d'ordonnanceurs à bulles dispose d'une riche interface de programmation pour pouvoir parcourir la machine et placer bulles et threads.

Pour simplifier la programmation, les threads et les bulles peuvent être indifféremment considérés comme **entités** (*entities*). De manière similaire, les listes de threads (aussi appelées *runqueues*) et les bulles peuvent être indifféremment considérées comme **conteneurs**. Ainsi on peut simplement dire que les *conteneurs* (listes et bulles) peuvent contenir des *entités* (threads et bulles). L'annexe A.4 (page 117) détaille les fonctions de conversion entre les différents types.

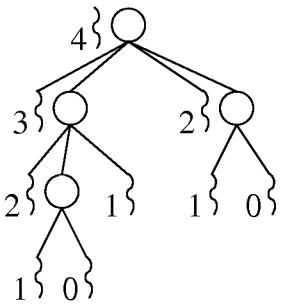
Pour écrire un ordonnanceur à bulles, il suffit alors d'écrire une série de méthodes qui sont appelées lors d'évènements particuliers. La méthode `init` est appelée pour initialiser l'ordonnanceur, et la méthode `exit`, pour le terminer. La méthode `submit` est appelée lorsqu'une nouvelle bulle est soumise par l'application (par la fonction `marcel_wake_up_bubble()`) et permet à l'ordonnanceur à bulles de s'adapter à la nouvelle situation. Rappelons que l'ordonnancement de base, exécuté par chaque processeur lorsqu'il devient inactif, est de piocher dans les listes qui le couvrent un thread à exécuter. Si aucun thread n'a été trouvé pendant cette recherche, la méthode `idle` de l'ordonnanceur à bulles est appelée. L'ordonnanceur peut alors par exemple effectuer un vol de travail ou

```
/* Initialiser une bulle. */
marcel_bubble_init(marcel_bubble_t *b);

/* Insérer un thread dans une bulle. */
marcel_bubble_insertthread(marcel_bubble_t *b, marcel_t *t);

/* Insérer une bulle dans une autre bulle. */
marcel_bubble_insertbubble(marcel_bubble_t *b, marcel_bubble_t *sb);

/* Prédéfinir l'insertion d'un thread dans une bulle. */
marcel_attr_setinitbubble(marcel_attr_t *attr, marcel_t *b);
```

(a) Extrait de déclaration des fonctions de manipulation de bulles

```
void * fibonacci(void *arg) {
  int n = (intptr_t) arg;
  void *r1, *r2;
  marcel_t t1, t2;
  marcel_attr_t attr;
  marcel_bubble_t b;

  if (n <= 1)
    return 1;

  marcel_attr_init(&attr);
  marcel_bubble_init(&b);
  marcel_bubble_insertbubble(bubble_holding_task(marcel_self()), &b);
  marcel_attr_setinitbubble(&attr, &b);

  marcel_create(&t1, &attr, fibonacci, (void*)(intptr_t) (n-1));
  marcel_create(&t2, &attr, fibonacci, (void*)(intptr_t) (n-2));

  marcel_join(t1, &r1);
  marcel_join(t2, &r2);
  marcel_bubble_join(&b);

  return (intptr_t)r1 + (intptr_t)r2;
}
```

(b) Exemple d'enrichissement du calcul naïf de la suite de FIBONACCI à l'aide de bulles.



(c) Hiérarchie de bulles créée par un appel à `fibonacci(4)`.

FIG. 3.1 – Exemple pratique de construction imbriquée de bulles.

bien prendre le temps de redistribuer toute une bulle sur tout ou partie de la machine. Il retourne alors une valeur booléenne indiquant s'il est utile de réessayer de trouver un thread à ordonnancer ou si le processeur doit réellement devenir inactif (lorsqu'il n'a pas pu voler de travail par exemple). Si l'ordonnanceur de base a trouvé un thread pendant son parcours des listes, il l'exécute simplement. Si par contre il trouvé une bulle, la méthode `sched` de l'ordonnanceur à bulles est appelée. L'implémentation par défaut parcourt récursivement la bulle et ses sous-bulles à la recherche d'un thread à ordonnancer. On verra à la section 3.2.1 que l'ordonnanceur *Burst* implémente un autre comportement. La méthode `tick` est enfin appelée régulièrement (toutes les 10 ms) avec en paramètre la bulle qui contient le thread en cours d'exécution. Un ordonnanceur peut ainsi par exemple faire « vieillir » la bulle et au bout d'un certain temps effectuer une action appropriée.

Un ordonnanceur à bulles peut également lancer lui-même des threads qui passent la plupart de leur temps à dormir, et de temps en temps se réveiller pour effectuer des réordonnancements, faisant office de *démons*, donc. On peut alors y utiliser tous les mécanismes de synchronisation usuels entre threads : mutexes, variables conditionnelles, etc.

Selon les besoins des algorithmes, l'ordonnanceur peut parcourir les listes modélisant la machine de différentes manières. Lors d'une redistribution sur tout ou partie de la machine, on peut utiliser un accès vectoriel comme illustré figure 3.2(a). Lorsqu'on s'intéresse à un niveau donné de la machine, les nœuds NUMA par exemple, il est possible d'accéder directement aux listes de ce niveau, comme illustré figure 3.2(b). Enfin, lors des vols de travail notamment, on pourra utiliser un parcours arborescent à partir du processeur courant par exemple, comme illustré figure 3.2(c).

Le parcours d'une liste donnée s'effectue alors à l'aide d'une macro `ma_runqueue_foreach(rq, &e)`, qui énumère les entités `e` contenues sur la liste `rq`. Lorsqu'on rencontre une bulle, on peut également énumérer son contenu à l'aide de la macro `ma_bubble_foreach(b, &e)`. On peut éventuellement découvrir ainsi des sous-bulles, les parcourir, etc.

Dans les deux cas, on peut enfin utiliser les fonctions `ma_get_entity(e)` et `ma_put_entity(e, h)` pour prélever une entité de son conteneur actuel et le déposer dans un autre conteneur, ou plus directement `ma_move_entity(e, h)` pour déplacer une entité dans un autre conteneur. Ces opérations en apparence simples permettent en fait d'obtenir toutes les distributions de bulles et threads exposées au chapitre précédent.

Lors du parcours des conteneurs, et encore plus lorsque l'on déplace une entité entre différents conteneurs, il faut *verrouiller* les conteneurs. Des primitives simples sont donc fournies pour verrouiller et déverrouiller les conteneurs individuellement. On verra à la section 4.2.3 (page 68) les détails de convention de verrouillage. Assez souvent le verrouillage fin n'est pas nécessaire, lorsque l'on redistribue complètement une hiérarchie de bulles par exemple, et l'on peut alors se contenter d'utiliser une fonction qui verrouille la totalité de la machine et des bulles, permettant alors d'effectuer des migrations dans toutes les directions sans se soucier des conventions de verrouillage.

```
for (int i=0; i<marcel_topo_nblevels; i++) {
  for (int j=0; j<marcel_topo_level_nbitems[i]; j++) {
    ma_runqueue_t *rq = &marcel_topo_levels[i][j].sched;
    ...
  }
}
```

(a) Parcours vectoriel.

```
for (int j=0; j<marcel_topo_nbnodes; j++) {
  ma_runqueue_t *rq = &marcel_topo_node_level[j].sched;
  ...
}
```

(b) Parcours direct.

```
void f(struct marcel_topo_level *l, struct marcel_topo_level *son) {
  ...
  // Parcourir les frères d'abord
  for (int i=0; i<l->arity; i++)
    // Éviter de revenir en arrière
    if (l->children[i] != son)
      f(l->children[i], NULL);
  ...
  // Parcourir plus haut
  f(l->father, l);
  ...
}

f(marcel_topo_vp_level[marcel_current_vp()],NULL);
```

(c) Parcours arborescent.

FIG. 3.2 – Différents types de parcours des listes.

3.1.3 Calculer et maintenir automatiquement des statistiques liées aux bulles

Assez souvent, il est utile que certaines informations soient attachées aux threads et bulles, ainsi que détaillé à la section 1.4.4 (page 22), par exemple la quantité de mémoire allouée, la charge de calcul estimée, ou encore des statistiques d'exécution telles que le taux de défauts de cache. Il est alors utile de disposer de résumés hiérarchiques de ces statistiques lorsqu'un ordonnanceur de threads veut prendre des décisions pour les bulles les plus externes par exemple, sans avoir à parcourir lui-même toute la hiérarchie de bulles. Nous avons donc mis en place un mécanisme hiérarchique générique de gestion de statistiques attachées aux bulles et threads. La figure 3.3 montre un exemple de définition de la charge d'un thread et de son utilisation dans un morceau d'algorithme à bulles simpliste.

Le principe est donc similaire à la fonction POSIX `pthread_key_create` : pour chaque type de statistique (nombre de threads actifs, taux de défauts de cache, charge mémoire, etc.), on appelle la fonction `ma_stats_alloc()` qui retourne un identifiant[1], que l'on peut alors utiliser pour accéder à la statistique d'un thread ou d'une bulle donnée grâce à la macro `marcel_stats_get(t, kind)`. Un appel à une fonction `ma_bubble_synthesize_stats(b)` permet alors de calculer un résumé hiérarchisé pour toute une hiérarchie de bulles, et l'on peut alors accéder au résumé statistique pour une des bulles de la hiérarchie grâce à la macro `ma_bubble_hold_stats_get(b, kind)`. Pour pouvoir effectuer le résumé, on doit fournir lors de l'appel à la fonction `ma_stats_alloc()` deux méthodes, `reset` et `synthesis`. `Reset` doit, à partir de l'adresse d'une statistique, remettre celle-ci à une valeur neutre, 0 pour l'addition, par exemple. `Synthesis` est la fonction de synthèse : à partir des adresses d'une statistique d'*accumulation* et d'une statistique *source*, elle doit accumuler la statistique source dans la statistique d'accumulation. Pour l'addition, on peut par exemple utiliser l'opérateur +=. Les méthodes `reset` et `synthesis` sont déjà fournies pour l'addition et le maximum d'entiers longs. Un certain nombre de statistiques sont prédéfinies, telles que le nombre de threads vivants, le nombre de threads actifs, la quantité de mémoire allouée, etc., le détail est disponible à l'annexe A.3 (page 115). Selon les applications, on pourra définir toute autre statistique utile pour prendre des décisions au sein de l'ordonnanceur.

Du point de vue du programmeur d'application, il suffit alors simplement d'utiliser la macro `marcel_stats_get(t, kind)` pour accéder à la statistique `kind` d'un thread donné `t`, c'est-à-dire la consulter ou la modifier. Le programmeur (ou le compilateur) peut ainsi par exemple indiquer une estimation de la charge en temps de calcul qu'un thread représente. La macro `marcel_bubble_stats_get(b, kind)` permet quant à elle d'accéder à la statistique `kind` d'une bulle donnée `b`, permettant ainsi d'indiquer par exemple l'« élasticité » d'une bulle.

Du point de vue du programmeur d'ordonnanceur, les mêmes macros permettent d'accéder aux statistiques des bulles et threads. La fonction `ma_bubble_synthesize_stats(b)` permet en outre de calculer un résumé hiérarchisé des statistiques. En effet, chaque bulle possède, en plus de sa zone de statistiques pour son propre usage, une zone de statistiques

[1] En fait, comme pour `pthread_key_create`, c'est un *offset* qui est retourné. Tous les threads et bulles ont une zone de mémoire dédiée aux statistiques, et `ma_stats_alloc()` alloue simplement pour la statistique une partie de ces zones pour stocker la donnée (c'est le même *offset* dans toutes les bulles et tous les threads existants ou à venir).

Au sein de MARCEL :

```
void ma_stats_long_sum_reset(void *dest) {
  long *data = dest;
  *data = 0;
}
void ma_stats_long_sum_synthesis(void * dest, const void * src) {
  long *dest_data = dest;
  const long *src_data = src;
  *dest_data += *src_data;
}

/* Exécuté à l'initialisation */
void ma_stats_init(void) {
  ...
  marcel_stats_load_offset = ma_stats_alloc(ma_stats_long_sum_reset,
      ma_stats_long_sum_synthesis, sizeof(long));
}
```

Au sein de l'application,

```
*(long*)marcel_stats_get(thread, marcel_stats_load_offset) = load;
```

Au sein de l'ordonnanceur,

```
void _explode_big(marcel_bubble_t *b) {
  ma_entity_t *e;
  if (ma_bubble_hold_stats_get(b) > BIG_LOAD) {
    explode(b);
    ma_bubble_foreach(b, &e)
      if (e->type == BUBBLE)
        _explode_big(ma_bubble_entity(e));
  }
}

void explode_big(marcel_bubble_t *b) {
  ma_bubble_synthesize_stats(b);
  _explode_big(b);
}
```

FIG. 3.3 – Exemple de définition et d'utilisation d'une statistique.

d'accumulation. Cette fonction parcourt alors récursivement en profondeur toutes les sous-bulles de la bulle `b`, et accumule à chaque fois les statistiques des sous-bulles et threads à l'aide des méthodes `reset` et `synthesis`, et ce pour tous les types de statistiques à la fois. Une fois cette fonction appelée, il suffit d'utiliser la macro `ma_bubble_hold_stats_-get(b, kind)` pour consulter le résumé synthétique correspondant à toute la descendance d'une bulle `b` pour la statistique `kind`.

Une autre approche aurait été de tenir les statistiques à jour en permanence, c'est-à-dire que lorsqu'une valeur change pour une entité, il faut mettre à jour la valeur correspondante pour la bulle qui la contient, puis pour la bulle qui contient cette bulle, etc. Dans le cas où la statistique est un maximum, on pourrait même s'arrêter dès que cela n'est plus utile. Une approche plus légère, évitant d'avoir à verrouiller les bulles pour assurer un calcul correct, serait de marquer les bulles pour lesquelles la statistique doit être recalculée. Cependant, cela ajoute tout de même un certain surcoût aux opérations de l'ordonnanceur de base, exécuté très souvent. Les décisions d'un ordonnanceur à bulles étant *a priori* prises plutôt à moyen terme, la collecte complète de statistiques n'est pas effectuée souvent, et n'apparaît donc pas coûteuse en comparaison.

3.2 Quelques exemples pour s'inspirer

Selon les applications, les besoins en ordonnancement sont très différents : les affinités entre threads, cache et mémoire, les étapes de synchronisation, la répartition de temps de calcul, etc. peuvent varier très fortement d'une application à une autre. Il est donc inévitable d'implémenter des ordonnanceurs variés qui établissent chacun un compromis adapté à un certain type d'application. Un programmeur d'application peut par ailleurs tester différents ordonnanceurs, essayer de régler quelques paramètres, et choisir finalement celui qui semble fonctionner le mieux pour son application.

Cette section présente quelques exemples d'ordonnanceurs à bulles qui ont déjà été développés. Ces exemples sont très variés, allant d'une simple répartition très guidée à une répartition fine avec vol de travail, en passant par la rigueur du *Gang Scheduling*. *A priori*, il est préférable de « prévenir plutôt que guérir », c'est-à-dire d'essayer d'obtenir une répartition initiale la meilleure possible plutôt que corriger les déséquilibres *a posteriori*. Cependant, sur des applications très irrégulières, il est très difficile de prévoir une bonne répartition initiale, et donc les algorithmes de redistribution prennent aussi une place importante.

3.2.1 *Burst*, un ordonnancement par niveau

Le premier algorithme à avoir été implémenté est une stratégie de répartition basique pour laquelle l'application indique, en même temps qu'elle crée des bulles, à quels niveaux celles-ci doivent « éclater », à l'aide de la fonction `marcel_entity_setschedlevel()`. Les bulles et threads sont alors répartis sur la machine de manière opportuniste en faisant éclater les bulles aux niveaux indiqués. Le code source est détaillé à la figure 3.4.

Cet ordonnanceur fournit tout d'abord la méthode *sched*, appelée par l'ordonnanceur de base pendant son parcours des listes. Lorsque celui-ci trouve une bulle, au lieu de la par-

```
marcel_t sched(entity_t *e, ma_runqueue_t **sched_rq) {
  ma_runqueue_t *rq = *sched_rq;

  if (e->level > rq->level) {
    /* Entité pas assez basse, la faire descendre. */
    int state = ma_get_entity(e);
    ma_runqueue_unlock(rq);
    for(rq = cpu_rq[cpu_self()]; e->level < rq->level; rq = rq->father)
      ;
    ma_runqueue_lock(rq);
    ma_put_entity(e, rq, state);
    *sched_rq = rq;
  }

  /* Un thread, le retourner pour exécution. */
  if (ma_entity_type(e) == MA_THREAD_ENTITY)
    return ma_thread_entity(e);

  /* Une bulle, l'éclater et laisser l'ordonnanceur de base recommencer. */
  marcel_bubble_t *b = ma_bubble_entity(e);
  ma_bubble_foreach(b, &e)
    ma_move_entity(e, rq);
  return NULL;
}

void idle(int num) {
  ma_bubble_gather(&marcel_root_bubble);
}

void tick(marcel_bubble_t *b) {
  if (!--b->timeslice) {
    /* Le timeslice de la bulle a expiré, la reformer */
    ma_bubble_gather(b);
    b->timeslice = BUBBLE_TIMESLICE;
  }
}

void *daemon(void *arg) {
  while(1) {
    marcel_sleep(60);
    ma_bubble_gather(&marcel_root_bubble);
  }
}

marcel_t t;
void init(void) {
  marcel_create(&t, NULL, daemon, NULL);
}
void exit(void) {
  marcel_cancel(t, NULL);
}
```

FIG. 3.4 – Code source de l'ordonnanceur *Burst*.

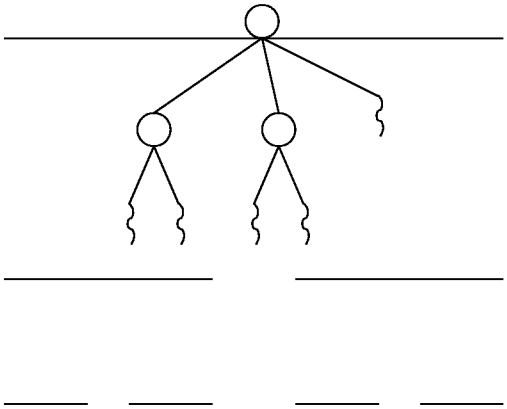
courir pour trouver un thread à exécuter (le comportement par défaut), l'algorithme *Burst* fait descendre cette bulle vers le processeur courant jusqu'à une liste d'un niveau au moins aussi profond que celui attribué à cette bulle par le programmeur d'application. Une fois déplacée sur cette liste, la bulle est « éclatée », c'est-à-dire que son contenu (threads et sous-bulles) est libéré sur la liste. L'algorithme de base peut alors reprendre pour découvrir ce contenu. Dans le cas de threads, il peut alors les exécuter ; dans le cas de sous-bulles, la méthode *sched* sera de nouveau appelée pour faire descendre une des sous-bulles encore plus bas vers le processeur courant. Cet ordonnanceur effectue ainsi une répartition opportuniste prenant en compte les affinités : ce sont les processeurs qui « tirent » vers eux de manière opportuniste les bulles et threads, mais les bulles ne sont éclatées que progressivement, permettant ainsi de répartir leurs threads de manière groupée. La figure 3.5 illustre l'exécution de cet algorithme sur la hiérarchie de bulles et threads donnée en exemple à la section 2.1.1 (page 28).

Cet ordonnanceur fournit de plus une méthode *idle*, appelée lorsqu'un processeur devient inactif. Cette méthode « rassemble » la hiérarchie à l'aide de la fonction `ma_bubble_gather()`, c'est-à-dire que chaque bulle voit son contenu revenir à elle, le résultat final est ainsi de revenir à l'étape de la figure 3.5(a). Lorsque la hiérarchie de bulles et threads évolue, la distribution opportuniste peut ainsi recommencer avec la nouvelle hiérarchie. Il est également possible de programmer un rassemblement périodique, soit individuellement pour chaque bulle à l'aide de la méthode *tick*, soit de manière globale en lançant à l'initialisation un thread démon qui lance régulièrement un rassemblement global.

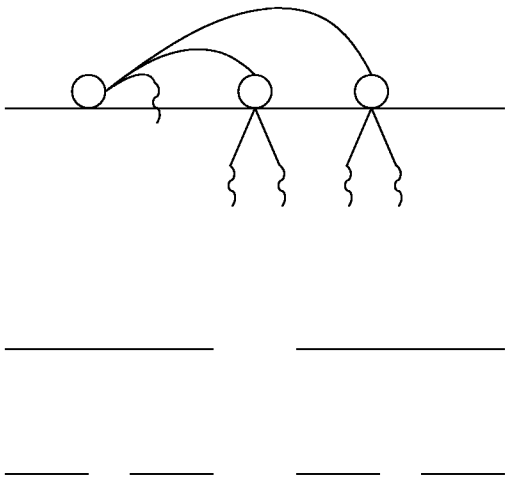
Le principal défaut de cet ordonnanceur est évidemment qu'il suppose que le programmeur d'application fournisse explicitement les niveaux auxquels les bulles doivent exploser. Lorsqu'ils ne sont pas fournis, il est supposé par défaut que la bulle racine doit exploser sur la liste de la machine, les sous-bulles de la bulle racine doivent exploser sur les sous-listes de cette liste, etc. Cet ordonnanceur reste tout de même particulièrement intéressant pour le cas où l'on dispose de paires de threads dont on sait qu'ils s'exécutent bien ensemble sur un processeur hyperthreadé : on peut alors créer une bulle pour chaque paire et indiquer pour chaque bulle qu'elle doit descendre jusqu'au niveau des cœurs, où les deux threads de la bulle pourront ainsi être exécutés par les deux processeurs logiques.

3.2.2 *Gang*, un ordonnancement par tourniquet

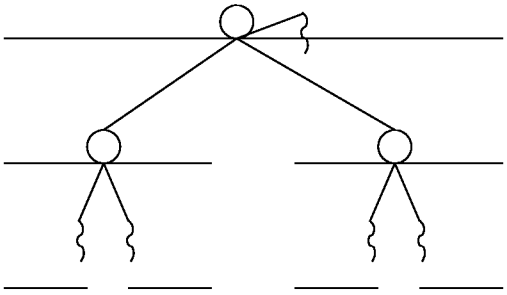
Un autre ordonnanceur assez basique est l'implémentation du *Gang Scheduling*. Avec l'émergence dans les années 80 des réseaux de machines multiprocesseurs, OUSTERHOUT [Ous82] propose de regrouper les threads et données par affinités sous forme de *gangs*, et d'ordonnancer non plus des threads sur les processeurs de l'ensemble des machines, mais des gangs sur des machines : chaque gang est exécuté dans sa totalité (threads et données) sur une même machine, sur laquelle aucun autre thread ne tourne. Ce principe se modélise naturellement dans notre plate-forme en utilisant une bulle pour chaque gang. Pour effectuer le *gang scheduling* proprement dit, il suffit de lancer un thread démon qui effectue la rotation des bulles, voir le code de la figure 3.6. Il utilise une liste `nosched_rq` supplémentaire, non consultée par l'ordonnanceur de base, où il met de côté toutes les bulles (i.e. les gangs) sauf une, qu'il laisse sur la liste principale pendant un certain laps de temps, avant de recommencer pour placer une autre bulle.



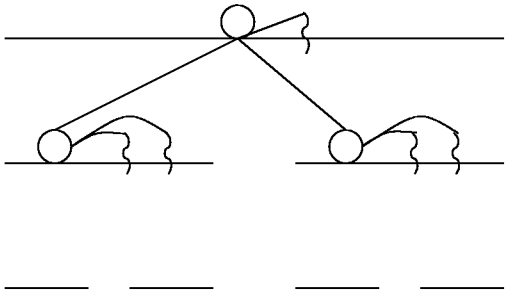
(a) La bulle principale a été placée sur la liste de la machine.



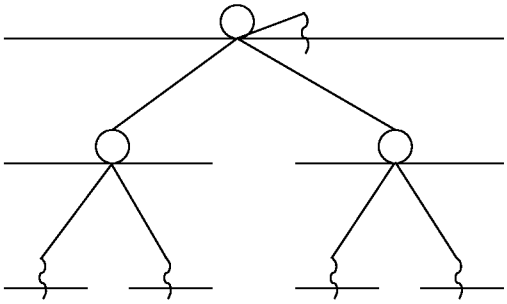
(b) Elle éclate, libérant le thread d'affichage qui peut déjà être exécuté et deux sous-bulles qui peuvent descendre dans la hiérarchie.



(c) Descente effectuée.



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



(e) Les threads de calcul sont bien répartis et peuvent démarrer en parallèle.

FIG. 3.5 – Exécution de l'ordonnanceur *Burst*.

Le résultat correspond bien aux attentes : la figure 3.7 montre les temps d'exécution de 3 gangs contenant respectivement 5, 6 et 7 threads sur une machine quadriprocesseur : les 400% de temps CPU disponibles sont équitablement répartis entre les différents gangs, puis entre les différents threads de chaque gang : $5 * 26,6 = 133\%$ sont attribués au gang 1, $6 * 22,2 = 133\%$ sont attribués au gang 2, et $7 * 19 = 133\%$ sont attribués au gang 3.

Il est bien sûr possible de modifier facilement la tranche de temps utilisée si elle se révèle non adaptée. Il est, de plus, possible de lancer plusieurs *gangs schedulers* en parallèle : on peut, par exemple, lancer un thread pour chaque nœud NUMA en lui passant en paramètre l'adresse de la liste correspondant à ce nœud, pour lequel il effectuera un *gang scheduling*. Ceci est illustré figure 3.8, et c'est ce qui a été utilisé pour exécuter l'application SuperLU, détaillée à la section 6.2.2 (page 93).

Il est possible d'aller encore plus loin : dans ce modèle originel du gang scheduling, des processeurs risquent de rester inactifs parce qu'une même machine ne peut exécuter qu'un gang à la fois, même si celui-ci est trop « petit » par rapport au nombre de processeurs disponibles. FEITELSON *et al.* [FR96] proposent un contrôle hiérarchique des processeurs, pour pouvoir exécuter plusieurs gangs se partageant alors la même machine. Une telle approche peut être implémentée en exécutant un thread démon pour chaque élément de la hiérarchie de listes. Une certaine synchronisation est nécessaire entre ces threads, mais cela peut être facilement effectué à l'aide de sémaphores usuels, par exemple.

3.2.3 *Spread*, un ordonnancement par équilibrage de charge

L'ordonnanceur *Spread* utilise un algorithme plus évolué pour distribuer bulles et threads sur la machine en prenant en compte une notion de charge fournie par exemple par l'application. En fait, il met en œuvre une généralisation de l'algorithme glouton classique utilisé pour résoudre le problème de *Bin Packing*, la différence étant qu'ici le but n'est pas simplement de répartir des objets dans n conteneurs, mais de répartir une arborescence d'objets (la hiérarchie de bulles et de threads) sur une arborescence de conteneurs (la hiérarchie de listes de threads). Alors que cette dernière induit simplement une récursivité de l'algorithme, l'arborescence d'objets permet un certain "relâchement" du problème. En effet, dans le problème de *Bin Packing*, même lorsqu'un objet est « gros », on est obligé de l'attribuer à un conteneur. Dans notre cas, les objets « gros » sont typiquement des bulles, que l'on peut donc éventuellement se permettre de « percer », c'est-à-dire de considérer plutôt leur contenu, qui sera alors plus facile à répartir. Le pseudo-code de l'algorithme est représenté figure 3.9 (les détails de la gestion des tableaux triés ont été volontairement omis pour plus de clarté). La fonction principale récursive prend en paramètre un ensemble de listes de threads (au départ, la liste de la machine) et l'ensemble des entités à distribuer dessus. L'objectif est alors de distribuer les entités sur les listes, données en paramètres, puis de recommencer récursivement. La stratégie utilisée est de calculer d'abord la charge moyenne que l'on pourrait idéalement obtenir pour chaque liste, de percer les bulles qui sont plus grosses que cette charge moyenne, et de distribuer gloutonnement le résultat sur les sous-listes. Les entités « petites » (a priori des threads de communication ou toute autre tâche très peu consommatrice de temps processeur, mais nécessitant une bonne réactivité) sont par contre laissées le long de la hiérarchie pour leur permettre d'être exécutées par n'importe quel processeur, de manière plus immédiate en réaction à une entrée/sortie par exemple. La figure 3.10 montre le déroulement de

```
ma_runqueue_t nosched_rq;

void *daemon(void *arg) {
  ma_runqueue_t *work_rq;   while(1) {
    /* Attendre une tranche de temps. */
    delay(timeslice);

    ma_rq_lock(&work_rq);
    ma_rq_lock(&nosched_rq);

    /* Mettre de côté toutes les bulles actives. */
    ma_runqueue_for_each_entry(&work_rq, &e) {
      ma_get_entity(e);
      ma_put_entity(e, &nosched_rq);
    }

    /* Déposer une seule bulle sur la liste principale. */
    if (!ma_runqueue_empty(&nosched_rq)) {
      e = ma_runqueue_entry(&nosched_rq);
      ma_get_entity(e);
      ma_put_entity(e, &work_rq);
    }

    ma_runqueue_unlock(&work_rq);
    ma_runqueue_unlock(&nosched_rq);
  }
}

marcel_t t;
void init(void) {
  marcel_create(&t, NULL, daemon, &ma_main_rq);
}
void exit(void) {
  marcel_cancel(t, NULL);
}
```

FIG. 3.6 – Code source du *Gang Scheduler*.

```
                name  cpu%  s  cpu
          gang sched   0.0  I    0
       gang1-thread1  26.4  R    3
       gang1-thread2  26.6  R    0
       gang1-thread3  26.8  R    2
       gang1-thread4  25.6  R    1
       gang1-thread5  26.6  R    3
       gang2-thread1  21.9  R    2
       gang2-thread2  22.2  R    0
       gang2-thread3  22.5  R    3
       gang2-thread4  22.1  R    2
       gang2-thread5  21.9  R    0
       gang2-thread6  22.6  R    1
       gang3-thread1  19.2  R    3
       gang3-thread2  18.9  R    1
       gang3-thread3  19.3  R    0
       gang3-thread4  19.8  R    2
       gang3-thread5  19.1  R    3
       gang3-thread6  19.3  R    1
       gang3-thread7  19.2  R    2
```

FIG. 3.7 – Statistiques d'exécution fournies par notre outil `top`.

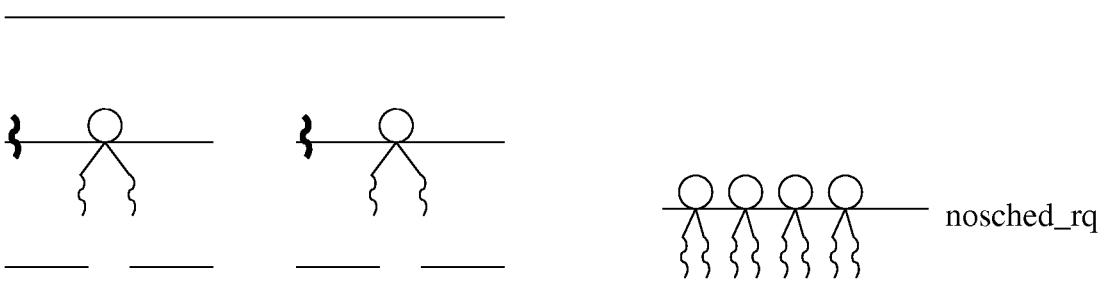


FIG. 3.8 – Exécution de deux *gang schedulers* en parallèle (en gras), chacun piochant dans une liste commune une bulle à faire exécuter sur la partie de machine dont il a la charge.

l'ordonnanceur *Spread* sur la hiérarchie de bulles donnée en exemple figure 2.4(f) (page 36)

Toute la difficulté est de choisir *quand* et *quelle* bulle percer. Ici la tactique est plutôt simple : on privilégie avant tout l'équilibrage de charge. Cette tactique pourrait bien sûr être affinée par des heuristiques qui décideraient de manière plus adaptée quelles bulles éclater en fonction de la finesse de charge que leur contenu pourrait apporter. Nous verrons à la section 3.2.5 que l'ordonnanceur *Affinity* utilise une stratégie de distribution similaire à *Spread*, mais que la tactique est au contraire de limiter le plus possible les éclatements, quitte à obtenir des déséquilibres de charges que l'on compensera dynamiquement ultérieurement.

Par ailleurs, lors de la distribution gloutonne des entités sur les sous-listes, il serait possible de prendre en compte des contraintes telles que l'encombrement mémoire ou l'utilisation de bande passante. Antonopoulos *et al.* ont en effet montré [ANP03] que pour certaines applications on obtient de piètres performances si l'on ne prête pas attention à la bande passante intrinsèque de la machine. Une autre approche est d'utiliser simplement pour la charge une estimation de la bande passante utilisée, et l'algorithme *Spread* distribue alors l'utilisation de la bande passante sur la machine. *Spread* peut être particulièrement utile si l'on exécute moins de threads qu'il n'y a de processeurs : les threads se retrouvent distribués le plus largement possible sur la machine et peuvent ainsi profiter du maximum de bande passante possible sans avoir recours à un artefact de numérotation tel qu'expliqué à la section 1.2.4 (page 15).

Blikber *et al.* [BS05] proposent un algorithme de distribution de parallélisme imbriqué assez similaire à cet algorithme *Spread*. Cependant, leur algorithme ne prend pas en compte la structure hiérarchique de la machine, et les équipes de threads se retrouvent alors souvent « à cheval » sur deux nœuds NUMA par exemple.

Se pose enfin la question : *quand* appeler l'algorithme *Spread* ? En effet, il vaut mieux par exemple attendre que l'application ait fini de créer ses threads et ses bulles avant de les distribuer. On voit ici, de nouveau, l'utilité de savoir quand l'initialisation de l'application est terminée (voir section 1.4.4 page 22). Par ailleurs, si un certain nombre de threads terminent leur travail et que des processeurs deviennent ainsi inoccupés, pendant que d'autres threads ont été créés ailleurs sur la machine par exemple, il est *a priori* intéressant de refaire une distribution adaptée à la nouvelle charge, on verra à la section 3.2.5 comment utiliser la méthode *idle* pour l'effectuer. Par contre, lorsque l'application entre en phase de terminaison, une redistribution risque d'être intempestive.

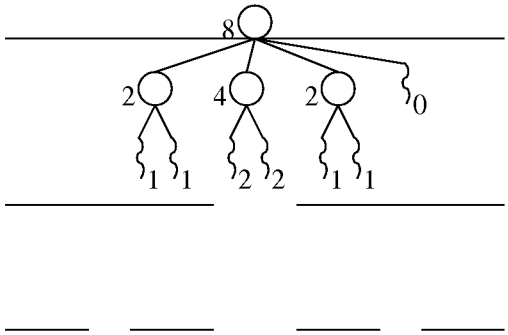
3.2.4 *Steal*, un ordonnancement par vol de travail

Une approche complètement différente est le *vol de travail* implémenté par l'ordonnanceur *Steal*. Le principe de base est que l'on dépose initialement toutes les bulles et tous les threads tout en bas sur la liste du premier processeur. Lorsque la méthode *idle* est appelée par un processeur inactif, elle utilise les pointeurs père / fils des listes pour chercher du travail à voler localement d'abord (sur la liste de l'autre core de la même puce par exemple), puis plus globalement, jusqu'à trouver du travail. L'implémentation actuelle utilise pour l'instant un simple parcours arborescent semblable à celui présenté figure 3.2(c) (page 41).

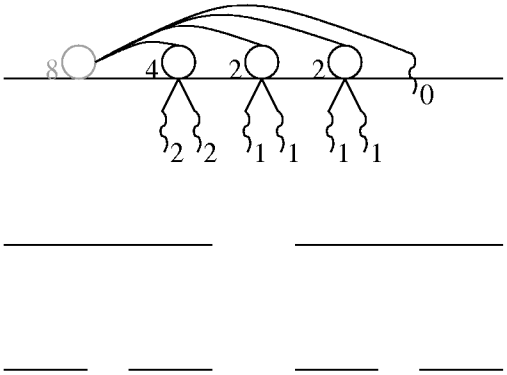
Savoir quelle entité voler et comment la voler est par contre un problème algorithmique non trivial : seule une partie de la hiérarchie de bulles devrait être tirée vers le processeur inactif.

```
void spread(ma_entity_t entities[], ma_runqueue_t lists[]) {
  /* Calculer la charge moyenne par liste. */
  int totload = 0;
  foreach entity in entities do totload += load(entity);
  int avgload = totload / size(lists);

  /* Trier les entités par charge décroissante. */
  qsort(entities, decreasing_load);

  /* Éclater les bulles les plus grosses. */
  int i = 0;
  while (i<size(entities) and load(entities[i]) <= avgload) {
    if (type(entities[i]) == BUBBLE) {
      bubble = entities[i];
      remove(entities, i);
      foreach entity in bubble do
        insert_sorted(entities, entity, decreasing_load);
    } else i++;
  }

  if (size(entities) == 1) return;
  /* Listes triées par charge croissante. */
  ma_runqueue_t *sublists[size(lists)];
  int subload[size(lists)];
  for i in 0 .. size(lists) do {
    sublists[i] = lists[i];
    subload[i] = 0;
  }

  /* Parcours des entités par charge décroissante. */
  for i in 0 .. size(entities) do {
    if (load(entities[i]) < avgload * TINY_RATIO) {
      /* Entité trop petite (probablement communications), la laisser sur place. */
      ma_move_entity(sublists[0]->father);
      continue;
    }

    /* Déposer l'entité sur la liste la moins chargée. */
    ma_move_entity(entities[i], sublists[0]);
    subload[0] += load(entities[i]);
    /* Garder les listes par ordre croissant de charge. */
    re_sort(sublists, subload);
  }

  for i in 0 .. size(lists) do
    if (sublists[i]->children)
      /* On peut encore distribuer plus bas, continuer. */
      spread(sublist[i]->entities, sublists[i]->children);
}
```

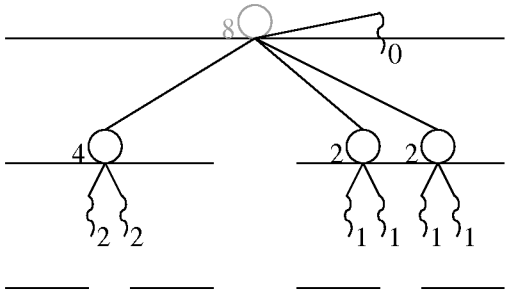
FIG. 3.9 – Pseudo-code source de l'ordonnanceur *Spread*.



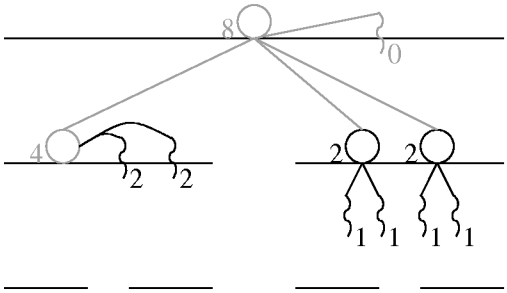
(a) Situation de départ : une bulle de charge 8.



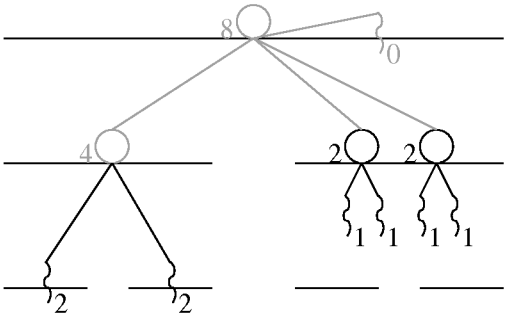
(b) La bulle est éclatée, libérant 3 bulles et un thread, triés par charge décroissante.



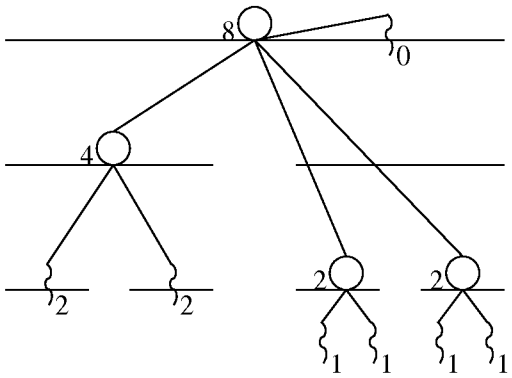
(c) Les bulles ont été réparties gloutonnement sur les sous-listes ; le thread, non chargé, est resté sur la liste de la machine. On peut effectuer les appels récursifs.



(d) La bulle de charge 4 est éclatée, libérant 2 threads.



(e) Les deux threads sont répartis sur les sous-listes à gauche ; l'algorithme est terminé pour la partie gauche.



(f) Les deux bulles sont réparties sur les sous-listes à droite ; l'algorithme est terminé.

FIG. 3.10 – Déroulement de l'ordonnancement *Spread*.

De plus, la structure de la hiérarchie devrait être prise en compte : plus qu'une simple migration de threads, il faudrait donc véritablement effectuer une sorte d'« étirement local de l'arbre des bulles » le long de la machine. En effet, si l'on n'y prend pas garde, voler aveuglément des threads ou des bulles conduit rapidement à rompre les affinités entre threads et produire des croisements tels qu'illustré figure 2.4(d) (page 36). Tous les attributs et statistiques attachés aux bulles décrits dans la section 2.1.3 (page 31) devraient également être soigneusement pris en compte de manière heuristique pour obtenir une répartition bien adaptée à l'application. En effet, selon les cas il faudra privilégier plutôt une répartition de l'occupation mémoire, ou parfois plutôt de la bande passante nécessaire ; un compromis est ici à trouver.

Cet algorithme est assez semblable à l'approche utilisée pour le langage CILK [FLR98]. L'environnement d'exécution fourni pour ce dernier met en effet en œuvre un vol de travail très léger, à un niveau très fin. Le programmeur d'application peut spécifier lors d'un appel de fonction que le résultat n'est pas nécessaire immédiatement, et qu'un autre processeur peut donc « voler » la suite, c'est-à-dire continuer en parallèle l'exécution du programme après l'appel de fonction. Le parallélisme ainsi exprimé est récursif : une fonction ainsi appelée peut elle-même en appeler d'autres, etc. Le vol de travail est alors plutôt effectué au niveau le plus haut pour limiter le nombre de vols. Cependant, et une discussion avec Leiserson l'a confirmé, le vol de travail ne prend aucunement en compte les affinités entre tâches ni la structuration de la machine. La justification est que les programmes Cilk effectuent typiquement une recherche récursive telle que le jeu d'échec, où les affinités entre tâches sont très faibles. À l'inverse, notre algorithme peut être influencé dans son choix de vol par des indications (éventuellement implicites) de l'application.

3.2.5 *Affinity*, un ordonnancement par affinités

L'ordonnanceur *Affinity*, développé par François BROQUEDIS pendant son stage de Master 2 [Bro07] au sein de l'équipe dans le cadre du projet ANR PARA, est assez semblable à l'ordonnanceur *Spread* : il distribue récursivement bulles et threads sur la machine. Cependant, il met l'accent sur les affinités plutôt que sur la répartition de charge. En effet, il ne perce de bulle que s'il y est contraint pour pouvoir fournir au moins un thread à exécuter pour chaque processeur, quitte à obtenir alors un déséquilibre de charge. Par contre, le choix des bulles à percer peut prendre en compte toute information utile fournie par l'application, soit implicitement (il vaut mieux *a priori* percer les bulles contenant le plus de threads par exemple), soit explicitement (coefficient d'élasticité, quantité de données partagées, etc.)

Pour compenser les déséquilibres de charge éventuels, la méthode *idle* effectue un vol de travail. Pour cela, elle examine d'abord si des threads peuvent être volés directement sur les processeurs voisins[2]. S'il n'existe pas de tel thread, il faut réorganiser la répartition de bulles le long de la machine. La tactique utilisée est simplement de rassembler threads et bulles, et de relancer une distribution qui s'adapte alors à la nouvelle situation. Pour éviter des surcoûts prohibitifs, cette réorganisation est faite de manière la plus locale possible, mais suffisamment large tout de même pour trouver assez de threads pour tous les processeurs

[2] « Voisins » au sens hiérarchique : deux processeurs peuvent ne pas être voisins même s'ils portent des numéros consécutifs, lorsque l'un est le dernier processeur d'un nœud NUMA et l'autre le premier processeur du nœud NUMA suivant par exemple.

considérés. La figure 3.11 montre le déroulement de l'ordonnanceur *Affinity* sur un exemple de bulle déséquilibrée.

3.2.6 *MemAware*, un ordonnancement dirigé par la mémoire

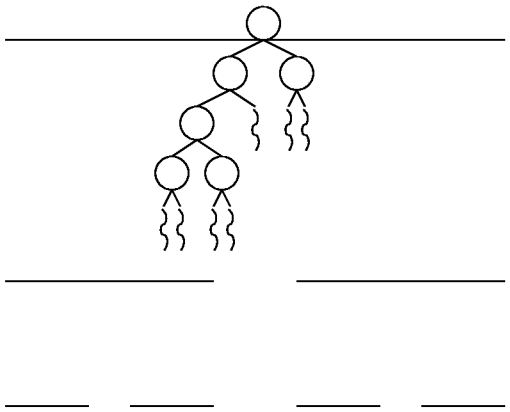
Nous avons vu à la section 1.2.1 (page 8) que sur les machines NUMA il était essentiel de prendre garde au placement des données en mémoire, or les ordonnanceurs décrits ci-dessus ne le font pas réellement. Pendant son stage de Master 2 [Jeu07] au sein de l'équipe dans le cadre du projet ANR NUMASIS, Sylvain JEULAND a donc affiné les ordonnanceurs *Spread* et *Steal* pour qu'ils prennent ce critère en compte lorsqu'ils choisissent le placement des threads sur les différents nœuds NUMA.

Il a pour cela travaillé avec l'équipe-projet Mescal de Grenoble sur l'interface de leur bibliothèque de gestion fine de tas, pour qu'elle lui permette notamment de savoir où les données sont effectivement allouées et d'enregistrer des informations fournies par l'application. Celles-ci peuvent être extraites de compteurs matériels ou bien déterminées par profilage logiciel, telles que la fréquence estimée d'accès, les taux de défauts de cache, etc. (voir section 1.4.4 page 22). Cette bibliothèque permet également de fusionner plusieurs tas, migrer tout ou partie des données entre nœuds NUMA, attacher artificiellement à la gestion des tas des zones déjà allouées par ailleurs (définitions statiques par exemple), etc.

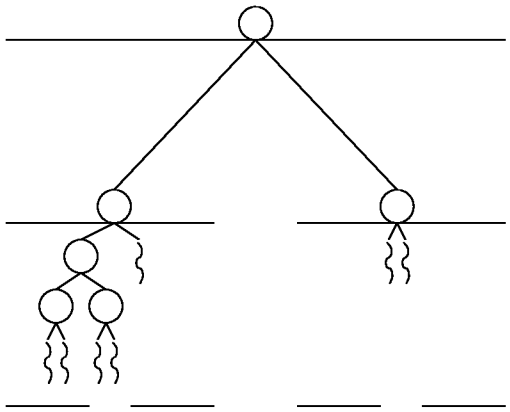
Le principe est alors de créer un tas pour chaque thread et chaque bulle. Lorsque l'application effectue une allocation de données, elle indique (ou bien le compilateur détermine automatiquement) quels threads accéderont le plus à ces données : le thread qui effectue l'allocation, un autre thread, ou bien l'ensemble des threads contenus dans une bulle donnée ; elle peut également indiquer une fréquence d'accès aux données. L'allocation est alors effectuée dans le tas correspondant au thread ou à la bulle indiquée. Ceci permet ainsi d'avoir à tout moment une connaissance hiérarchisée des affinités entre threads, données et nœuds NUMA.

Lors d'une distribution d'une hiérarchie de bulles, la version *MemAware* de l'ordonnanceur *Spread* prend garde à l'encombrement mémoire d'une bulle. Pour choisir quelle bulle éclater, elle considère la quantité de données qui a été allouée pour cette bulle et la fréquence d'accès (donc la quantité de données partagées correspondant à cette bulle). Elle préfère alors éclater les bulles pour lesquelles il y a le moins de données partagées (les moins « épaisses »), évitant ainsi de séparer les threads qui travailleront souvent sur les mêmes données.

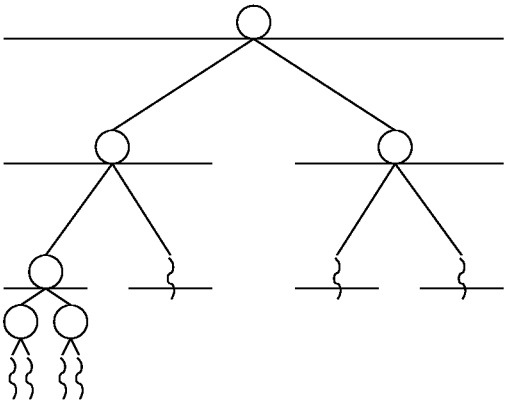
D'autre part, lors d'une redistribution d'une hiérarchie de bulles (pour s'adapter à une nouvelle situation de charge par exemple), le choix du placement des threads et bulles est guidé par les données effectivement allouées sur les nœuds NUMA. Pour chacun d'entre eux, on détermine en effet le *bassin d'attraction* : le nœud NUMA où se trouvent la plupart de ses données. Lors du déroulement de la distribution gloutonne, on privilégie alors les placements sur les bassins d'attraction. Lorsque c'est contre-indiqué pour des raisons de répartition de charge, on fait à l'inverse migrer les données sur le nouveau nœud NUMA choisi, adaptant ainsi le placement des données en fonction de la nouvelle répartition de charge. Le choix entre placement dans le bassin d'attraction existant ou déplacement (coûteux) des données pour pouvoir changer le bassin d'attraction fait bien sûr l'objet d'heuristiques qui prennent en compte les caractéristiques de la machine et dépendent de l'application. D'autre part, une



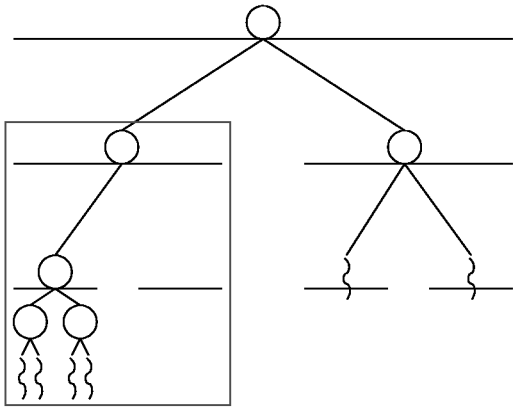
(a) Situation de départ.



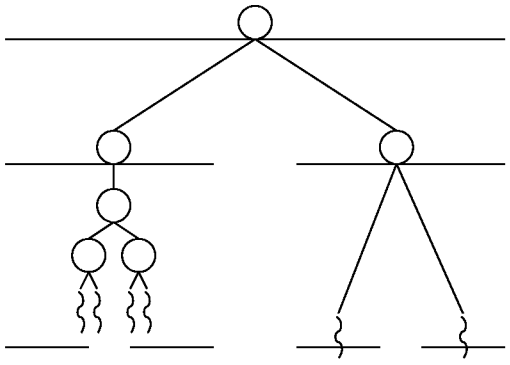
(b) Il faut éclater la bulle principale pour pouvoir alimenter les deux sous-listes.



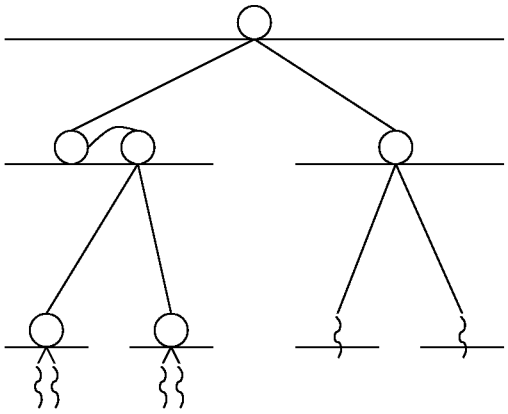
(c) Il faut éclater les deux sous-bulles pour pouvoir alimenter tous les processeurs. L'algorithme est terminé, malgré un déséquilibre en nombre de threads.



(d) Un thread s'est terminé, un processeur devient inactif et il n'y a pas de thread directement accessible chez le processeur voisin, l'algorithme de vol considère donc la liste juste supérieure et ses sous-listes.



(e) Les entités de cette partie de machine sont rassemblées.



(f) L'algorithme de distribution est relancé sur cette partie de machine, il est alors obligé de percer deux bulles. On obtient une nouvelle distribution adaptée à la nouvelle charge.

FIG. 3.11 – Déroulement de l'ordonnancement *Affinity*.

fois que la distribution est descendue en-dessous du « niveau » NUMA, l'algorithme essaie le plus possible de rejouer la même distribution que précédemment pour privilégier la réutilisation de cache. Il est envisagé d'utiliser des modélisations de vieillissement de cache pour affiner cette stratégie.

Enfin, la version *MemAware* de l'ordonnanceur *Steal* prend en compte le coût que peut avoir un vol de travail. En effet, lors du choix des entités à voler (bulles comme threads), il privilégie celles qui sont les moins « lourdes », c'est-à-dire qui ont leurs données sur le nœud NUMA du processeur inactif, ou qui ont le moins possible de données allouées, ou qui y accèdent peu souvent. De plus, lorsque l'ordonnanceur se permet de déplacer une entité entre différents nœuds NUMA, il migre également les données les plus accédées pour ne pas trop briser les affinités entre cette entité et ses données. Selon les cas, il pourra éventuellement se permettre de prendre le temps de migrer aussi les données moyennement accédées.

3.2.7 Discussion

Cette section a permis de présenter différents algorithmes pour exploiter les machines hiérarchiques par différentes approches, parfois complètement opposées : l'algorithme *Spread* se focalise par exemple sur la répartition de charge tandis que l'algorithme *Affinity* privilégie plutôt les affinités au détriment de la répartition de charge (dont on ne dispose de toutes façons pas nécessairement une estimation). Bien souvent, il reste bien sûr des questions ouvertes, des réglages possibles, des heuristiques à inventer. D'autre part, il est *a priori* utile de combiner différentes approches pour obtenir un algorithme *raisonnable*. Les quelques algorithmes présentés dans cette section peuvent par exemple servir de briques de base pour des algorithmes plus évolués (tout comme les algorithmes *Spread* et *Steal* ont servi de base pour les algorithmes *MemAware*). On s'aperçoit ici que ces considérations sont de l'ordre purement algorithmique, les aspects techniques d'implémentation sous-jacents ne posent pas problème. Cela montre combien l'interface de programmation fournie, bien que relativement simple, a permis d'abstraire la problématique en fournissant un outil de haut niveau pour implémenter différentes stratégies d'ordonnancement. Cependant, pour pouvoir expérimenter ces réglages et heuristiques, il est nécessaire de disposer d'un environnement d'expérimentation adéquat, ce qui est l'objet de la section suivante.

D'un point de vue technique, il reste cependant quelques interrogations sur l'interface fournie, notamment sur les questions de verrouillage : respecter les conventions de verrouillage décrites plus loin à la section 4.2.3 (page 68) pourrait apparaître difficile, et l'on est alors vite tenté d'effectuer un simple verrouillage global, ce qui peut être très pénalisant en termes de performances s'il est effectué souvent. On pourrait par exemple fournir des itérateurs sur bulles qui effectuent le verrouillage de manière appropriée et appellent une fonction fournie par le programmeur d'ordonnanceur. Pour simplifier le verrouillage, il serait aussi possible d'implémenter un mécanisme RCU (*Read-Copy Update*, [MS98]) au sein de MARCEL.

Par ailleurs, l'écriture d'un ordonnanceur en langage C peut parfois se révéler pénible lorsqu'il s'agit de gérer des ensembles d'objets triés selon un attribut, itérer au sein d'une topologie complexe, prendre des décisions avancées, etc. Il pourrait être intéressant d'embarquer par exemple un interpréteur pour un langage de plus haut niveau pour permettre l'écriture d'un ordonnanceur dans ce langage, permettant ainsi de s'abstraire complètement des contraintes techniques de l'implémentation en C.

3.3 Un environnement pour expérimenter

Les différents ordonnanceurs décrits à la section précédente sont relativement simples. Ils peuvent en fait servir de briques de base pour développer des ordonnanceurs plus complets, en les combinant de manière variée, et en les réglant pour fonctionner avec telle ou telle application. Notre plate-forme propose alors des outils pour observer le comportement de tels ordonnanceurs et comprendre où se situent les pertes de performances.

3.3.1 Combiner des ordonnanceurs

Par la nature même d'un ordonnanceur (un ensemble de *méthodes*), il est assez facile de combiner par exemple l'ordonnanceur *Spread* et l'ordonnanceur *Steal*, pour obtenir un ordonnanceur qui commence par effectuer une distribution des bulles et threads sur la machine à l'aide de *Spread*, puis utilise la méthode *idle* de *Steal* pour corriger les déséquilibres éventuels.

Il est cependant aussi possible de combiner les ordonnanceurs de manière *spatiale*. En effet, lorsqu'une application est par exemple composée d'un couplage de deux codes, Nikolopoulos *et al.* montrent [NPP98] qu'il vaut mieux partager la machine en deux parties, chacune exécutant l'un des deux codes. Si ces deux codes ont des comportements différents, il sera alors utile d'essayer d'utiliser des ordonnanceurs différents sur les deux parties de machine.

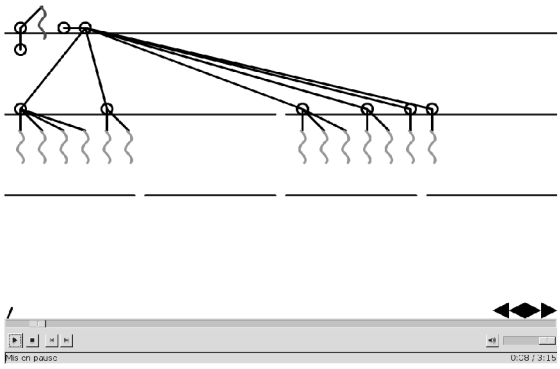
Il est aussi envisageable de combiner des ordonnanceurs de manière *hiérarchisée*. On peut par exemple utiliser l'ordonnanceur *Gang* comme ordonnanceur *primaire*, et pour chaque bulle déposée sur la machine (ou partie de machine) par l'ordonnanceur *Gang*, on utilise l'ordonnanceur *Spread* pour répartir son contenu sur la machine. Il se pourrait même qu'un micro-ordonnancement de *gangs* soit de nouveau utile au niveau de processeurs hyperthreadés pour co-ordonnancer des paires de threads dont on sait qu'ils s'exécutent bien ensemble.

Enfin, lorsqu'une application est décomposée en plusieurs phases, il se peut que celles-ci se comportent de manière assez différente. Il sera alors utile de changer d'ordonnanceur au cours du *temps*. Dans le cas de l'application MPU présentée à la section 6.2.4 (page 97), une première phase très irrégulière nécessitera l'emploi de l'ordonnanceur *Affinity*, tandis que la deuxième phase, très régulière et très indépendante, ne nécessitera pas du tout d'ordonnanceur, une simple répartition (même aléatoire !) sur les processeurs étant suffisante.

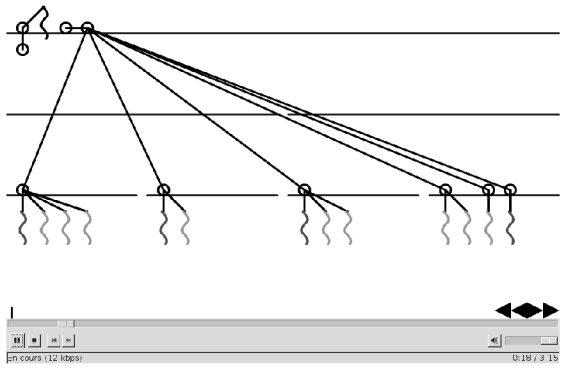
3.3.2 Comprendre les performances

A priori, il est difficile de comprendre pourquoi les performances obtenues par une application ne sont pas aussi bonnes que ce qui était espéré. Sans outils adaptés, la seule valeur facilement mesurable est le temps d'exécution qui permet certes de vérifier si l'on obtient effectivement un *speedup*, mais qui n'indique pas du tout pourquoi. C'est pourquoi il est très courant de fournir des outils d'analyse d'exécution de programme, des compteurs de performances des processeurs, etc. Ces outils ne sont évidemment pas adaptés à l'observation du comportement d'ordonnanceurs à bulles, et notre plate-forme fournit donc son propre outil d'observation.

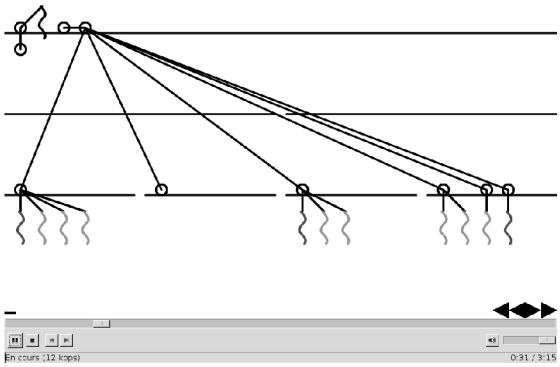
Lors de l'exécution de l'application, une trace de tous les évènements d'ordonnancement



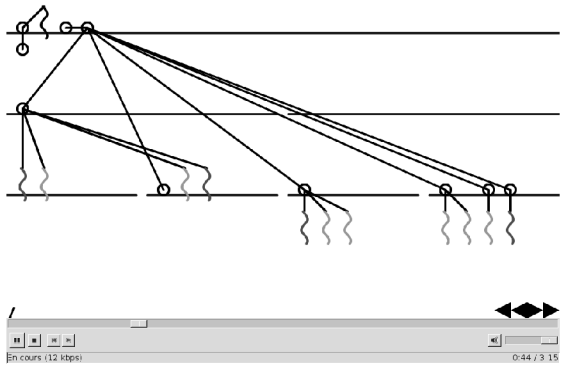
(a) Répartition en cours.



(b) Répartition terminée.



(c) Deux threads terminent, un processeur devient inactif.



(d) L'algorithme *Spread* a été relancé localement pour combler le déséquilibre.

FIG. 3.12 – *Action Replay* de l'ordonnancement *Spread*.

peut être enregistrée de manière légère [DW05] : création, endormissement, destruction des threads, structuration en arbre des bulles, répartition le long de la machine, etc. Après l'exécution, il est alors possible d'analyser cette trace pour observer de manière fine ce qui s'est passé. Pour cela, elle est convertie en animation *flash* en s'appuyant sur la bibliothèque MING [min]. La figure 3.12 montre par exemple l'évolution de l'ordonnanceur *Spread* en réaction à l'inactivité d'un processeur.

Il est ainsi possible de vérifier rapidement et précisément ce qui s'est passé au cours de l'exécution, ce qui est précieux pour *déboguer* les ordonnanceurs. Bien sûr, la barre de défilement des lecteurs *flash* permet de se déplacer rapidement au sein de l'animation pour atteindre les périodes intéressantes. Il est par ailleurs possible d'ajouter toute information utile, telle que les attributs de threads (noms, priorité, charge, mémoire allouée, etc.) ou des bulles (élasticité, mémoire allouée, etc.) pour vérifier facilement le comportement de l'algorithme face à ces informations.

Cet outil est également très utile à des fins de démonstration : pour expliquer de manière visuelle un algorithme à un collègue ou au sein de présentations lors de colloques par exemple.

3.3.3 Multiplier les tests

Une fois un ordonnanceur développé, il est utile de vérifier qu'il se comporte bien quelle que soit la hiérarchie de bulles considérée. Pour pouvoir expérimenter rapidement de nombreux cas typiques de hiérarchies de bulles, notre plate-forme fournit un outil de génération de bulles appelé *BubbleGum*, dont le développement a été confié à un groupe d'étudiants. La figure 3.13 montre l'interface graphique de cet outil. La partie de gauche permet de construire récursivement des hiérarchies de bulles de manière intuitive par sélection et raccourcis claviers ou clics sur les boutons. Quelques attributs tels que la charge, la priorité ou le nom d'un thread peuvent être spécifiés en même temps que la construction. De plus, la charge est rendue visuellement en changeant les couleurs des threads. Il est possible de corriger les relations en effectuant une sélection puis un simple glisser-lâcher entre bulles. Un clic sur un bouton permet enfin de lancer la génération d'un programme C synthétique, son exécution, et la récupération de la trace, qui est ici encore convertie en une animation, intégrée cette fois à l'interface sur la partie droite. Cela permet notamment de sélectionner threads et bulles à partir de leur placement effectif sur la machine.

Bien sûr, il est possible de *sauvegarder* une hiérarchie de bulles pour pouvoir la reprendre plus tard, ou bien l'intégrer dans une autre hiérarchie. On peut ainsi se constituer un panel de hiérarchies de bulles synthétiques de toutes sortes : régulières, déséquilibrées, irrégulières, voire pathologiques. Cela permet également de s'échanger entre développeurs certains cas pathologiques par exemple.

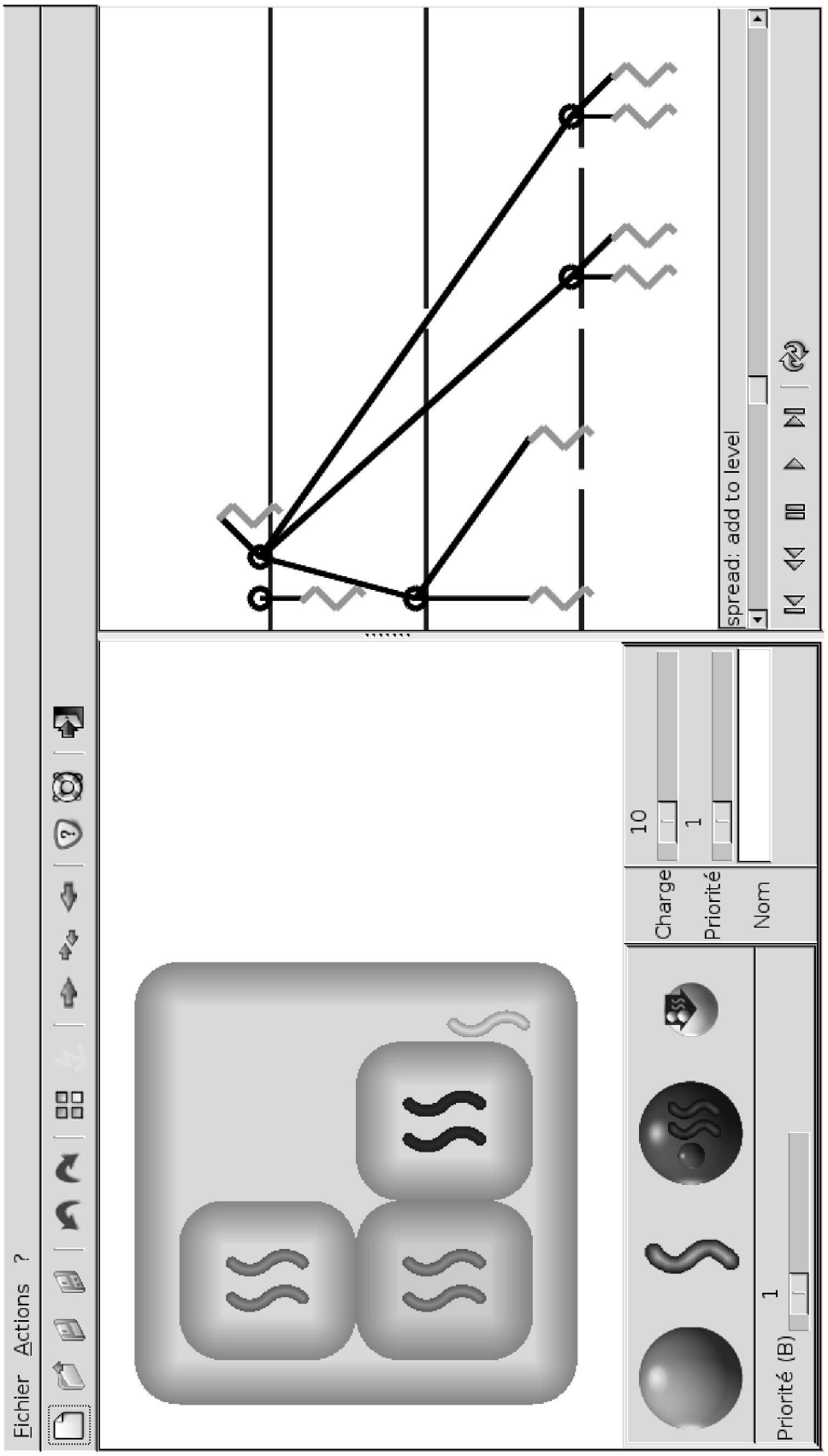


FIG. 3.13 – Interface graphique de génération de bulles : *BubbleGum*.

Chapitre 4

Éléments d'implémentation

Sommaire

4.1 Sur les épaules de MARCEL 63
4.2 Ordonnancement 66
4.2.1 Ordonnanceur de base 66
4.2.2 Mémoriser le placement 67
4.2.3 Verrouillage 68
4.3 Portabilité 68
4.3.1 Systèmes d'exploitation : découverte de topologie 68
4.3.2 Architectures : opérations de synchronisation 71
4.4 Optimisations 72
4.4.1 Parcours des bulles à la recherche du prochain thread à exécuter . . 73
4.4.2 Plus léger que les processus légers 73
4.4.3 Distribution des allocations 75

Dans ce chapitre, nous présentons l'implémentation en elle-même de la plate-forme BubbleSched. Nous expliquons d'abord pourquoi et comment elle a été réalisée au sein de MARCEL, la bibliothèque de threads de l'environnement d'exécution PM^2. Nous détaillons ensuite quelques points d'implémentation intéressants d'un point de vue méthodologie : de l'ordonnancement de haut niveau d'abord, puis des optimisations nécessaires à une exécution efficace.

4.1 Sur les épaules de MARCEL

La bibliothèque de threads utilisateur MARCEL a été originellement développée par Jean-François MÉHAUT et Raymond NAMYST pour l'environnement distribué de programmation multithreadée PM^2 [Nam97]. Cet environnement combine en effet MARCEL avec une bibliothèque de communication, MADELEINE, l'objectif étant de coupler de manière efficace calcul parallèle et communications, en utilisant notamment des threads pour faire progresser les communications de manière asynchrone et réactive. La librairie MARCEL fonctionne entièrement en espace utilisateur à l'aide d'appels à `setjmp` et `longjmp` pour changer de

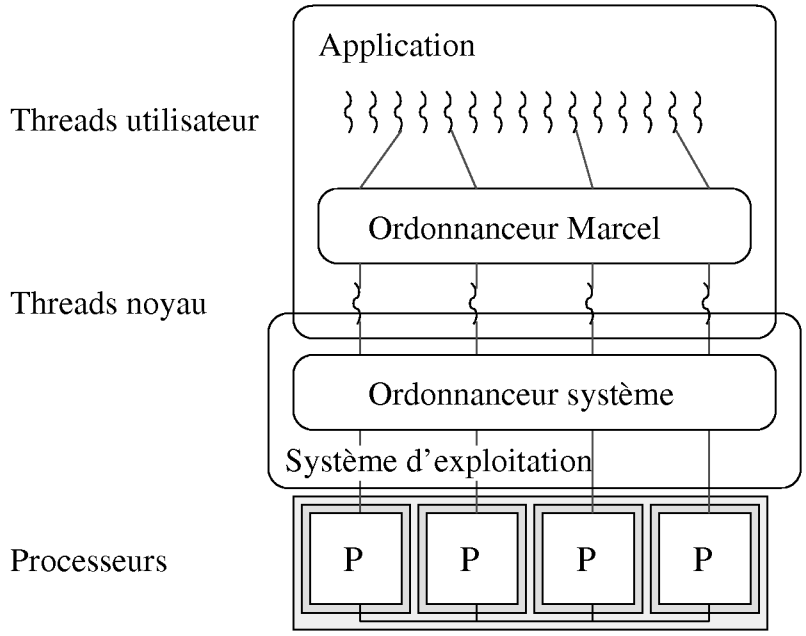


FIG. 4.1 – Mode de fonctionnement de base de Marcel : l'ordonnanceur du système est en fait court-circuité.

contexte entre les threads MARCEL, ce qui lui permet d'être très légère en comparaison de toute bibliothèque de threads basée sur des threads de niveau noyau, ainsi que d'être utilisée sans avoir à changer le noyau du système. Son portage sur de nombreux systèmes (notamment GNU/Linux, BSD, AIX, Irix et OSF) et architectures (notamment x86, x86_64, Itanium, PowerPC, Alpha, Mips et Sparc) n'a par ailleurs nécessité que l'adaptation de quelques fonctions de bas niveau. À l'origine, MARCEL ne supportait que les machines monoprocesseurs. Vincent DANJEAN a par la suite étendu MARCEL aux machines multiprocesseurs en introduisant un ordonnanceur mixte [Dan98] qui exécute tour à tour les threads utilisateur sur autant de threads noyau qu'il y a de processeurs. De plus, lorsque le système d'exploitation le permet (ce qui est le plus souvent le cas), MARCEL lui demande de fixer les threads de niveau noyau sur les processeurs, ce qui lui permet alors de réellement contrôler l'exécution précise des threads sur les processeurs sans aucune interaction ultérieure avec le système (en supposant qu'aucune autre application ne s'exécute sur la machine). Ce mode de fonctionnement est représenté sur la figure 4.1. Vincent DANJEAN a par ailleurs implémenté le mécanisme de *Scheduler Activations* [DNR00a] pour que les threads Marcel puissent effectuer des appels système bloquants sans pour autant limiter l'exploitation de tous les processeurs de la machine. Enfin, il a intégré à MARCEL une des premières versions stables de l'ordonnanceur en $O(1)$ de LINUX 2.6 avec les outils associés (listes, *spinlocks*, *runqueues*, *tasklets*, etc.) [Dan04].

Il était ainsi naturel de poursuivre le développement de MARCEL pour lui permettre d'exploiter au mieux les machines NUMA. Il a d'abord fallu consolider le support SMP existant pour qu'il puisse exploiter de nombreux processeurs, en distribuant par exemple certains mécanismes qui étaient encore centralisés ; quelques points sont donnés en exemple à la section 4.4. Il a ensuite été possible d'implémenter l'ordonnancement à bulles en modifiant

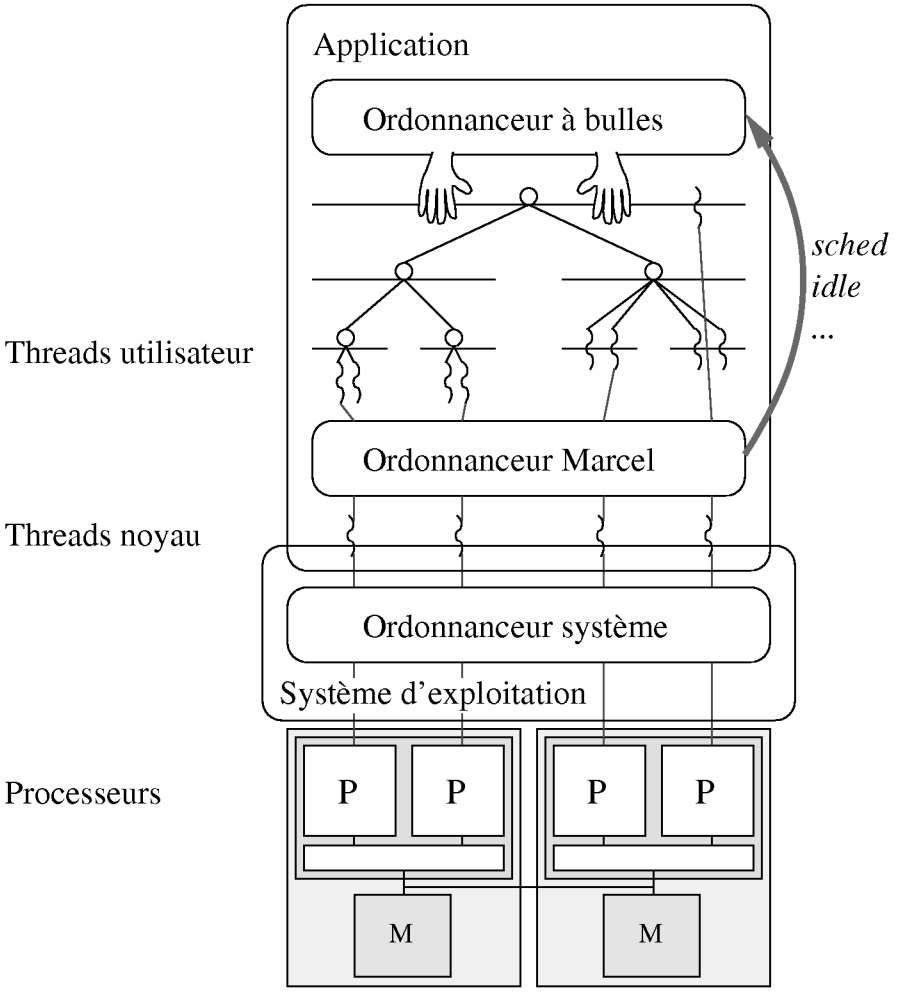


FIG. 4.2 – Mode de fonctionnement de Marcel avec bulles : l'ordonnanceur de base de Marcel est conditionné par le placement des threads et bulles.

l'ordonnanceur de base. Celui-ci n'était au départ prévu que pour consulter une liste locale au processeur, avec une répartition de charge ne prenant en compte ni affinités entre threads ni topologie de la machine sous-jacente. Le nouveau mode de fonctionnement est représenté figure 4.2. L'ordonnanceur de base consulte désormais une hiérarchie de listes de threads et de bulles, automatiquement construite à partir des informations de topologie fournies par le système d'exploitation. Par ailleurs, il appelle aux moments appropriés les méthodes de l'ordonnanceur à bulles courant, qui peut alors prendre l'initiative de déplacer threads et bulles sur la hiérarchie de listes. Ainsi MARCEL délègue la responsabilité de la répartition des threads à l'ordonnanceur à bulles, et se contente de suivre les contraintes imposées par les placements effectués sur la hiérarchie de listes.

Il serait bien sûr possible de réaliser une implémentation à l'aide d'approches micro-noyau telles qu'ExoKernel [EKO95] ou Nemesis [RF97] ou d'une approche *Scheduler Activations* telle que K42 [AAD+02], car elles permettent de réaliser des ordonnanceurs en espace utilisateur. Cependant, de telles approches nécessitent d'amorcer la machine utilisée avec ces

noyaux, or c'est en général impossible, par manque de support du matériel par ces noyaux, ou plus simplement pour des raisons administratives ou de sécurité. De fait, il est rare que l'administrateur de machines de calcul accepte de démarrer un autre système que celui qu'il administre[1]. L'approche que suit MARCEL permet par un petit effort de portage d'essayer différentes machines, indépendamment du système d'exploitation.

À plus long terme, il serait par contre intéressant d'essayer d'intégrer un ordonnanceur à bulles générique (ou plusieurs interchangeables) au noyau LINUX par exemple, en utilisant au moins les informations disponibles implicitement telles que l'imbrication de threads en processus, eux-mêmes regroupés en groupes et en sessions, etc. Il pourrait alors être utile d'étendre l'interface de threads POSIX pour permettre aux applications de spécifier au moins les affinités entre threads d'une manière analogue à nos bulles, pour permettre un ordonnancement plus fin. Une telle approche ne pourra pas atteindre les performances d'une approche complètement en espace utilisateur, car il n'est pas aussi aisé d'opérer des échanges d'informations entre l'ordonnanceur et l'application lorsque ceux-ci sont séparés par la barrière entre espace utilisateur et espace noyau. Cette approche devrait cependant permettre d'obtenir des gains appréciables. Nous reviendrons sur se point dans la section 7.2 qui développe les principales perspectives que nous dégageons de nos travaux.

4.2 Ordonnancement

L'implémentation des concepts théoriques décrits dans les chapitres précédents a parfois nécessité des choix intéressants que cette section développe.

4.2.1 Ordonnanceur de base

À la section 2.2 (page 33), nous avons vu que l'ordonnanceur de base doit parcourir une hiérarchie de listes de threads et bulles pour trouver le prochain thread à exécuter. Il doit par ailleurs respecter les priorités indiquées par l'application. La figure 4.3 montre un exemple où un thread de priorité 2 a été laissé sur la liste principale, un thread de communication par exemple. Dans un tel cas, dès que ce thread est prêt à être exécuté, un des processeurs doit l'exécuter à la place des threads de moindre priorité, même si ceux-ci sont sur des listes plus proches des processeurs.

La structure de liste utilisée (les *runqueues* de l'ordonnanceur $O(1)$ de LINUX 2.6) permet de réaliser cela de manière assez efficace. En effet, ces listes sont *hachées* par priorité, ce qui permet d'extraire en $O(1)$ une entité (thread ou bulle) d'une priorité donnée. De plus, chacune mémorise en permanence dans un champ de bits les priorités qu'ont les entités qu'elle contient, ce qui permet de trouver rapidement[2] la priorité maximale parmi celles des entités contenues dans cette liste.

[1] Une exception notable est la plate-forme Grid5000, qui cependant ne comporte que des machines avec au plus 4 processeurs.

[2] L'ordonnanceur $O(1)$ de LINUX 2.6 n'est en réalité pas tout à fait en $O(1)$, mais en O(nombre de priorités), où le nombre de priorités est actuellement fixé à 140 pour pouvoir supporter une priorité absolue pour les threads noyau, 99 priorités temps réel, et des indices de politesse (`nice`) entre -20 et 19. En pratique, la recherche dans le champ de bits est écrite en assembleur pour qu'elle soit la plus optimisée possible. Dans le cas de MARCEL, le nombre de priorités est configurable à la compilation et vaut par défaut 23, on conservera l'approximation $O(1)$.

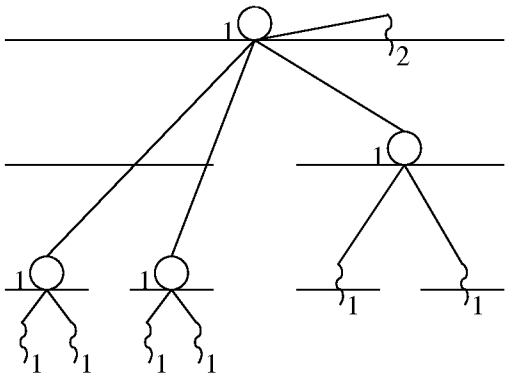


FIG. 4.3 – Distribution de bulles et threads avec priorités.

La recherche du prochain thread à exécuter utilise une méthode classique qui consiste à observer sans verrouillage, puis verrouiller et vérifier que l'observation est restée correcte. Ainsi, la recherche est effectuée en deux temps. La première se contente de parcourir les listes qui « couvrent » le processeur courant à la recherche de la liste la plus proche du processeur dont le champ de bits indique une entité de priorité la plus forte. On verrouille alors cette liste et l'on vérifie que la priorité maximale des entités contenues est au moins aussi grande que ce qui avait été observé avant verrouillage. On peut alors prélever une entité de cette priorité, exécuter la méthode *sched* de l'ordonnanceur à bulles si cette entité est une bulle, sinon (c'est un thread) déverrouiller la liste, basculer l'exécution sur ce thread, et déposer le thread précédent sur sa propre liste (avec verrouillage).

Au final, la complexité de cet ordonnanceur de base est donc $O(l)$, où l est la profondeur de la hiérarchie de la machine, qui est intrinsèquement limitée par le nombre de processeurs. Cette complexité est donc bornée par $O(log(P))$ où P est le nombre de processeurs.

4.2.2 Mémoriser le placement

La gestion du placement des bulles et threads sur les listes est un problème délicat dans certains cas, notamment lorsque l'on veut déplacer d'une liste à une autre un thread qui est en cours d'exécution sur un autre processeur, et que ce thread est contenu dans une bulle. Il est ainsi nécessaire de mémoriser trois *conteneurs* (liste ou bulle) pour chaque entité. Le conteneur *initial* est celui que le programmeur a défini dans son programme. Comme nous l'avons vu, il peut changer dynamiquement au cours de l'exécution lorsqu'il y a besoin de réorganiser la hiérarchie de bulles par exemple. Le conteneur d'*ordonnancement* est celui qui a été choisi par l'ordonnanceur à bulles, c'est-à-dire celui depuis lequel l'entité devrait être exécuté. Ce n'est pas nécessairement encore le cas, si elle est encore en cours d'exécution sur un autre processeur, mais lors de la prochaine exécution de l'entité, ce sera le cas. Enfin, le conteneur d'*exécution* est celui depuis lequel l'entité s'exécute effectivement, et pour lequel elle contribue aux statistiques. Ainsi, lorsqu'un ordonnanceur à bulles déplace un thread d'une liste à une autre alors qu'il est en cours d'exécution sur un autre processeur, seul le conteneur d'ordonnancement change, et il s'ensuit une période durant laquelle le conteneur d'exécution n'est pas « à jour ». Cette période dure jusqu'au prochain réordonnancement sur ce processeur, qui s'occupera alors de mettre à jour le conteneur d'exécution en fonction

du conteneur d'ordonnancement. Si l'ordonnanceur à bulles a besoin que la période transitoire soit la plus courte possible (lors d'un *Gang Scheduling* par exemple), il suffit d'appeler la fonction `ma_resched_task()` qui envoie un signal Unix au processeur concerné pour forcer une préemption immédiate.

4.2.3 Verrouillage

Le verrouillage des conteneurs doit être réalisé avec un soin particulier : plus la machine utilisée comporte de processeurs, plus il devient indispensable de bien distribuer le verrouillage. Chaque conteneur comporte ainsi un verrou qui protège l'accès à la liste des entités contenues. Il est alors essentiel de définir un ordre de verrouillage des différents conteneurs pour éviter tout inter-blocage. La convention qui a été choisie est de toujours verrouiller du haut en bas et par ordre de numérotation ou de position dans les listes croissante, un parcours en profondeur donc. Pour verrouiller tout ce qui se trouve sur un cœur par exemple, il faut commencer par verrouiller la liste du cœur, puis la première bulle de cette liste, puis ses sous-bulles par un parcours en profondeur, puis la deuxième bulle de cette liste, puis son contenu, etc. puis le même parcours sur les listes des processeurs virtuels du cœur. Cette convention est venue assez naturellement par les nombreux parcours des bulles du haut vers le bas. Dans certaines situations, il est cependant nécessaire de libérer un verrou pour pouvoir en prendre un autre, mais dans l'implémentation actuelle cela ne survient qu'à deux endroits, tandis que d'autres conventions auraient apporté un plus grand nombre de telles situations.

4.3 Portabilité

Une des caractéristiques de la bibliothèque MARCEL est sa *portabilité*, au sens où il est très aisé de la porter sur différents systèmes d'exploitation et architectures. Un des objectifs de l'implémentation de la plate-forme BubbleSched était donc de conserver cette portabilité. Nous voyons dans cette section quelques points intéressants.

4.3.1 Systèmes d'exploitation : découverte de topologie

La bibliothèque MARCEL dépend en fait très peu du système d'exploitation utilisé, pour peu que celui-ci fournisse une interface POSIX basique. En effet, MARCEL utilise la fonction `pthread_create` pour créer les threads noyau, puis les fonctions `setjmp` et `longjmp` pour basculer d'un thread utilisateur à un autre (la création de thread dépend essentiellement du processeur, voir section suivante). Quelques détails de compilation diffèrent, tels que les options du compilateur, mais ils sont très limités.

La découverte de topologie, par contre, n'est pas standardisée, même l'obtention du nombre de processeurs varie. De nombreux systèmes proposent pour la fonction POSIX `sysconf()` un paramètre non POSIX `_SC_NPROCESSORS_ONLN` qui lui fait retourner le nombre de processeurs disponibles. D'autres systèmes fournissent plutôt un paramètre `_SC_NPROC_CONF`, d'autres encore tels que DARWIN fournissent une fonction complètement non standard. On encapsule donc cette fonctionnalité dans une simple fonction `int ma_nbprocessors()`

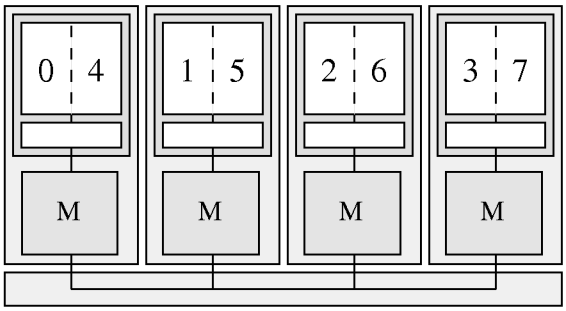


FIG. 4.4 – Exemple de machine contenant deux nœuds NUMA comportant chacun un cœur hyperthreadé, avec une numérotation non consécutive (voir section 1.2.4 page15).

que l'on doit éventuellement réécrire pour un nouveau système d'exploitation. La fixation d'un thread noyau sur un processeur particulier nécessite par ailleurs l'emploi d'une fonction propre à chaque système, mais là aussi il est facile d'encapsuler cette fonctionnalité dans une fonction `void ma_bind_on_processor(int target)`.

La découverte des relations entre processeurs n'est absolument pas standardisée, chaque système possède sa propre interface complètement différente des autres, fournissant plus ou moins de détails de manières très variées. Il a donc été nécessaire, pour faciliter les portages, de trouver un dénominateur commun qui soit facile à implémenter sur chaque système, mais qui suffise cependant aux besoins de notre plate-forme. Ce dénominateur commun s'est révélé relativement simple, mais au prix d'une analyse générique complexe, si bien que le module de gestion de topologie est le troisième plus gros module de MARCEL, après l'ordonnanceur de base et la gestion des signaux Unix !

Le principe est que pour un système d'exploitation donné, une fonction `look_topology` décrit les niveaux de hiérarchie de la machine, des plus globaux aux plus locaux[3]. Cette description comporte simplement le type de hiérarchie (nœud NUMA, puce, partage de cache, cœur ou processeur logique) et l'ensemble des processeurs concernés par ce niveau (par un masque de bits). Sur le cas de la figure 4.4, `look_topology` pourra par exemple d'abord indiquer qu'il existe quatre nœuds NUMA ayant respectivement pour masque de processeurs `0x11`, `0x22`, `0x44` et `0x88`, puis qu'il existe quatre cœurs ayant respectivement pour masque de processeurs `0x11`, `0x22`, `0x44` et `0x88`, et enfin qu'il existe huit processeurs logiques ayant respectivement pour masque de processeurs `0x01`, `0x02`, `0x04`, `0x08`, `0x10`, `0x20`, `0x40` et `0x80`. L'implémentation effective varie selon les systèmes : certains tels qu'OSF ne peuvent fournir d'information que pour les nœuds NUMA, c'est alors trivial. D'autres tels qu'AIX permettent d'utiliser une boucle `for` pour itérer la découverte sur les différents types de niveaux de topologie. Enfin, certains systèmes tels que LINUX nécessitent l'analyse d'un fichier texte.

Une fois ce résultat brut obtenu, il s'agit de trier et filtrer ces informations. La figure 4.5 illustre le déroulement de cette analyse. Une première étape éventuelle, non nécessaire ici, s'occupe de trier les niveaux par masque de processeur pour obtenir une numérotation finale cohérente. Une deuxième étape rassemble pour chaque niveau les sous-niveaux qui

[3] Il serait même possible de lever cette contrainte en effectuant automatiquement un tri topologique, mais en pratique elle s'est toujours révélée facile à respecter.

Machine :	0xff							
Nœuds NUMA :	0x11	0x22	0x44	0x88				
Cœurs :	0x11	0x22	0x44	0x88				
Processeurs logiques :	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80

(a) Informations brutes fournies par le système d'exploitation.

Machine :	0xff							
Nœuds NUMA :	0x11		0x22		0x44		0x88	
Cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(b) Sous-niveaux rassemblés selon le niveau supérieur.

Machine :	0xff							
Nœuds NUMA + cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(c) Niveaux redondants fusionnés.

Machine :	0xff							
Artificiel :	0x33				0xcc			
Nœuds NUMA + cœurs :	0x11		0x22		0x44		0x88	
Processeurs logiques :	0x01	0x10	0x02	0x20	0x04	0x40	0x08	0x80

(d) Niveau intermédiaire artificiel ajouté.

FIG. 4.5 – Déroulement de l'analyse de topologie ; à chaque étape est indiqué pour chaque niveau le masque de processeurs de chaque élément.

le composent. Ce n'est ici pas nécessaire pour les cœurs, mais ça l'est pour les processeurs logiques. Une troisième étape fusionne les niveaux qui sont redondants : ici chaque nœud NUMA contient exactement un cœur. Enfin, une dernière étape éventuelle ajoute un niveau intermédiaire pour éviter une arité entre niveaux trop grande. Il est par ailleurs possible de choisir de n'utiliser qu'une partie de la machine en indiquant le numéro du premier processeur à utiliser, le nombre de processeurs à utiliser, et le « pas » (ici on peut indiquer par exemple 2 pour ignorer un processeur logique sur deux[4]). Une étape intermédiaire est alors insérée pour tronquer les niveaux non utiles et laisser la troisième étape simplifier éventuellement la topologie.

Enfin, pour choisir sur quel nœud NUMA une zone mémoire doit être allouée, ou pour migrer une zone mémoire sur un autre NUMA, le standard POSIX ne fournit de nouveau aucune interface. Il suffit cependant pour chaque système d'exploitation d'écrire des fonctions `void *ma_malloc_node(size_t size, int node)` et `void ma_migrate_mem(void *ptr, size_t size, int node)`.

[4]On pourrait imaginer des manières plus portables d'effectuer cette sélection, en indiquant par exemple qu'on ne veut utiliser qu'un processeur logique par cœur, ou bien quatre processeurs partageant le moins de bande passante possible, etc.

En résumé, pour porter la plate-forme BubbleSched sur un système d'exploitation, il suffit d'implémenter quelques fonctions assez simples : retourner le nombre de processeurs, fixer un thread noyau sur un processeur, décrire la topologie de manière basique, allouer une zone mémoire sur un nœud donné, migrer une zone mémoire sur un nœud donné.

4.3.2 Architectures : opérations de synchronisation

Le fonctionnement de MARCEL dépend assez peu du type de processeur utilisé sur la machine cible. La construction d'un nouveau contexte de thread nécessite simplement de réécrire pour chaque processeur une macro qui définit le cadre de pile, en modifiant par exemple les registres `sp` et `bp` sur l'architecture x86.

Cependant, un point de portage qui mérite attention est l'écriture des opérations de synchronisation. En effet, MARCEL nécessite pour son fonctionnement une série d'opérations qui ne peuvent pas s'écrire en C pur : opérations atomiques (incrémentation, décrémentation, addition, manipulations de bits, etc.), verrous rotatifs (simples, lecteur/rédacteur), *compare & exchange*, et barrières mémoire. Pour faciliter les portages, Vincent DANJEAN a choisi [Dan04] de réutiliser simplement l'implémentation LINUX de ces opérations, avec quelques adaptations un peu fastidieuses nécessaires à l'intégration dans MARCEL (découpage des en-têtes et renommage des fonctions, notamment). Dans certains cas (notamment les verrous rotatifs), l'implémentation LINUX utilise cependant des fonctionnalités propres à la chaîne de compilation GNU, les sections ELF notamment qui ne sont pas disponibles sur certains systèmes, et il faut alors modifier encore l'implémentation. Par ailleurs, les compilateurs GCC récents fournissent des fonctions *builtin* qui effectuent un certain nombre d'opérations atomiques utiles, mais toutes ne sont pas toujours disponibles selon les processeurs.

Ainsi, pour simplifier encore l'effort de portage sur un nouveau processeur, la bibliothèque MARCEL possède des versions génériques des différentes opérations atomiques et verrous, basées sur certaines opérations atomiques élémentaires parmi *test & set*, *compare & exchange* et les verrous rotatifs. La figure 4.6 montre le graphe des implémentations effectuées. Les implémentations en pointillés fins (*compare & exchange* et *test & set* à partir des verrous rotatifs) ne peuvent pas passer à l'échelle car elles nécessitent l'emploi d'un verrou centralisé. L'implémentation des opérations atomiques à partir des verrous rotatifs n'est pas très efficace car elle utilise un verrou par variable atomique. Cependant, ces implémentations sont suffisantes pour que tous les outils soient disponibles en implémentant pour la machine cible au moins l'une des trois opérations de base représentées par des rectangles. Ainsi, lors d'un portage de MARCEL sur une nouvelle architecture, on peut choisir la quantité d'effort fourni :

Aucun effort : Par défaut, MARCEL utilise les fonctions POSIX `pthread_mutex` ou `pthread_spin` pour implémenter les verrous rotatifs. Cela impose d'utiliser des implémentations peu efficaces et qui ne passent pas à l'échelle, mais l'avantage est que cela suffit sur n'importe quel système POSIX.

***Test & set* :** Certains processeurs ne fournissent comme opération atomique que le *test & set*. L'implémentation des variables atomiques nécessite alors un verrou rotatif associé à la valeur stockée. Par ailleurs, les opérations *compare & exchange* doivent prendre un verrou global, ce qui ne passe pas à l'échelle.

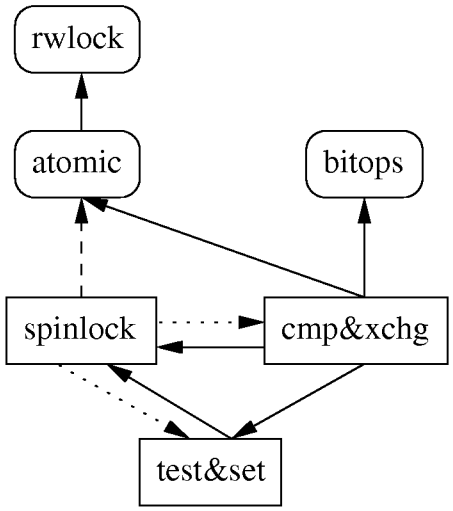


FIG. 4.6 – Implémentations des opérations atomiques et verrous rotatifs.

`pthread_mutex`	180ms
Test & Set	140ms
Compare & Exchange	135ms
Complet	130ms

TAB. 4.1 – Efficacité du programme *sumtime* selon les différents efforts de portage.

***Compare & exchange* :** La plupart des processeurs proposent l'opération *Compare & exchange* de manière native. Il est alors possible d'implémenter toutes les autres opérations de manière efficace.

Effort complet : Enfin, comme le propose Vincent DANJEAN, il est possible d'effectuer un portage natif complet de toutes les opérations pour les optimiser au maximum.

La table 4.1 montre la variation, en fonction de l'effort de portage, de l'efficacité du programme de test *sumtime* [Dan04] qui crée, synchronise et détruit récursivement de nombreux threads (voir section 6.1.2 page 90 pour plus de détails), ici sur une machine à deux processeurs x86. On s'aperçoit que porter seulement les opérations *Test & Set* ou *Compare & Exchange* suffit à obtenir des performances très proches d'un portage complet. Pour pouvoir passer à l'échelle, il faudra cependant absolument effectuer au moins un portage de *Compare & Exchange*.

Pour ce qui est des barrières de synchronisation mémoire, leur implémentation consiste en quelques lignes, et la réutilisation de celles du noyau LINUX ne pose pas de problème.

4.4 Optimisations

Comme souligné par Edler [Edl95], la qualité en elle-même d'une implémentation est essentielle pour obtenir de bonnes performances. Cette section détaille donc certaines optimisations qu'il a été utile de faire, et dont les applications peuvent ainsi profiter de manière

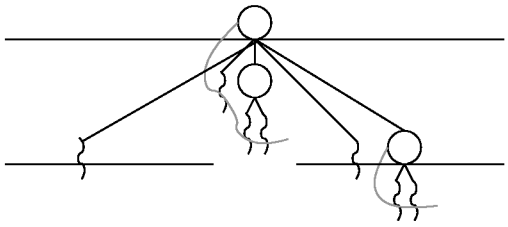


FIG. 4.7 – Illustration du cache de threads.

transparente.

4.4.1 Parcours des bulles à la recherche du prochain thread à exécuter

Lorsque l'ordonnanceur de base rencontre une bulle sur une liste, la méthode *sched* de l'ordonnanceur est appelée. L'implémentation par défaut parcourt cette bulle et ses sous-bulles à la recherche d'un thread à exécuter. La complexité de cette recherche n'est *a priori* pas bornée, puisque la hiérarchie de bulles peut être d'une taille arbitraire. Pour conserver un ordonnancement de base efficace, chaque bulle contient donc un *cache* de thread, illustré à la figure 4.7. C'est la liste des threads contenus dans cette bulle et ses sous-bulles qui ne sont pas placés sur d'autres listes. La recherche de thread peut alors s'effectuer en temps constant. La contrepartie est un surcoût en $O(1)$ de la mise à jour du cache lors des opérations sur les bulles et threads, mais ces opérations sont bien moins courantes que l'ordonnancement de base.

4.4.2 Plus léger que les processus légers

Lors de la parallélisation d'une application, une question qui revient souvent est « combien de threads créer ? ». En effet, si l'on crée peu de threads qui ont chacun une grande quantité de travail à faire, cela empêche une bonne répartition de charge puisque la granularité avec laquelle l'ordonnanceur pourra jouer est grande. À l'inverse, si l'on crée de nombreux threads qui ont chacun très peu de travail (dans certains cas, quelques microsecondes seulement), cela permet certes une bonne répartition de charge *a priori*, mais le coût de création d'un thread peut devenir prohibitif (typiquement quelques microsecondes, même pour les implémentations en espace utilisateur). Une approche typiquement utilisée est alors de gérer des « tâches » qui sont attribuées par l'application (ou par l'environnement d'exécution, dans le cas d'OPENMP par exemple) à un certain nombre de threads, nombre qui est choisi en fonction du rapport entre le coût de création d'un thread et le temps d'exécution d'une tâche. Cela ne permet cependant pas d'exprimer *tout* le parallélisme de l'application, ce qui est dommage. Engler *et al.* décrivent également ce problème [EAL93] et proposent d'alléger la notion de thread en une notion de *filament*, qui est une version très limitée de thread, spécialisée dans trois types de tâches particulières : s'exécuter sans interruption jusqu'à terminaison, itération avec barrière et *fork/join*. Ils sont exécutés par de véritables threads verrouillés sur chaque processeur. Cela permet d'obtenir une implémentation beaucoup plus simple et efficace, réduisant ainsi le coût de création presque à l'ordre de grandeur d'un appel de fonction. D'autres implémentations de bibliothèques de threads (MPC [Pér06] par

	Création	Exécution
Thread	$0,85\,\mu$s	$0,60\,\mu$s
Graine	$0,22\,\mu$s	$0,22\,\mu$s
NPTL	$6,4\,\mu$s	$11,3\,\mu$s

TAB. 4.2 – Coût de création et d'exécution d'un thread et d'une graine de thread. À titre de comparaison, valeurs pour la bibliothèque de threads native de Linux, la NPTL

exemple) utilisent aussi ce genre de simplification. Par ailleurs, la plupart des implémentations d'OPENMP utilisent ce genre de mécanisme pour effectuer les entrées et sorties de sections parallèles sans pour autant recréer à chaque fois tous les threads.

Pour répondre à ce besoin, la bibliothèque MARCEL utilise une approche qui obtient au final le même résultat, mais de manière légèrement différente. Elle dispose désormais de la notion de « graine de thread » : lors de la création d'un thread, on peut indiquer à l'aide d'un attribut que l'on ne veut pas nécessairement qu'un véritable thread soit créé tout de suite, et MARCEL peut ainsi se contenter de ne mémoriser que la fonction à exécuter et son paramètre, ce qui est très léger en comparaison de l'initialisation complète d'un thread. Ensuite, lorsque l'ordonnanceur rencontre une graine de thread, si le thread précédent vient de se terminer (cas le plus courant dans le contexte considéré), il est réutilisé tel quel pour exécuter la graine de thread. Sinon, un véritable thread est créé. En pratique, chaque processeur est alors capable de réutiliser en permanence le même thread pour exécuter rapidement les graines de thread qui ont été placées sur les listes qui le couvrent. La table 4.2 montre les coûts de création et d'exécution des threads et des graines de threads MARCEL sur une machine double-bicœur Opteron cadencée à $1,8$ GHz. On s'aperçoit que le coût (correspondant essentiellement à la gestion des listes de threads, relations de bulles et ordonnancement de base) devient réellement très faible, il est de l'ordre de 400 cycles à la fois pour la création et pour l'exécution. Il devient ainsi envisageable pour une application d'exprimer absolument tout son parallélisme intrinsèque en créant une graine de thread pour chacune de ses tâches. L'ordonnanceur ne pourra alors que mieux les répartir sur la machine. La contrepartie est que certaines opérations, telles que l'annulation d'une graine de thread en cours d'exécution, ne peuvent pas être effectuées. En pratique, cela n'est pas contraignant. Le résultat final est ainsi similaire à l'approche *filaments*, mais quasiment sans changer l'interface de programmation de threads de MARCEL, l'optimisation de création de thread se faisant de manière opportuniste seulement : ce n'est par exemple que si la fonction exécutée par une graine de thread se bloque réellement que l'on est obligé de créer un autre thread pour exécuter une autre graine de thread.

Il serait possible d'aller un peu plus loin si l'application peut promettre que la tâche qu'elle veut faire exécuter n'utilisera jamais d'opérations bloquantes (`mutex`, entrées/sorties, etc.), les opérations non-bloquantes (création de thread, verrous rotatifs, etc.) étant par contre toutes autorisées. Dans ce cas il n'est même pas nécessaire de créer de thread exécutant une graine de thread, la tâche peut être exécutée par l'ordonnanceur directement depuis le thread `idle`.

4.4.3 Distribution des allocations

Il est essentiel d'éviter toute centralisation d'opération pour limiter les contentions. L'allocation d'objets est un des cas typiques pour lesquels une implémentation simpliste peut être catastrophique du point de vue des performances. Par exemple, la bibliothèque MARCEL utilise un « cache » de piles pour éviter d'avoir à solliciter le système à chaque fois qu'elle crée un thread. Dans la version originelle, ce cache est centralisé, cela pose donc des problèmes de contention lorsque MARCEL est utilisé avec de nombreux processeurs : la courbe du haut de la figure 4.8 montre que le temps de création concurrente de nombreux threads explose avec le nombre de processeurs. Pour éviter cela, il est possible de créer un cache par processeur, mais un cas pathologique apparaît alors lorsque par exemple un thread s'exécutant sur le processeur 0 passe son temps à créer des threads qui sont répartis sur la machine et se terminent ainsi loin du processeur 0. Dans ce cas, le cache du processeur 0 est toujours vide pendant que les caches des autres processeurs se remplissent... Ce genre de cas est très difficile à traiter. Lorsqu'elles peuvent être utilisées, les graines de threads permettent de le résoudre en retardant l'allocation effective éventuelle de la pile au moment où l'on exécute les graines. Sinon, une solution est d'avoir quelques niveaux supplémentaires de caches partagés entre processeurs, pour distribuer la réutilisation de piles.

La bibliothèque MARCEL intègre en fait une infrastructure, dont le développement a été confié à un groupe d'étudiants, de cache d'allocation distribuée qui, de plus, prend en compte la structure de la machine. Cette infrastructure est générique : on peut créer un cache d'allocation d'un type d'objet quelconque en fournissant simplement une fonction d'allocation (*e.g.* `malloc`) et une fonction de destruction (*e.g.* `free`). Elle est alors non seulement utilisée pour les allocations de piles de threads, mais aussi pour d'autres structures de la bibliothèque MARCEL telles que les graines de threads, et même les données internes à cette infrastructure elle-même.

Pour un cache donné, à chaque élément hiérarchique de la machine (processeurs, cœurs, nœuds NUMA, machine) est associé un *conteneur*. Lorsqu'un objet est libéré, la fonction de libération du cache essaie d'ajouter l'objet libéré au conteneur du processeur courant. S'il est plein (à cause d'un seuil réglable), elle essaie de l'ajouter au conteneur de la puce courante, puis du nœud NUMA courant, et si l'objet est indépendant des nœuds NUMA, au conteneur de la machine. Si tous ces conteneurs sont pleins, l'objet est réellement libéré à l'aide de la fonction fournie au moment de la création du cache. À l'inverse, lorsque la fonction d'allocation du cache est appelée, elle essaie de piocher un objet depuis le conteneur du processeur courant, puis de la puce courante, puis du nœud NUMA courant, puis éventuellement de la machine, et si tous ces conteneurs sont vides un objet est réellement alloué à l'aide de la fonction fournie.

La courbe du bas de la figure 4.8 montre que le coût de création de nombreux threads reste alors assez stable, même sur 16 processeurs. Cette infrastructure, relativement simple malgré sa généricité, pourrait être améliorée en introduisant par exemple des rééquilibrages entre les conteneurs pour compenser des comportements pathologiques. L'intérêt est que ce genre d'améliorations profiterait alors automatiquement à tous les types d'objets alloués ainsi dans MARCEL.

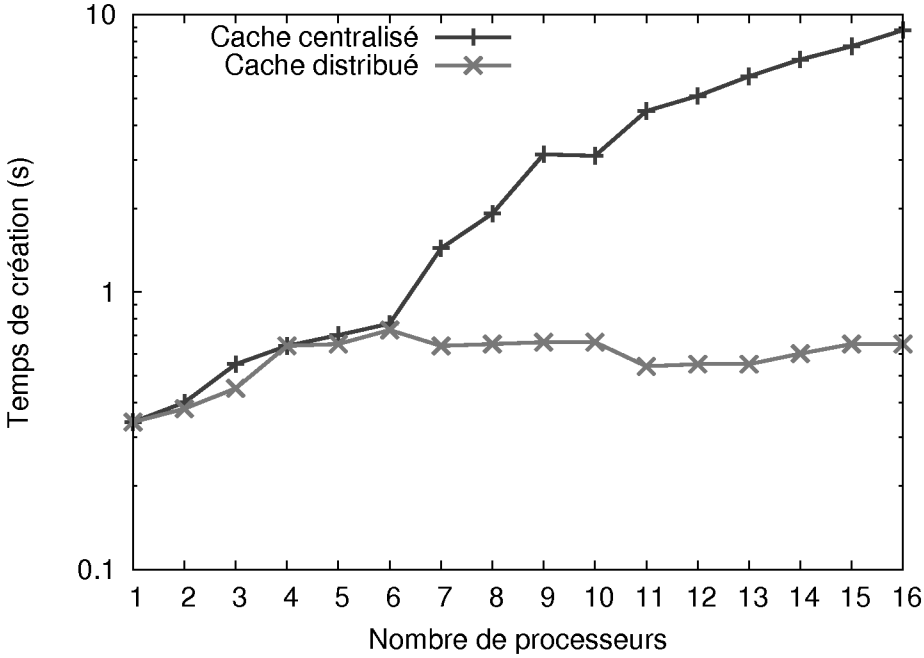


FIG. 4.8 – Temps de création et de destruction parallèle de 100 000 threads.

Chapitre 5

Diffusion

Sommaire

5.1 Compatibilité POSIX et construction automatisée de hiérarchies de bulles 77
5.2 Support OPENMP : FORESTGOMP 79
5.2.1 Parallélisme imbriqué en OPENMP 80
5.2.2 Vers l'utilisation de MARCEL 81
5.2.3 Intégration au sein de GCC : FORESTGOMP 81
5.2.4 Discussion 83

Dans ce chapitre, nous présentons les travaux que nous avons réalisés pour augmenter l'impact de nos résultats de recherche et la diffusion de notre plate-forme BubbleSched. En effet, pour qu'elle puisse être utilisable avec de véritables applications, il faut qu'elle puisse être utilisée avec les environnements de programmation parallèle existants. Nous avons ainsi d'abord adapté notre plate-forme aux threads POSIX, ce qui permet d'exécuter une très grande part des applications, puisqu'ils sont les briques de base classiques des environnements de programmation parallèle. Pour obtenir plus d'informations sur l'application, nous avons également adapté un support à l'environnement OPENMP, dont l'expressivité est plus riche, et pour lequel les compilateurs sont plus à même de déterminer des informations sur l'application.

5.1 Compatibilité POSIX et construction automatisée de hiérarchies de bulles

L'interface la plus naturelle que la bibliothèque MARCEL se doit de supporter est bien sûr l'interface des threads POSIX. L'interface de MARCEL, bien que plus ancienne, est déjà assez proche de celle de POSIX, elle ne diverge que par quelques détails de sémantique. Pendant son post-doctorat, Vincent DANJEAN a ajouté à MARCEL deux couches de compatibilité POSIX. La première est une couche de compatibilité assez légère au niveau interface de programmation (API), qui permet de recompiler pour MARCEL une application existante écrite pour utiliser l'interface de threads POSIX. La deuxième est une couche de compatibilité au niveau interface binaire (ABI), qui permet d'exécuter avec MARCEL une application déjà compilée pour utiliser la bibliothèque de threads POSIX de GNU/Linux, la NPTL. Elle per-

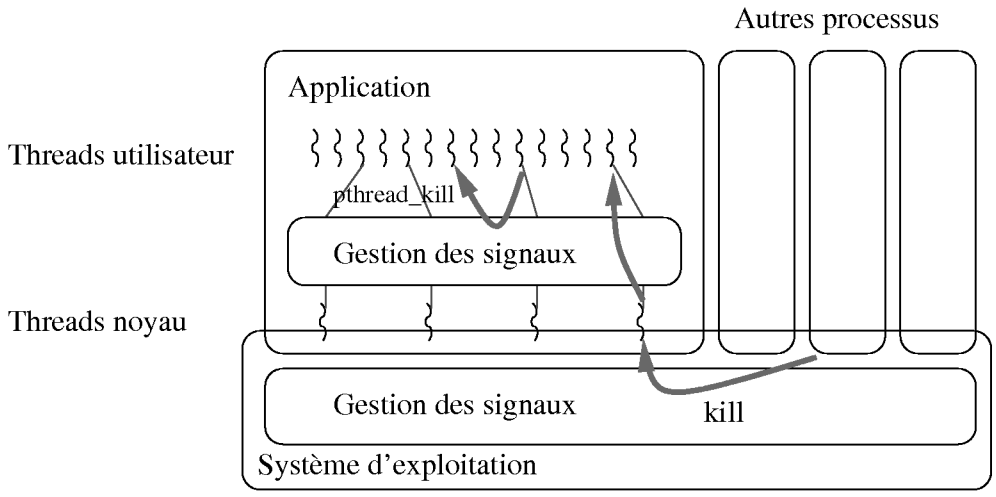
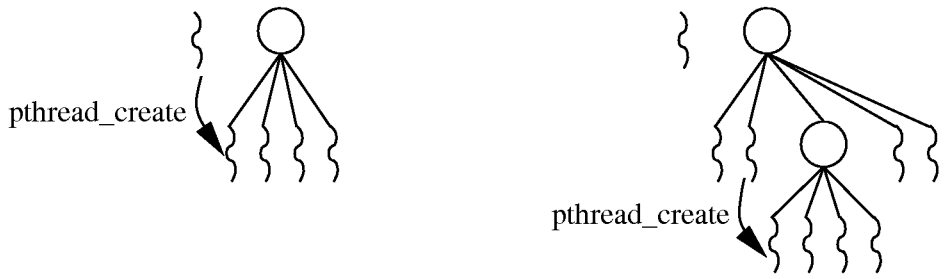


FIG. 5.1 – Modèle hybride de gestion des signaux.
Lorsqu'un autre processus envoie un signal à l'application (représenté à droite), le noyau choisit un thread noyau auquel envoyer le signal. MARCEL choisit alors un thread MARCEL auquel envoyer le signal, si possible celui qui s'exécute déjà actuellement sur le thread noyau ayant reçu le signal par exemple. Les signaux envoyés entre threads MARCEL (tels que représenté au milieu) sont traités directement en interne sans passer par le noyau, sauf s'ils doivent terminer le processus et générer éventuellement un fichier *core*.

met également d'exécuter avec MARCEL une application utilisant un environnement d'exécution basés sur cette même bibliothèque. Cette deuxième couche est plus lourde à implémenter car elle impose de respecter les conventions binaires existantes (tailles des structures de données, valeurs des constantes, etc.). Elle a été portée sur les architectures x86, x86_64 et Itanium.

Ces couches de compatibilité n'étaient cependant pas complètes, il leur manquait notamment la gestion des signaux et la gestion des variables par thread (*Thread Local Storage*, TLS). J'ai donc encadré Sylvain JEULAND durant son stage de Master 1 [Jeu06], qui a consisté à finaliser le travail de Vincent DANJEAN, et notamment implémenter la gestion des signaux au sein de MARCEL. Pour conserver un fonctionnement léger, toute cette gestion est effectuée en espace utilisateur. MARCEL met en place auprès du noyau ses propres traitants de signaux pour réceptionner ceux qui proviennent des autres processus. Par contre, les signaux envoyés entre threads MARCEL ou par la fonction `raise` sont gérés complètement en espace utilisateur. Par ailleurs, j'ai implémenté le mécanisme de TLS [Dre03] au sein de MARCEL. La difficulté est que sur les architectures x86 (resp. x86_64), cela implique le registre de segment `gs` (resp. `fs`) dont la mise à jour est contrainte pour des raisons de sécurité. Il est alors nécessaire, pour chaque thread MARCEL, d'enregistrer auprès du noyau une entrée dans la table LDT (*Local Descriptor Table*). Heureusement, le mécanisme de cache générique présenté à la section 4.4.3 (page 75) permet d'éviter d'effectuer cet enregistrement pour chaque création de thread. Avec ces dernières finitions, les interfaces de compatibilité POSIX sont presque complètes (il manque essentiellement le support des verrous inter-processus), et il est alors possible de lancer sans les modifier ni même les recompiler des applications conséquentes : Mozilla FireFox, OpenOffice.org, mais aussi la JVM (*Java Virtual Machine*) de SUN !

Ainsi il est possible d'exécuter facilement des applications existantes avec la bibliothèque



(a) Le thread principal a lancé quatre autres threads, qui se retrouvent dans une même bulle.

(b) Un des quatre threads a lui-même lancé quatre autres threads, qui se retrouvent dans une même sous-bulle.

FIG. 5.2 – Construction automatique de bulles à partir des relations de filiation POSIX.

MARCEL. Cependant, l'interface POSIX n'intègre aucun moyen d'indiquer les relations entre threads qui pourraient être utiles pour construire des hiérarchies de bulles. Nous avons donc ajouté un module optionnel utilisant les relations de parenté entre threads pour établir automatiquement une hiérarchie de bulles, comme illustré figure 5.2 : lorsqu'un thread crée d'autres threads, une bulle est automatiquement créée pour le père, et les fils sont placés dedans. Si ces fils créent eux-mêmes d'autres threads (exprimant ainsi un parallélisme imbriqué), des bulles sont créées automatiquement, elles sont insérées dans la bulle du père, les petits-fils sont insérés dans ces bulles de manière appropriée, etc. On construit ainsi automatiquement un arbre de bulles correspondant à l'arbre de parenté entre threads. Ainsi, lorsqu'une application ou un environnement de développement parallèle crée des threads, on peut au moins appliquer un ordonnancement à bulles sur l'arbre de parenté des threads, qui se trouve être souvent assez approprié, par exemple lorsque le parallélisme est *enfoui* dans des fonctions de bibliothèques de haut niveau. On verra à la section 6.2.2 (page 93) comment, dans le cas où une application soumet des factorisations LU en parallèle, il est possible, en utilisant simplement cette information, d'effectuer un ordonnancement adapté.

D'autres conventions de construction de bulles pourraient être envisagées, on pourrait par exemple préférer placer le thread père dans la bulle avec ses fils. C'est en fait l'approche qui est utilisée pour le support d'OPENMP, détaillé à la section suivante.

5.2 Support OPENMP : FORESTGOMP

Le standard OPENMP [ope] consiste en un ensemble de directives (sous forme de `pragma` en C et C++, sous forme de commentaire en Fortran) que le programmeur peut ajouter à son programme séquentiel pour indiquer au compilateur comment celui-ci peut être parallélisé, sous quelles contraintes de variables, etc. OPENMP est en fait devenu assez incontournable. D'une complexité relativement légère, il est supporté par tous les principaux compilateurs, notamment ceux d'INTEL, IBM, SUN, FUJISTU, GCC, et même dans la dernière version de VISUAL STUDIO de MICROSOFT. Il paraît donc naturel d'utiliser notre plate-forme BubbleSched dans ce contexte, d'autant plus qu'OPENMP permet d'exprimer du parallélisme imbriqué.

5.2.1 Parallélisme imbriqué en OpenMP

Le support dans OpenMP du parallélisme imbriqué est en fait quelque peu un problème de l'œuf et de la poule. En effet, bien que les sections parallèles emboîtées soient présentes dans le standard depuis sa première version, leur implémentation effective n'a jamais été obligatoire, et il a fallu plusieurs années pour voir apparaître les premières implémentations, qui étaient encore sommaires. Ainsi, il n'existe que très peu de *benchmarks* utilisant du parallélisme imbriqué : à ce jour, la suite SPEC OMP [SPE] n'en utilise pas du tout, et même si trois programmes parmi la suite *NAS Parallel Benchmarks* existent en version multizone [WJ03], ils n'utilisent pas de section parallèle imbriquée OpenMP, mais plusieurs processus lançant chacun une section parallèle... Par conséquent, les auteurs de compilateurs OpenMP n'ont que peu de raisons de dépenser du temps à réaliser un bon support du parallélisme imbriqué. Certains compilateurs ne le supportent encore pas du tout et se contentent de sérialiser les sections imbriquées ; la plupart se contentent de lancer des threads noyau à chaque entrée en section parallèle, en les réutilisant éventuellement au sein d'un réservoir pour limiter le coût que cela représente ; certains enfin limitent la profondeur de récursivité. Très peu de compilateurs effectuent de véritable *placement* intelligent sur la machine sous-jacente. Il est ainsi devenu répandu au sein de la communauté OpenMP que « le parallélisme imbriqué n'est pas efficace », et il n'est donc pas utilisé, etc...

Pourtant, beaucoup d'applications se prêtent bien au parallélisme imbriqué, que ce soient les maillages adaptatifs décrits à la section 1.1.1 (page 6) ou par blocs [GSW+06], les approches « diviser pour régner » [Nit06], ou le parallélisme enfoui au sein de bibliothèques de calcul. De nombreuses études [MAN+99, TTSY00, AGMJ04, DSCL04, DGC05, BS05, GSW+06, MST07] montrent qu'en fait cela permet en général de faciliter le passage à l'échelle d'un programme parallèle. Comme le disent Tian *et al.* [THH+05], les sections parallèles imbriquées d'OpenMP sont d'ailleurs une manière assez naturelle d'exprimer le parallélisme récursif. L'introduction imminente des *task queues* dans OpenMP 3.0 [ACD+07] sera enfin une manière supplémentaire de pouvoir exprimer un tel parallélisme.

Par ailleurs, plusieurs compilateurs (la plupart expérimentaux) ont un support plus ou moins optimisé. Xinmin Tian a travaillé au sein du compilateur Intel `icc` pour s'assurer d'avoir une *implémentation efficace* du point de vue de la création et attente des threads noyau [TGS+03], des verrous [THH+05], et autres optimisations usuelles (ordonnancement de boucle, utilisation d'instructions vectorielles, etc. [TGBS05]). Rien n'est fait cependant pour placer les threads de manière cohérente, et d'autant moins sur machine hiérarchique. Omni/ST est un portage du compilateur Omni sur la bibliothèque de threads utilisateur StackThreads/MP. Un effort y est fait pour distribuer les niveaux de parallélisme OpenMP de manière *cohérente* sur les processeurs, mais l'équilibrage de charge n'utilise ni les informations d'affinités entre threads ni la nature hiérarchique de la machine. La dernière version à ce jour du compilateur OMPi [HD07], basée sur une bibliothèque de threads utilisateur `psthreads`, distribue le premier niveau de parallélisme sur les différents processeurs, puis pour les niveaux suivants crée localement des threads utilisateur qui sont placés en tête d'une liste locale au processeur. L'équilibrage de charge est alors assuré par un *vol hiérarchisé*, en général selon la structure de la machine. La localité d'exécution est favorisée car les processeurs piochent d'abord en tête de leur propre liste, et sinon volent en queue de liste des autres processeurs (donc les threads correspondant aux niveaux de parallélisme les plus externes). Cependant, les relations entre threads ne survivent pas au vol : une fois

volé, rien ne lie un thread à ceux qui avaient été créés en même temps, et les affinités entre threads ne sont alors plus prises en compte, ce qui peut amener des pertes de performances. Enfin, certaines implémentations [GOM+00, AGMJ04] proposent une extension à OPENMP qui permet de placer explicitement les threads sur la machine. Une telle approche ne peut cependant pas être portable, car la numérotation des processeurs ne sera pas cohérente selon l'organisation de la machine, entre une machine quadruple-bicœur ou une machine comportant deux nœuds NUMA composés chacun de deux processeurs double-bicœurs, par exemple.

5.2.2 Vers l'utilisation de MARCEL

La plupart des compilateurs utilisent, pour leur environnement d'exécution, des threads POSIX. Il est alors facile de leur faire utiliser la bibliothèque MARCEL au lieu des threads noyau, grâce à sa couche de compatibilité POSIX. À la section précédente, nous avons vu une manière de construire automatiquement des bulles à partir de la filiation des threads. Ici, cette manière n'est pas tout à fait appropriée. La figure 5.3 montre une variante plus adaptée au cas d'OPENMP. La différence essentielle est que lorsqu'un thread en crée d'autres, il se place lui-même dans la bulle où il a placé les threads créés. En effet, dans le modèle de programmation d'OPENMP, lorsqu'un thread entre dans une section parallèle, il participe à l'exécution de section en tant que thread maître.

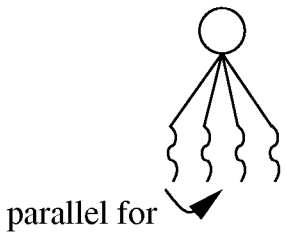
Le problème est qu'il n'est pas possible de parvenir automatiquement à une telle construction. En effet, si l'on considère le thread principal, rien n'indique à coup sûr à quel moment il a terminé de créer les threads correspondant au premier niveau de parallélisme et qu'il va donc commencer à créer des threads pour le deuxième niveau de parallélisme. Des heuristiques basées sur les intervalles entre les créations de threads pourraient aider à déterminer ce moment, mais une telle approche est bien peu sûre. Par ailleurs, la plupart des compilateurs utilisent un pool de threads pour éviter d'avoir à recréer des threads à chaque entrée de section parallèle. Cette réutilisation se fait de plus sans prendre en compte l'« histoire » des threads, c'est-à-dire qu'un thread peut être réutilisé pour une quelconque section parallèle sans se soucier des sections parallèles auxquelles il a participé par le passé. Dans ce cas, la bibliothèque de threads n'est pas informée de ces réutilisations, et donc MARCEL ne peut pas savoir comment construire automatiquement la hiérarchie de bulles.

5.2.3 Intégration au sein de GCC : FORESTGOMP

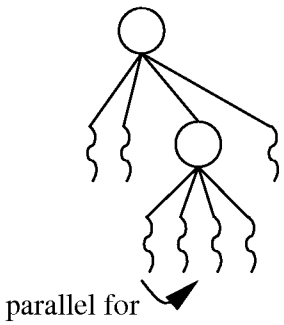
Pour pouvoir contrôler plus précisément la manière dont les bulles sont créées, et surtout pouvoir à terme leur attacher le plus d'informations possible pour guider l'ordonnanceur à bulles, nous avons intégré l'utilisation de la bibliothèque MARCEL et des outils de la plate-forme BubbleSched au sein de l'implémentation OPENMP de GCC, GOMP, pour en faire une version utilisant toutes les possibilités de notre plate-forme, FORESTGOMP. Comme de nombreux autres compilateurs OPENMP, GOMP dispose en effet de plusieurs *backends* pour pouvoir utiliser différentes bibliothèques de threads (POSIX, SOLARIS, WINDOWS), il a donc suffit d'ajouter un *backend* MARCEL. Nous avons de plus étendu le code interne de GOMP, notamment pour lui faire créer des bulles ainsi qu'expliqué à la figure 5.3, le code est détaillé à la figure 5.4. Nous en avons également profité pour optimiser l'utilisation de barrières

```
#pragma omp parallel for
  for (i=0; i<4; i++)
#pragma omp parallel for
    for (j=0; j<4; j++)
```

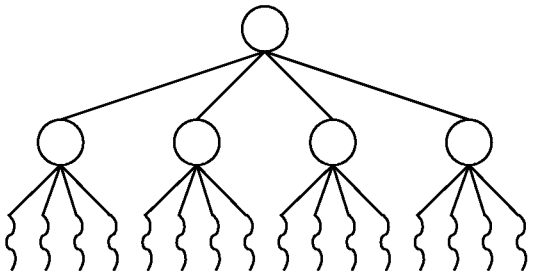
(a) Extrait de code OPENMP



(b) Le thread principal est entré dans la section parallèle externe et a donc créé trois autres threads. Tous se retrouvent dans une même bulle correspondant à cette équipe.



(c) Un des quatre threads est entré dans la section parallèle interne et a donc créé trois autres threads. Tous se retrouvent dans une même sous-bulle correspondant à cette sous-équipe.



(d) Les trois autres threads du premier niveau de parallélisme sont entrés dans la section parallèle interne et ont donc chacun créé trois autres threads. Chaque sous-équipe de thread se retrouve ainsi dans sa propre sous-bulle.

FIG. 5.3 – Construction automatique de bulles à partir des relations de filiation OPENMP.

```
void gomp_team_start (void (*fn) (void *), void *data, unsigned nthreads,
                      struct gomp_work_share *work_share) {
  struct gomp_team *team;
  team = new_team (nthreads, work_share);
  ... /* Enregistrer 'fn' et 'data' dans la structure 'start_data' */

  if (nthreads > 1 && team->prev_ts.team != NULL) {
    /* parallélisme imbriqué, créer une sous-bulle */
    marcel_bubble_t *holder = marcel_bubble_holding_task (marcel_self());
    marcel_bubble_init (&team->bubble);
    marcel_bubble_insertbubble (holder, &team->bubble);
    marcel_bubble_insertthread (&team->bubble, marcel_self());
    marcel_attr_setinitbubble (&gomp_thread_attr, &team->bubble);
  }

  for(int i=1; i < nbthreads; i++) {
      pthread_create (NULL, &gomp_thread_attr,
                      gomp_thread_start, start_data);
      ...
  }
}
```

FIG. 5.4 – Code de construction des bulles adaptées à OPENMP.

partielles en début et en fin de sections parallèles. En effet, en fin de section parallèle par exemple, le thread maître d'une équipe de threads doit effectivement bien s'assurer de la terminaison des autres threads avant de commencer à exécuter les instructions situées *après* la section parallèle. Les autres threads n'ont cependant aucunement besoin de s'attendre les uns les autres, et peuvent ainsi terminer de manière asynchrone. Enfin, il sera à terme utile d'ajouter un transfert d'informations depuis le compilateur vers l'ordonnancement à bulles, ainsi que décrit à la section 1.4.4 (page 22) : schéma d'accès au données, zones de données effectivement les plus utilisées, etc. Ceci permettra par exemple de donner les informations dont l'ordonnanceur *MemAware* a besoin pour effectuer des décisions selon le placement des données.

Au final, il suffit, pour exécuter un programme OPENMP avec notre version FORESTGOMP, de le compiler normalement avec GCC, puis d'utiliser, par exemple, la variable d'environnement `LD_LIBRARY_PATH` pour imposer l'utilisation de la version modifiée au lieu de la version standard.

Il est à noter que, en même temps et de manière indépendante, Panagiotis HADJIDOUKAS a ajouté un *backend* MARCEL au compilateur OMPI [HD07]. L'implémentation actuelle ne fait pas usage des bulles, mais une collaboration entre nos deux équipes de recherche devrait commencer sous peu.

5.2.4 Discussion

Le portage étant effectué, se posent alors de nombreuses questions. Comment ordonnancer les threads d'OPENMP ? Combien en créer ? Quelles extensions utiliser pour apporter toujours plus d'informations à l'ordonnanceur pour qu'il puisse effectuer un travail approprié ?

Quel ordonnanceur utiliser ?

Une approche d'ordonnancement possible est par exemple d'utiliser d'abord un algorithme de type *Spread* (*Spread*, *Affinity* ou la version *MemAware* de *Spread*) pour répartir les threads OpenMP de manière adaptée à la machine. Lors de la création dynamique d'une sous-équipe de threads pour réaliser du parallélisme interne, la bulle correspondante est conservée « sur place », à l'endroit où le thread qui a créé cette équipe se trouvait. Si à cause d'un déséquilibre de charge un processeur devient inactif, on peut utiliser un algorithme de type *Steal* ou bien une redistribution locale à l'aide d'un algorithme de type *Spread*. Il se trouve que cela revient à peu près à effectuer l'ordonnancement proposé dans l'environnement d'exécution du compilateur OMPi, la différence la plus notable étant qu'ici les informations d'affinités sont conservées, ce qui permet de prendre des décisions appropriées *tout au long de l'exécution*.

Combien de threads créer ?

Une question importante est de déterminer quelle arité choisir pour chaque section parallèle, c'est-à-dire le nombre de threads à créer à chaque entrée de section.

Certaines approches [DSCL04] consistent, lors de l'entrée dans une section parallèle, à comparer la taille du domaine d'indice de la section avec le nombre de processeurs encore disponibles. Si la section fournit un parallélisme suffisant pour alimenter tous les processeurs, les sections emboîtées plus profondément seront exécutées en série. Une telle approche permet certes d'éviter une surcharge de threads tout en conservant une occupation de tous les processeurs, mais elle ne laisse que bien peu de liberté à l'ordonnanceur, qui ne peut alors guère faire mieux que placer un thread par processeur, au risque d'obtenir alors un déséquilibre de charge potentiellement important.

Pour d'autres approches on se permet de créer bien plus de threads qu'il n'y a de processeurs, ce qui permet à l'ordonnanceur d'effectuer une répartition de charge dynamiquement adaptée à l'exécution de l'application. Il reste cependant *a priori* nécessaire d'éviter de créer des myriades de threads, Duran *et al.* [DGC05] utilisent une méthode dynamique qui essaie différentes distributions de nombres de threads sur les niveaux de parallélisme, observe les temps d'exécution, et choisit la distribution qui paraît la meilleure.

En pratique, Nitsure [Nit06] observe que sur ses algorithmes Lattice-Boltzmann, la meilleure méthode est encore d'utiliser systématiquement une arité égale au nombre de processeurs disponibles. Cela paraît en effet assez logique : cela correspond à la capacité maximale de parallélisme disponible sur la machine. Indiquer des arités moindres peut risquer de laisser des processeurs inoccupés si les autres sections parallèles sont inactives. À l'inverse, indiquer des arités plus grandes que le nombre de processeurs n'est *a priori* pas utile pour gagner en parallélisme.

En fait, il faut sans doute effectivement observer le rapport entre le parallélisme disponible et le nombre de processeurs. Si le parallélisme disponible au niveau d'une section parallèle est bien plus grand que le nombre de processeurs, on peut utiliser autant de threads qu'il y a de processeurs, et laisser la distribution des indices de boucle s'effectuer de manière statique (si la répartition de charge est uniforme) ou dynamique (si la répartition de charge

est irrégulière). S'il n'y a pas d'autres sections parallèles ou qu'elles sont inactives, on pourra ainsi utiliser tous les processeurs. Sinon, l'ordonnanceur pourra effectuer une répartition des threads eux-mêmes sur les processeurs disponibles. Si, à l'inverse, le parallélisme disponible n'est pas beaucoup plus grand que le nombre de processeurs, il vaudra sans doute mieux créer autant de threads que le parallélisme le permet, l'ordonnanceur s'occupant alors de répartir la charge.

On pourrait cependant aussi utiliser une approche extrême : créer autant de threads que le parallélisme le permet. Nous avons décrit à la section 4.4.2 (page 73) comment il était possible de créer de manière très légère (de l'ordre de 200 ns) des graines de threads, qui peuvent alors être exécutées très rapidement (de l'ordre de 200 ns également). On peut alors envisager de créer une graine de thread pour chaque itération de la section parallèle à l'aide de la directive OPENMP `schedule(static,1)`. Un ordonnanceur à bulles dispose ainsi de toute latitude pour répartir la charge de calcul de manière appropriée. Comme nous l'avons vu, l'exécution proprement dite se fait en réutilisant le plus possible les véritables threads, si bien qu'en pratique, chaque processeur réutilise en permanence un même véritable thread pour exécuter de manière très légère toutes les graines de threads qui lui ont été confiées par l'ordonnanceur à bulles. En pratique, aussi faibles soient-ils, les coûts de création et d'exécution d'une graine de thread peuvent se révéler prépondérants devant le temps de calcul d'une itération de boucle, il sera dans ce cas utile d'utiliser par exemple la directive `schedule(static, 10)` pour rassembler une dizaine d'itérations de boucles par graine de thread pour que le coût des graines redevienne raisonnable.

Enfin, il serait sans doute utile d'étendre l'interface OPENMP pour ajouter des directives permettant par exemple de choisir l'ordonnanceur à bulles utilisé, fournir la charge de calcul estimée d'une itération de boucle, indiquer les zones mémoire les plus utilisées, préciser les étapes de progression de l'application (initialisation, terminaison par exemple), etc. Nous reviendrons sur cette perspective à la section 7.2.

Chapitre 6

Validation

Sommaire

6.1 Tests synthétiques 87
6.1.1 Opérations de base 87
6.1.2 Stress-test 90
6.2 Applications 90
6.2.1 ADVECTION / CONDUCTION, un parallélisme régulier 92
6.2.2 SUPERLU, un parallélisme de tâches 93
6.2.3 NPB BT-MZ, un parallélisme imbriqué irrégulier dans l'espace . . . 94
6.2.4 MPU, un parallélisme imbriqué irrégulier dans le temps 97

Dans ce chapitre, nous présentons les résultats d'expériences que nous avons menées afin d'une part de démontrer la pertinence de notre modèle d'ordonnancement et d'autre part d'évaluer les performances de sa mise en œuvre. Des tests synthétiques permettent d'abord de vérifier dans quelle mesure le surcoût introduit est réduit, ainsi que les gains obtenus sur des exemples critiques. Des applications concrètes, exécutées grâce aux supports POSIX et OPENMP détaillés au chapitre 5, viennent ensuite valider les différents ordonnanceurs développés dans la section 3.2.

6.1 Tests synthétiques

Tout d'abord, nous utilisons quelques tests synthétiques pour vérifier les performances brutes de l'implémentation de notre plate-forme. En effet, nous avons ajouté une grande quantité de code pour gérer les aspects NUMA de la machine ainsi que les hiérarchies de bulles, il faut donc vérifier que ce code n'apporte pas de surcoût excessif.

6.1.1 Opérations de base

À l'aide de quelques microbenchmarks, nous vérifions le surcoût ajouté aux opérations de base. Ces tests ont été exécutés sur une machine double-bicœur Opteron cadencée à $1,8$ GHz avec quatre profils de compilation de MARCEL : un profil monoprocesseur qui n'exploite

	Création	Exécution
MARCEL Mono graine	0,15	0,12
MARCEL SMP graine	0,21	0,22
MARCEL NUMA graine	0,22	0,22
MARCEL NUMA bulles graine	0,25	0,32
MARCEL Mono	0,60	0,46
MARCEL SMP	0,80	0,60
MARCEL NUMA	0,85	0,60
MARCEL NUMA bulles	0,85	0,70
NPTL	6,4	11,3

FIG. 6.1 – Temps de création de threads et de graines de threads (μs).

qu'un seul processeur (ce qui permet notamment de se passer de verrouillage), un profil SMP qui exploite les quatre processeurs sans se soucier des aspects hiérarchiques de la machine, un profil NUMA qui les prend en compte (gestion de topologie et ordonnanceur de base hiérarchisé), et un profil NUMA avec bulles où l'on ajoute le support des bulles. Pour comparaison, les tests ont également été exécutés avec la bibliothèque de threads native de Linux, la NPTL, avec un noyau Linux 2.6.22.

La figure 6.1 montre les résultats obtenus pour le temps de création et d'exécution de threads et de graines de threads. On constate que malgré un léger surcoût, créer des threads utilisateur reste bien moins coûteux que de créer des threads noyau avec la NPTL. La création de graines de threads (détaillée à la section 4.4.2) est encore moins coûteuse. Dans le cas du profil avec bulles, les threads sont créés au sein d'une bulle, ce qui permet de prendre en compte le coût lié à l'insertion dans celle-ci, qui se révèle léger[1].

La figure 6.2 présente le coût du changement de contexte entre deux threads. La colonne *Schedule* indique le temps d'exécution de l'ordonnanceur de base lorsqu'il n'y a pas de nouveau thread intéressant à exécuter, c'est-à-dire en quelque sorte le coût *à vide*. La colonne *Schedule+Switch* fournit quant à elle le temps d'exécution de l'ordonnanceur de base dans le cas où il décide de basculer sur un autre thread. Cela comprend donc non seulement le coût de basculement proprement dit d'un thread à un autre, mais aussi la maintenance des threads : prélèvement du prochain thread et remise en queue de liste du thread précédent. Dans le cas du profil avec bulles, un des deux threads est situé dans une bulle, ce qui permet de prendre en compte le surcoût lié à la recherche de ce thread au sein de la bulle. On s'aperçoit qu'il reste léger. En regard de la NPTL, on peut également remarquer qu'utiliser une bibliothèque de threads utilisateur permet de conserver un temps d'exécution de *Schedule* (utilisé en de nombreux endroits où il est potentiellement utile de rendre la main) inférieur à 50 ns, ce qui est difficile pour une bibliothèque de threads noyau à cause du coût d'un appel système.

Les ordonnanceurs à bulles étant censés faire un usage fréquent de la migration de threads entre processeurs, il est également utile de connaître le coût d'une telle opération. La figure 6.3 montre donc le coût de la migration d'un thread entre deux processeurs. Ce coût

[1] Il n'est pas utile de tester l'insertion profondément à l'intérieur d'une hiérarchie de bulles puisque, comme expliqué à la section 4.4.1, nous utilisons dans la bulle la plus englobante un *cache* de thread, assurant ainsi un parcours en $O(1)$.

	Schedule	*Schedule+Switch*
MARCEL Mono	0,024	0,135
MARCEL SMP	0,043	0,205
MARCEL NUMA	0,046	0,270
MARCEL NUMA bulles	0,046	0,310
NPTL	0,33	0,49

FIG. 6.2 – Temps d'ordonnancement et de changement de contexte(μs).

	Ordre (inactif)	Ordre (actif)	Latence
MARCEL SMP	0,047	0,050	2,1
MARCEL NUMA	0,047	0,051	2,2
MARCEL NUMA bulles	0,054	0,057	2,5
NPTL	0,8	8	4,8

FIG. 6.3 – Coût de migration d'un thread (μs).

est décliné en trois mesures : deux mesures du temps nécessaire pour donner l'ordre de migration, selon que le thread concerné est actuellement actif ou non, et une mesure du temps nécessaire pour que la migration soit effective (la *latence* de la migration, donc). On peut constater que donner l'ordre de migration reste très peu coûteux, que le thread soit actif ou non. En effet, il s'agit seulement de prendre deux verrous et de manipuler des listes. La migration effective se fait par contre de manière asynchrone. Soit elle est simplement faite à la prochaine préemption automatique de MARCEL, soit l'ordonnanceur à bulles peut décider d'envoyer des signaux Unix de préemption qui forcent la migration effective immédiate. Envoyer un tel signal coûte environ $1\,\mu$s, que l'ordonnanceur à bulles peut donc décider de dépenser ou non selon ses besoins. Au final, les mesures montrent qu'il faut compter un peu plus de $2\,\mu$s de latence entre l'ordre de migration et la migration effective (le temps que les signaux soient émis, se propagent *via* des interruptions inter-processeur, et que MARCEL effectue les préemptions sur les processeurs). Les mesures obtenues dans le cas de la NPTL sont intéressantes : si le thread dont on demande la migration n'est pas en cours d'exécution, le coût reste relativement raisonnable. S'il est par contre en cours d'exécution, le coût est très fort : $8\,\mu$s ! En effet, lorsque l'on effectue l'appel système `pthread_setaffinity_np()` pour demander la migration, cet appel ne retourne qu'une fois que le thread s'exécute effectivement sur le processeur cible, c'est-à-dire que l'interface fournie est synchrone, et ne laisse pas de possibilité de donner des ordres asynchrones.

De manière plus générale, on peut observer que l'ajout de la gestion multiprocesseur dans la bibliothèque MARCEL double presque les coûts à cause de la nécessité d'effectuer du verrouillage, de prendre en compte les situations de concurrence, etc. L'ajout du support NUMA ajoute un petit surcoût dû à la gestion de la topologie. Enfin, l'ajout de la notion de bulle n'ajoute qu'un léger surcoût. En comparaison, la bibliothèque de thread native de Linux a un coût toujours bien plus grand.

6.1.2 Stress-test

Le programme de test synthétique *sumtime* calcule la somme des nombres de 1 à n avec une méthode pour le moins inefficace : diviser pour régner. Pour calculer la somme de i à j, lorsque $i \neq j$ il crée donc deux threads, l'un s'occupant de calculer la somme de i à $\lfloor \frac{i+j}{2} \rfloor$ et l'autre celle de $\lfloor \frac{i+j}{2} \rfloor + 1$ à j. Ce programme crée ainsi un grand nombre de threads ($2n-1$) dont les relations sont organisées en arbre binaire équilibré, chaque thread ayant une charge de calcul utile quasi-nulle (retourner 1 ou effectuer une addition). La hiérarchie de bulles qu'il est naturel de construire est très similaire à celle qui a été présentée section 3.1.1 (page 38) pour le calcul de la suite *Fibonacci* : lorsqu'un thread crée d'autres threads, il les place dans une bulle qu'il insère dans la bulle où lui-même a été placé, produisant ainsi un arbre de bulles et threads correspondant à l'arbre de calcul.

La figure 6.4 montre les résultats obtenus selon les profils de compilation et les stratégies d'ordonnancement utilisés. On peut déjà remarquer que selon qu'on utilise un profil de compilation *mono*, SMP ou NUMA, on observe un léger surcoût cohérent avec les microbenchmarks effectués à la section précédente. Dans les cas où l'on ne crée pas de bulles, MARCEL intègre tout de même des stratégies de placement des threads créés, notamment *shared* et *affinity*. La stratégie *shared* consiste à placer tous les threads créés sur la liste globale, ce qui permet certes une répartition de charge triviale, mais induit également un point de contention et ne permet pas de privilégier les affinités. On constate effectivement que les résultats obtenus à l'aide de cette stratégie divergent nettement avec un nombre important de threads. La stratégie de placement de threads *affinity*, à l'inverse, place toujours les threads créés sur le processeur local, sauf si un processeur est inactif, auquel cas un thread lui est confié. En pratique, les premiers threads créés (faisant ainsi partie du haut de l'arbre de calcul) sont rapidement distribués sur les différents processeurs, et les autres threads sont quasiment tous créés localement, ce qui améliore le passage à l'échelle. Dans le cas où l'on crée des bulles, nous utilisons l'ordonnanceur à bulles *Burst*. En effet, les threads faisant partie du haut de l'arbre de calcul sont ainsi rapidement répartis sur les différents processeurs, et les créations de threads ultérieures sont toujours effectuées en local. Le résultat montre un surcoût non négligeable : le temps d'exécution est doublé par rapport à la version NUMA sans bulles. Ceci vient bien sûr du fait que chaque thread crée une bulle, l'insère dans une autre, et insère deux threads dedans. Cependant, par la localité de création des threads qui en résulte, cette approche passe également à l'échelle, ce qui finit par compenser le surcoût à partir de $n = 100\ 000$. À titre de comparaison, on peut observer que la bibliothèque de threads de Linux, la NPTL, obtient de bien moins bonnes performances, et ne parvient même plus à exécuter le programme pour $n \geq 15000$.

Ainsi, il apparaît que même si la création d'une hiérarchie de bulles a un certain coût, celui-ci reste raisonnable, et même sur des cas critiques avec de très nombreux threads, l'implémentation continue de passer à l'échelle.

6.2 Applications

Nous présentons les résultats obtenus avec quatre applications, qui permettent d'illustrer à la fois l'utilisation des différentes manières d'exprimer des bulles et l'utilisation des dif-

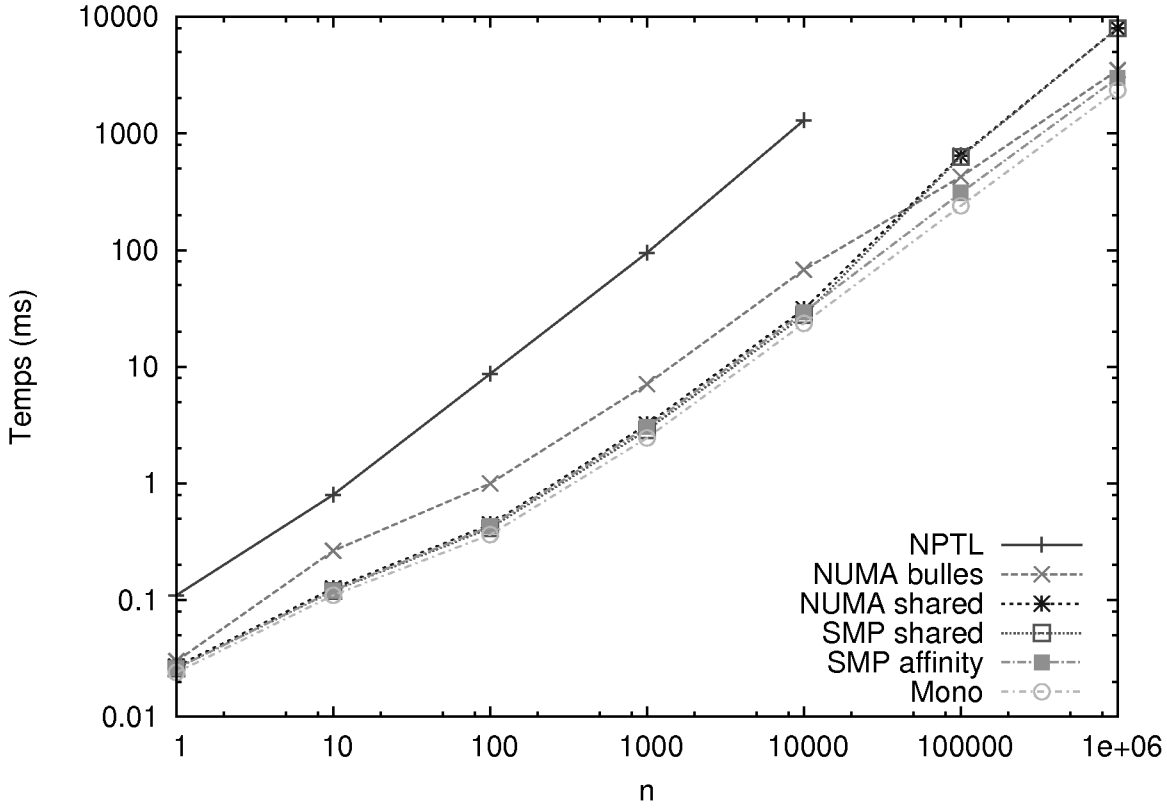


FIG. 6.4 – Performances du programme *sumtime* en fonction des profils de compilation et des stratégies d'ordonnancement.

férents ordonnanceurs détaillés à la section 3.2 (page 44). La première application, ADVECTION / CONDUCTION, assez simple, utilise une création explicite de bulles et l'ordonnanceur *Burst* pour répartir de manière opportuniste les threads sur la machine. La deuxième, SUPERLU, utilise la construction implicite de bulles par la couche POSIX et l'ordonnanceur de gang pour exécuter des jobs de manière concurrente. La troisième, BT-MZ, utilise l'interface OPENMP pour exprimer deux niveaux de parallélisme imbriqués, ainsi que l'ordonnanceur *Spread* pour répartir les bulles d'une manière équilibrée en terme de charge. La dernière, MPU, utilise l'interface OPENMP pour exprimer un parallélisme très déséquilibré et profite de l'implémentation des graines de threads pour pouvoir exprimer ce parallélisme à un grain extrêmement fin. Puisque la charge de calcul n'est *a priori* pas connue, MPU utilise l'ordonnanceur *Affinity*.

6.2.1 ADVECTION / CONDUCTION, un parallélisme régulier

Les applications ADVECTION et CONDUCTION [Pér05], dont le support commun d'exécution MPC implémenté par Marc PÉRACHE [Pér06] a été porté pour MARCEL, effectuent des simulations d'advection et de conduction de chaleur au sein d'un grand maillage 2D régulier découpé en bandes. L'exécution est composée de cycles de calcul purement parallèle séparés par des étapes de communication globale hiérarchique. Dans les deux applications le coût des communications entre les mailles est plutôt important par rapport au temps de calcul (à cause de la latence pour l'advection, à cause du débit pour la conduction). Il est donc important lors de la parallélisation de prendre garde aux positions relatives des threads.

La machine cible est une machine BULL NOVASCALE comportant 4 nœuds NUMA composés de 4 processeurs ITANIUM II et 16 Go de mémoire, soit un total de 16 processeurs et 64 Go de mémoire. Le facteur NUMA est théoriquement de 3, en pratique on peut mesurer un facteur $1,5$.

La figure 6.1 montre les résultats obtenus selon les approches utilisées pour ordonnancer les threads. Dans la version *shared*, on utilise la stratégie de placement *shared* de MARCEL, qui place les threads sur la liste globale, effectuant ainsi un simple ordonnancement opportuniste ne se préoccupant pas des affinités entre threads, l'accélération obtenue est alors assez limitée. Dans la version *fixée*, les applications ont été modifiées pour placer explicitement les threads sur les processeurs de la machine cible d'une manière adaptée à la hiérarchie de cette machine, ce qui permet ainsi d'obtenir un gain d'accélération appréciable. Le calcul de ce placement n'est cependant pas portable puisque la numérotation des processeurs ne peut être liée de manière standard à l'organisation hiérarchique de la machine. La version avec *bulles* a été obtenue en construisant simplement une hiérarchie de bulles selon la structure de l'application. En effet, pour éviter les contentions, le support d'exécution MPC effectue les communications entre threads de manière hiérarchique, en associant à 4 threads un thread de communication, qui lui-même est associé à 4 autres threads de communication et un thread de communication globale. Marc PÉRACHE a donc pu ajouter la construction de bulles d'une manière plutôt naturelle directement au sein du support d'exécution MPC, en rassemblant dans des bulles les threads par groupes de 4 avec le thread de communication associé, puis en rassemblant ces bulles et le thread de communication globale dans une bulle. Puisque l'application est ici très régulière, il n'y a besoin ni de répartition selon la charge ni de vol de travail, et l'on se contente d'utiliser l'ordonnanceur *Burst* (décrit à la section 3.2.3

	Advection		Conduction	
	Temps (s)	Accélération	Temps (s)	Accélération
Séquentiel	16,13		250,2	
Simple	1,77	9,11	23,65	10,58
Fixé	1,30	12,40	15,82	15,82
Bulles (*Spread*)	1,30	12,40	15,84	15,80

TAB. 6.1 – Performances des applications ADVECTION et CONDUCTION selon les approches.

page 48), qui fait en sorte que les processeurs « tirent » chacun les bulles vers eux de manière opportuniste. Le résultat est une distribution de threads équivalente (sans forcément être égale) au placement manuel effectué dans la version *fixée*. Les résultats obtenus sont ainsi très proches de cette version. La différence est que la version avec bulles a été plus naturelle à implémenter et surtout s'adapte automatiquement à d'autres organisations hiérarchiques de machines.

6.2.2 SUPERLU, un parallélisme de tâches

SUPERLU [DGL99] est une bibliothèque de résolution parallèle de grands systèmes d'équations linéaires creux et non symétriques (factorisation LU) qui s'appuie sur des routines BLAS pour profiter au mieux des capacités de calcul et de cache de la machine cible. Les affinités de cache sont ainsi prépondérantes pour obtenir une bonne efficacité.

La machine cible est la même que celle utilisée pour les microbenchmarks de la section 6.1.1, c'est-à-dire une machine double-bicœur Opteron. Le facteur NUMA entre les deux puces est mesuré à peu près à 1,4. La figure 6.5(a) montre combien SUPERLU parvient à profiter du parallélisme de la machine cible. L'accélération reste presque parfaite tant qu'il n'y a pas plus de threads que de processeurs[2]. Au-delà, les performances s'écroulent lorsque l'on utilise un ordonnanceur générique tel que celui de Linux (avec la bibliothèque NPTL) ou de la bibliothèque MARCEL avec une politique d'ordonnancement *shared* (une seule liste de threads prêts est partagée par tous les processeurs), car ceux-ci essaient de respecter l'équité entre threads, et pour cela les font tourner brièvement tour à tour sur les différents processeurs, ce qui brise les affinités de cache des routines BLAS utilisées.

Si l'on considère maintenant une situation où une grande application scientifique multi-échelle a besoin d'effectuer des tâches de factorisation LU d'une manière irrégulière, il existe plusieurs manières de traiter ces tâches, selon le nombre de threads à lancer par tâche et la manière de les ordonnancer. Utiliser un simple ordonnanceur par lot (lançant chaque tâche à l'aide de 4 threads sur les 4 processeurs jusqu'à terminaison), ou à l'inverse utiliser un ordonnanceur complètement distribué (lançant chaque tâche avec un seul thread sur un seul processeur jusqu'à terminaison) ne sont pas nécessairement les méthodes les plus adaptées, lorsque d'autres parties parallèles de l'application (tournant sur d'autres machines) ont rapidement besoin des résultats de certaines tâches, par exemple.

Les figures 6.5(b) et 6.5(c) montrent les résultats obtenus en utilisant plusieurs approches.

[2]Sur la machine Hagrid, décrite à la page 14, possédant 8 puces dual core équivalentes (soit 16 processeurs), l'accélération obtenue reste cependant limitée à 7,6, probablement à cause de son facteur NUMA.

Sur la figure 6.5(b), toutes les tâches ont été lancées avec 4 threads, tandis que sur la figure 6.5(c), elles ont toutes été lancées avec 2 threads. Les deux figures montrent clairement que les ordonnanceurs génériques de Linux et de la bibliothèque MARCEL obtiennent des performances mauvaises et même erratiques (les barres min-max sont très hautes), car ils considèrent indifféremment tous les threads de toutes les tâches sans prendre en compte des notions d'affinités.

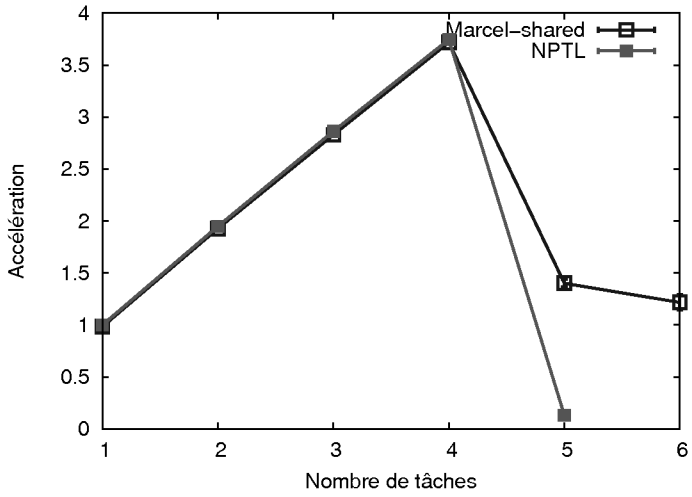
Nous avons donc utilisé la construction automatique de bulles de l'interface de compatibilité POSIX de MARCEL pour que les threads d'une même tâche (lancés dans SUPERLU par un même thread dans une simple boucle `for`) soient rassemblés au sein d'une même bulle, et nous avons alors utilisé l'ordonnanceur *gang* (détaillé à la section 3.2.2 page 46) pour ordonnancer ces tâches avec des tranches de temps de 0,2 ms (le temps typique de résolution d'une tâche avec un seul thread est de l'ordre de 10 s). Nous avons expérimenté en utilisant un seul ordonnanceur *Gang* pour l'ensemble de la machine 4 voies (courbes « Marcel-gang »), mais aussi en utilisant deux ordonnanceurs *Gang*, un par nœud NUMA 2 voies (courbes « Marcel-gang2 »).

- La courbe « Marcel-gang » en bas de la figure 6.5(c) montre que lancer un seul ordonnanceur *Gang* 4 voies pour des tâches à 2 threads limite évidemment l'accélération à 2.
- La courbe « Marcel-gang2 » en bas de la figure 6.5(b) montre que lancer des tâches à 4 threads sur des nœuds NUMA 2 voies produit également une accélération limitée (car il y a plus de threads que de processeurs, perdant ainsi les affinités de cache et de synchronisation).
- La courbe « Marcel-gang » en haut de la figure 6.5(b) montre que notre ordonnanceur *Gang* arrive assez bien à ordonnancer les tâches sur la machine : chaque tâche obtient une accélération d'à peu près $3,5$, quel que soit leur nombre.
- La courbe « Marcel-gang2 » de la figure 6.5(c) montre qu'utiliser deux ordonnanceurs *Gang* à deux voies permet d'obtenir un résultat similaire, pourvu qu'il y ait suffisamment de tâches bien sûr.

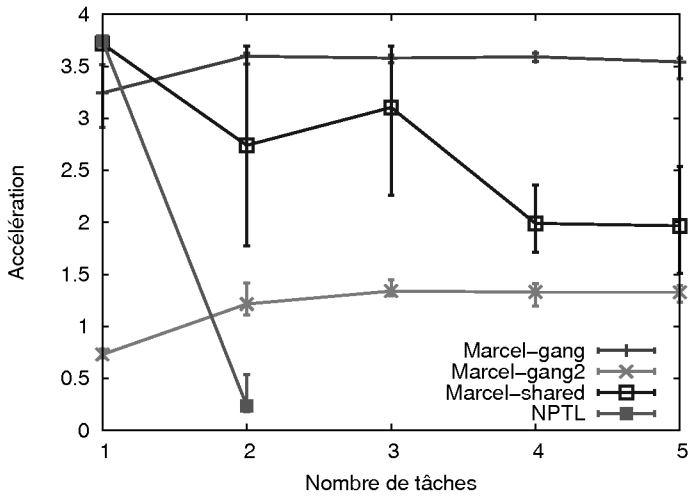
Au final, lorsque l'on compare de plus près ces deux dernières courbes, on peut constater que, pour cette application et sur cette machine, lancer un seul ordonnanceur *Gang* pour la machine entière permet en fait d'obtenir des performances légèrement meilleures qu'en lançant un ordonnanceur *Gang* sur chaque nœud NUMA. Cela est sans doute dû au fait que pour obtenir une meilleure répartition, nous avons choisi de laisser les ordonnanceurs *Gang* partager l'ensemble des tâches à traiter. Si l'application avait été légèrement différente (moins exigeante au niveau des caches, mais plus exigeante en termes d'accès mémoire), nous aurions observé l'inverse.

6.2.3 NPB BT-MZ, un parallélisme imbriqué irrégulier dans l'espace

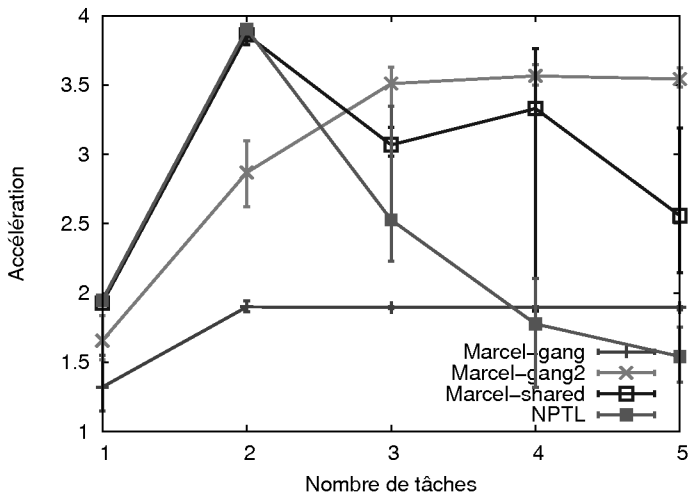
Parmi les applications de la suite de benchmarks parallèles de la NASA (NPB, *NAS Parallel Benchmarks*), les applications BT, LU et SP existent en version OPENMP multizone [WJ03], c'est-à-dire que le domaine de calcul est découpé en plusieurs zones qui sont par exemple distribuées sur différentes machines, où elles sont traitées de nouveau de manière parallèle. L'application BT est particulièrement intéressante car les tailles des zones sont irrégulières : la plus grande zone est typiquement 25 fois plus grande que la plus petite. De plus, le cas testé ne comporte que 16 zones. Il est donc essentiel d'effectuer un bon équilibrage de charge



(a) Accélération de la parallélisation brute.



(b) Exécution par tâches de 4 threads.



(c) Exécution par tâches de 2 threads.

FIG. 6.5 – Performances de la bibliothèque SUPERLU selon les stratégies d'ordonnancement.

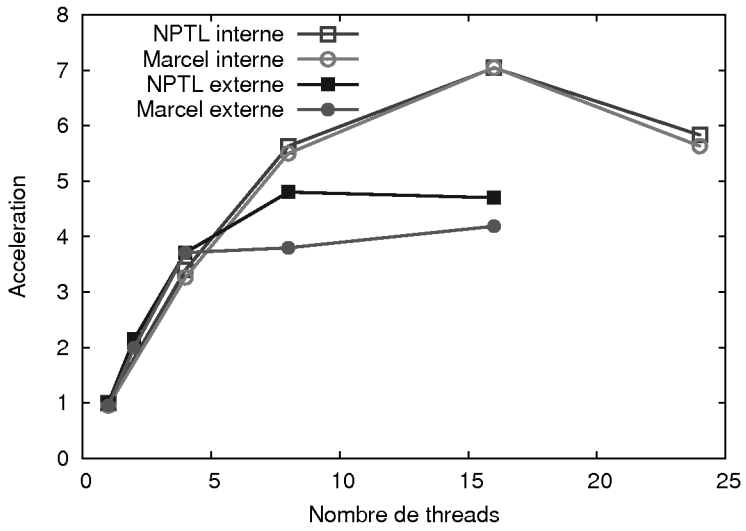


FIG. 6.6 – Parallélisme externe et parallélisme interne de l'application BT-MZ.

pour ne pas risquer d'obtenir une accélération limitée.

Cependant, comme indiqué à la section 5.2.1, la version OPENMP fournie sur le site de la NASA n'utilise pas d'imbrication de sections parallèles, mais lance plutôt plusieurs processus utilisant chacun une section parallèle, ce qui leur permet notamment d'utiliser une combinaison d'OPENMP et de MPI pour effectuer la parallélisation. Nous avons modifié cette version pour utiliser des sections parallèles imbriquées, nous permettant ainsi de contrôler le placement des threads au sein d'un même processus. Nous nous retrouvons ainsi avec une application comportant deux niveaux de parallélisme : un parallélisme *externe*, irrégulier, correspondant au découpage (x, y) du domaine en zones, et un parallélisme *interne*, régulier, correspondant à la parallélisation selon z de l'algorithme. Les résultats présentés ici ont été obtenus sur la machine Hagrid, octo-bicœur Opteron (donc composée de 8 nœuds NUMA de 2 cœurs, soit 16 cœurs en tout), détaillée à la page 14.

La figure 6.6 montre les résultats que l'on obtient si l'on active seulement un des deux niveaux de parallélisme. Si l'on n'exploite que le parallélisme *externe*, les zones elles-mêmes sont distribuées sur les différents processeurs. Puisque les tailles des zones varient fortement, l'accélération est limitée par le déséquilibre de charge de calcul dû aux zones les plus grandes. Lorsque l'on exploite le parallélisme *interne*, la répartition de charge est bonne, mais la nature même du calcul introduit de nombreux échanges de données entre processeurs. En particulier parce que la machine de test est une machine NUMA, l'accélération reste donc limitée à 7.

En combinant les deux niveaux de parallélisme, on peut espérer obtenir de meilleures performances en profitant des bénéfices des deux niveaux de parallélisme (localité et équilibrage de charge). Comme l'indiquent DURAN *et al.* [DGC05], l'accélération obtenue dépend du nombre relatif de threads créés au niveau du parallélisme externe et du nombre de sous-threads créés au niveau du parallélisme interne. Nous avons donc essayé toutes les combinaisons entre 1 et 16 threads pour le parallélisme externe (puisque ce parallélisme est de toutes façons limité aux 16 zones), et entre 1 et 8 sous-threads pour le parallélisme interne.

La figure 6.7(a) montre les résultats obtenus lorsque l'on utilise simplement la bibliothèque de threads de Linux, la NPTL. Lorsque l'on choisit 1 pour le parallélisme externe ou que l'on choisit 1 pour le parallélisme interne (ce qui revient à désactiver l'un ou l'autre), on retrouve évidemment les courbes de la figure 6.6. Lorsque l'on combine réellement les deux niveaux de parallélisme, l'accélération n'est en fait pas meilleure ($6,28$, contre $7,02$ obtenu avec le parallélisme interne seul), et elle retombe même à $1,92$! La bibliothèque MARCEL, dont les résultats sont montrés à la figure 6.7(b), s'en sort mieux parce que la création de threads est bien plus légère et que l'ordonnancement de base est moins agressif, mais elle n'obtient, au mieux, qu'une accélération de $8,16$. Ces accélérations limitées sont dues au fait que ni la NPTL ni MARCEL ne prennent en compte les affinités entre threads, si bien que les sous-threads sont répartis assez aléatoirement sur la machine, conduisant à de nombreux accès distants, des défauts de cache, etc. Nous avons donc introduit l'utilisation de bulles.

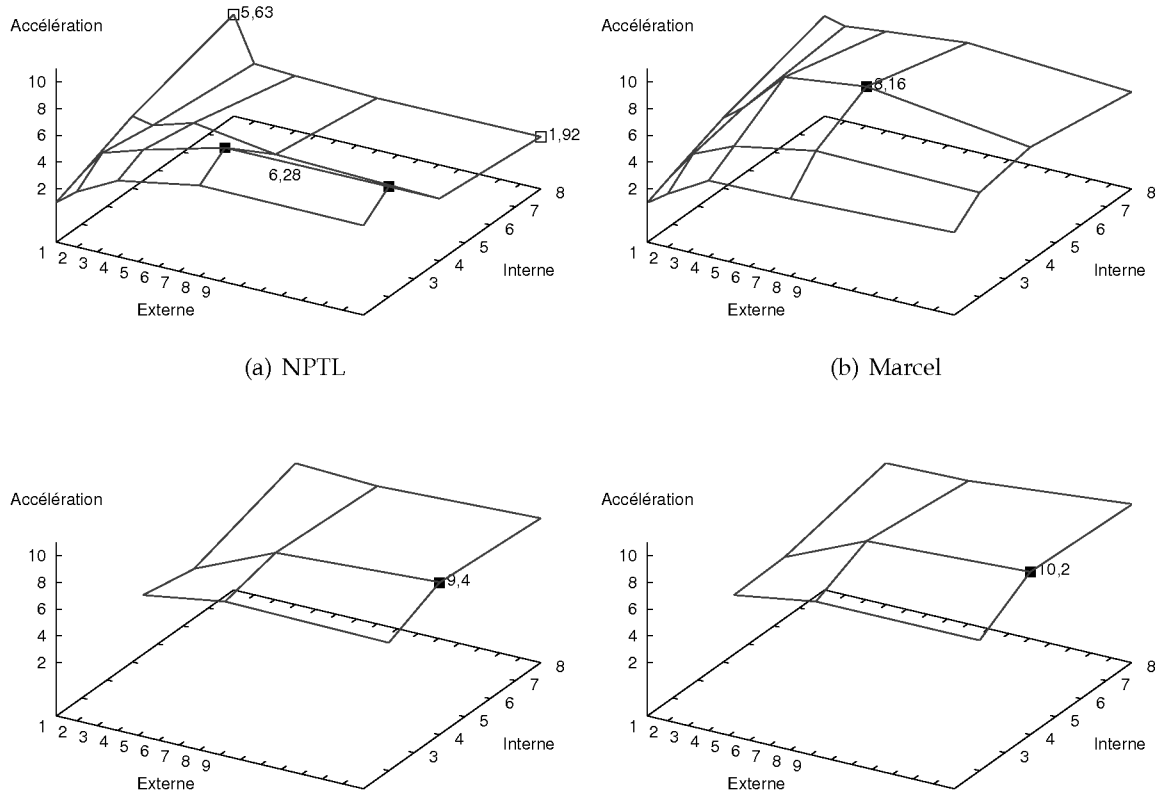
Comme décrit à la section 5.2.3 (page 81), une hiérarchie de bulles peut être construite automatiquement à partir de l'imbrication des sections parallèles. Cela conduit alors ici à créer pour chaque thread du parallélisme externe (traitant donc un certain nombre de zones) une bulle qui contient les sous-threads traitant le parallélisme interne associé à ce thread. Puisque la charge de calcul des différentes zones est déséquilibrée mais connue, nous avons ajouté une ligne de code dans l'application pour qu'elle indique cette charge. Nous avons alors utilisé l'ordonnanceur à bulles *Spread* pour répartir cette hiérarchie de bulles sur la machine tout en prenant en compte leur charge de calcul. Le résultat est visible sur la figure 6.7(c) : grâce au placement judicieux des threads selon leurs affinités, l'accélération passe à $9,4$. Cependant, en observant le comportement de l'ordonnanceur *Spread* à l'aide de notre outil d'animation de traces décrit à la section 3.3.2, nous avons réalisé que, parce que cet ordonnanceur fait descendre tous les threads jusqu'aux cœurs, cela induit un certain déséquilibre de charge. Nous avons donc réglé cet ordonnanceur pour qu'il ne fasse descendre les threads que jusqu'aux huit nœuds NUMA, et le résultat est visible sur la figure 6.7(d) : l'accélération devient encore meilleure, $10,2$.

Avec cet exemple, on s'aperçoit donc bien qu'exploiter plusieurs niveaux de parallélisme peut réellement apporter un gain de performances substantiel, à condition que l'environnement d'exécution utilise un ordonnanceur de threads adéquat.

6.2.4 MPU, un parallélisme imbriqué irrégulier dans le temps

Les dernières technologies d'acquisition 3D permettent de numériser des objets entiers sous forme de nuages de points. Le projet *Digital Michelangelo* a, par exemple, pour but de numériser les sculptures de Michel-Ange avec une précision suffisante pour pouvoir distinguer les coups de ciseau du sculpteur. Ces numérisations comportent typiquement des milliards de points. Pour pouvoir effectuer efficacement un bon rendu de l'objet, il est courant de reconstruire une surface à partir d'un tel nuage de points. L'algorithme utilisé au LABRI par Tamy BOUBEKEUR [BRS04] au sein de l'équipe IPARLA est MPU (*Multi-Level Partition of Unity*) [YAM+03], dont le principe dans le cas 2D est expliqué à la figure 6.8. La figure 6.9 montre le code C++ de la boucle principale de MPU : c'est simplement une fonction qui effectue une approximation sur un cube, et si celle-ci n'est pas suffisamment bonne, subdivise le cube en 8 sous-cubes pour lesquels elle s'appelle récursivement.

La parallélisation évidente de cet application à l'aide d'OPENMP revient alors à ajouter le



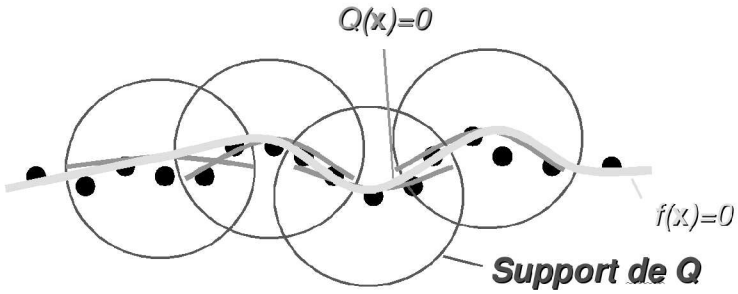
(a) NPTL

(b) Marcel

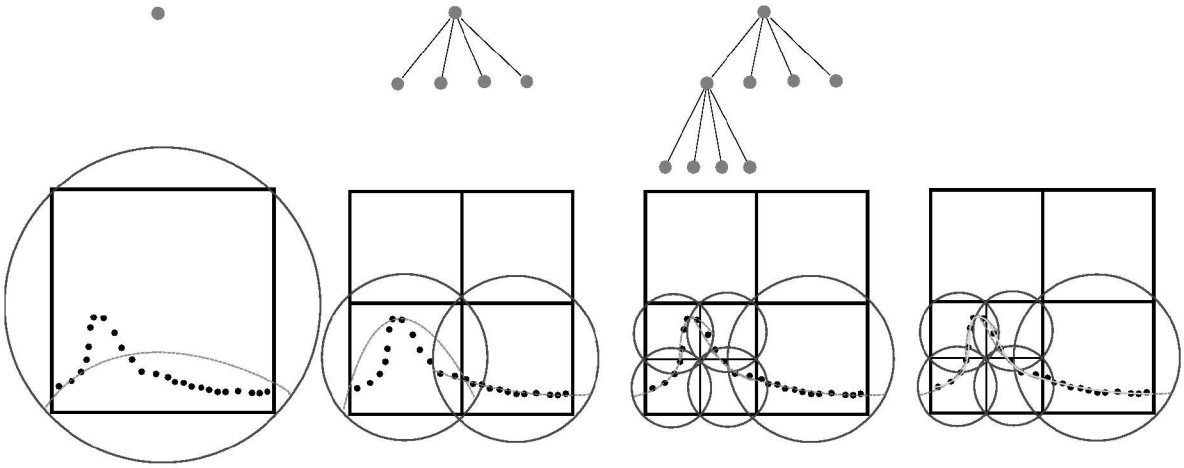
(c) Bulles

(d) Bulles avec ordonnancement optimisé

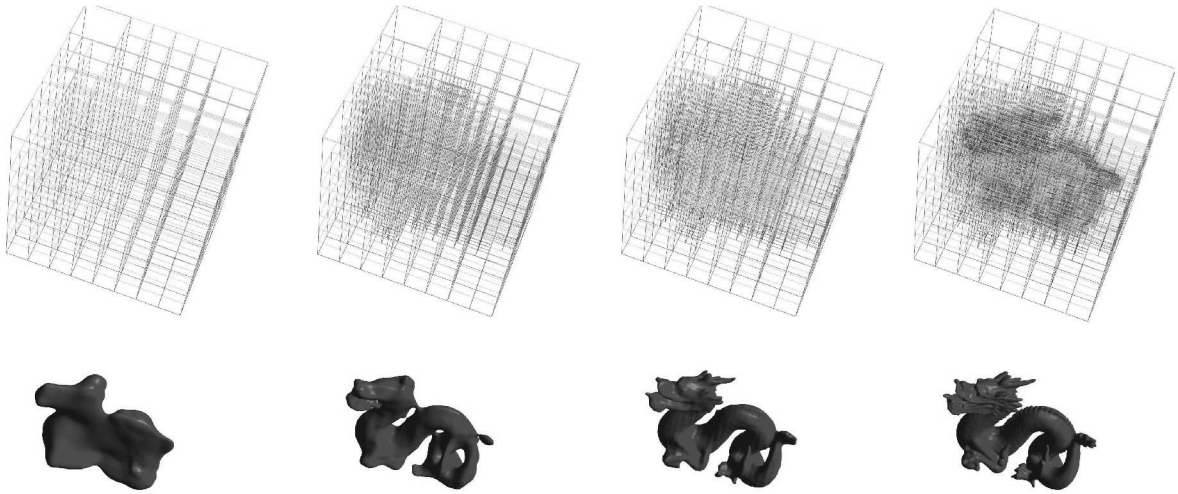
FIG. 6.7 – Parallélisme imbriqué.



(a) L'objectif est d'approcher implicitement un nuage de points (en noir) à l'aide d'une équation $f(x) = 0$ (en jaune). Le principe consiste à approximer localement une partie du nuage de points à l'aide d'une quadrique : pour chaque ensemble de points cerclé en rouge, une quadrique $Q(x) = 0$ est calculée (en vert). En sommant ces quadriques, on obtient une équation globale $f(x) = 0$.



(b) En pratique, des quadriques sont d'abord calculées pour les points compris dans des sphères englobant les cubes d'un maillage grossier. Les quadriques qui approximent suffisamment bien le nuage de point (c'est le cas en bas à droite) sont conservées telles quelles. Sinon, le maillage est raffiné localement, et de nouvelles quadriques sont calculées, comme illustré en bas à gauche. L'algorithme itère ainsi localement tant que l'approximation n'est pas jugée suffisante. La fonction globale est alors la somme de cette hiérarchie de quadriques.



(c) Selon la finesse d'approximation voulue, on effectue ainsi au final un raffinement hiérarchique du maillage adapté à la géométrie de l'objet, ici en 3 dimensions.

FIG. 6.8 – Principe de la partition de l'unité multi-niveau (MPU).

```
void ImplicitOctCell::buildFunction() {
  /* Approximation sur le cube. */
  computeLA();
  if (error < maxError)
    return;


  /* Approximation trop grossière, raffiner. */
  /* Supprimer l'approximation. */   deleteLA();
  /* Découper en sous-cubes. */   Split();
#pragma omp parallel for
  for (int i=0; i<8; i++)
    subCell[i]->buildFunction();
}
```

FIG. 6.9 – Boucle principale de l'application MPU.

`pragma` indiqué en gras. Les résultats obtenus avec la bibliothèque de threads de Linux, la NPTL, sont cependant loin d'être convaincants. Si l'on n'active pas le support des sections parallèles imbriquées, *i.e.* que seule la section parallèle la plus externe crée des threads, le parallélisme est limité à 8 voies, et la répartition de charge est en général plutôt déséquilibrée, si bien que même sur la machine Hagrid et ses 16 processeurs, l'accélération reste limitée à environ 5, comme on peut le voir sur la figure 6.11. Si l'on active le support des sections parallèles imbriquées, la répartition de charge est meilleure, mais Linux ne sait guère comment répartir la pléthore de threads ainsi créés (l'arbre de récurrence a typiquement une profondeur de 15 et une largeur de plusieurs centaines de threads), et l'accélération est, de fait, limitée à environ 4.

Durant son stage de Master 2 [Dia07] au sein de l'équipe, François DIAKHATÉ a développé sa propre version parallèle de MPU. Le principe consiste à lancer un thread par processeur, et de maintenir pour chacun d'entre eux une liste de cubes à traiter, implémentée à l'aide d'un *deque*, décrit à la figure 6.10. Lorsque le processeur local raffine un cube, il empile les sous-cubes de son côté de son propre *deque* et en pioche un, pendant que d'autres processeurs inactifs volent éventuellement depuis l'autre côté. Ainsi, les cubes des *deques* restent triés par ordre de taille, le processeur local piochant toujours un des plus petits, et les autres processeurs volant toujours un des plus gros. Cela permet d'assurer une bonne localité d'exécution du côté du processeur local (puisqu'au final il parcourt la hiérarchie de cubes en profondeur), et de limiter les vols des autres processeurs (puisqu'ils volent les cubes les plus gros). Les résultats obtenus sont bien meilleurs : comme on peut le voir avec la courbe *Manuel* de la figure 6.11, à l'aide des 16 processeurs de la machine Hagrid, l'accélération atteint 15,5 !

Ce très bon résultat a cependant nécessité des développements assez importants qui ne peuvent pas être facilement réutilisés pour d'autres applications. Il serait plus intéressant d'obtenir un résultat similaire sans modification de l'application (ou très peu). De nouveau, nous utilisons la construction automatique (décrite en section 5.2.3 page 81) d'une hiérarchie de bulles à partir de l'imbrication des sections parallèles. Le résultat est qu'à chaque cube correspond une bulle qui contient les threads travaillant sur les sous-cubes de ce cube. Puisque l'on ne connaît pas *a priori* la charge des threads et bulles, il vaut mieux utiliser l'or-

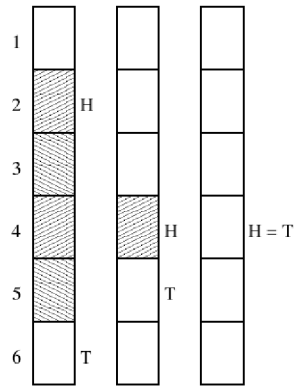


FIG. 6.10 – Principe de fonctionnement d'un *deque* du protocole THE simplifié. Le processeur local empile et dépile les cubes à traiter de manière LIFO du côté T pendant que les autres processeurs peuvent voler du côté H. Ces opérations peuvent être effectuées sans verrouillage tant que H≠T.

donnanceur *Affinity*, pour au moins privilégier les affinités entre threads et utiliser un vol de travail pour équilibrer la charge. De plus, puisque la quantité de calcul effectuée par les threads traitant de très petits cubes est potentiellement très faible, nous activons l'utilisation des graines de threads. En pratique, l'ordonnancement obtenu est alors très proche de celui effectué à la main par François DIAKHATÉ. En effet, la partie haute de la hiérarchie de bulles se retrouve rapidement distribuée sur la machine, et sur chaque processeur la partie basse de la hiérarchie de threads qui s'y retrouve est exécutée par un parcours en profondeur, car les graines de threads générées par l'entrée dans une section parallèle sont mises en tête de la liste locale. De plus, lorsqu'un processeur est inactif, il vole du travail à un autre processeur en prenant en compte la structure de bulle, *i.e.* il vole si possible une bulle de niveau le plus externe, et donc une quantité de travail la plus potentiellement conséquente possible, ce qui revient bien au vol de travail de l'ordonnancement manuel. Le résultat est visible sur la figure 6.11 : sans être équivalente (la gestion des graines de threads et des bulles est bien plus lourde que celle des tâches de l'ordonnancement manuel), l'accélération obtenue avec un ordonnancement à bulles se rapproche de celle obtenue avec un ordonnancement manuel, pour un temps de développement très réduit pour le programmeur d'application, puisqu'il lui a suffit d'ajouter une directive OPENMP, de choisir l'ordonnanceur *Affinity* et d'activer l'utilisation de graines de threads.

Une fois l'approximation du nuage de point effectuée, l'application MPU utilise une deuxième étape de construction d'un ensemble de polygones, détaillée à la figure 6.12. Il n'y a pas ici de structure hiérarchique qu'il serait véritablement naturel d'exprimer à l'aide de bulles, nous n'avons donc pas envisagé pour cette étape d'algorithme à bulles particulier, et répartissons donc simplement les threads sur chaque processeur. La parallélisation de cette étape a de toutes façons nécessité un certain travail qui se solde déjà par une accélération de presque 15. De telles performances n'ont pas réellement besoin d'être améliorées. Il est cependant important de noter ici l'utilité de pouvoir indiquer depuis l'application un changement d'étape : l'environnement d'exécution peut alors changer d'ordonnanceur, pour passer ici de l'ordonnanceur complexe *Affinity* à un ordonnancement bien plus simple, plus adapté à cette deuxième étape.

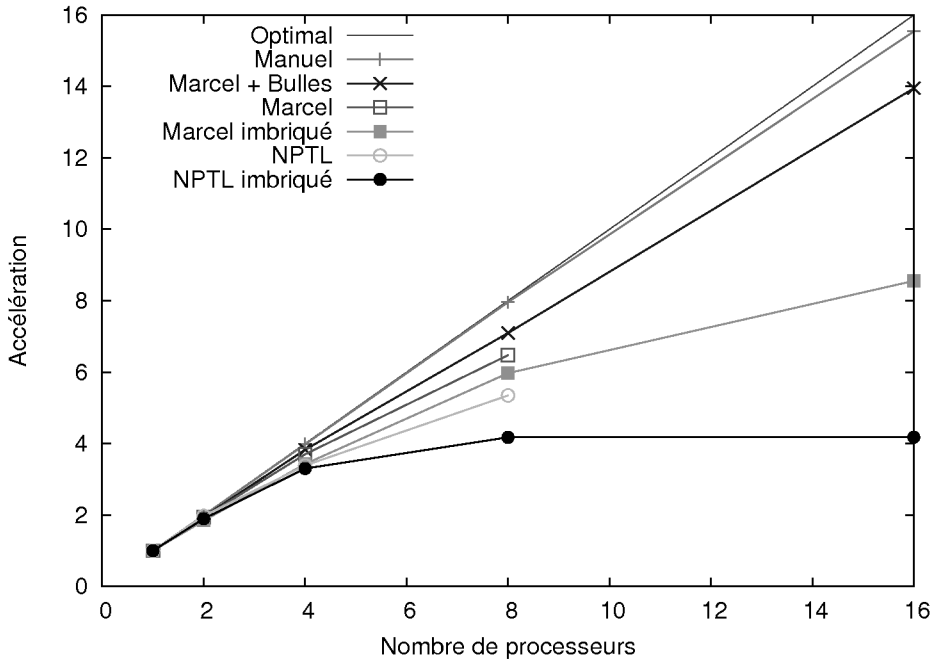
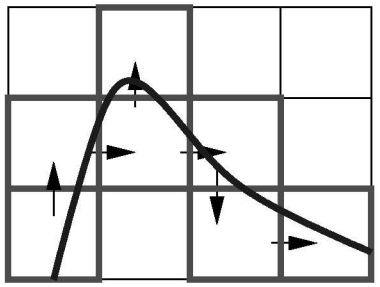
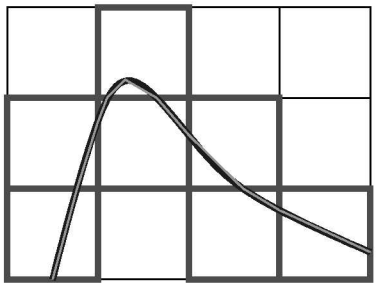


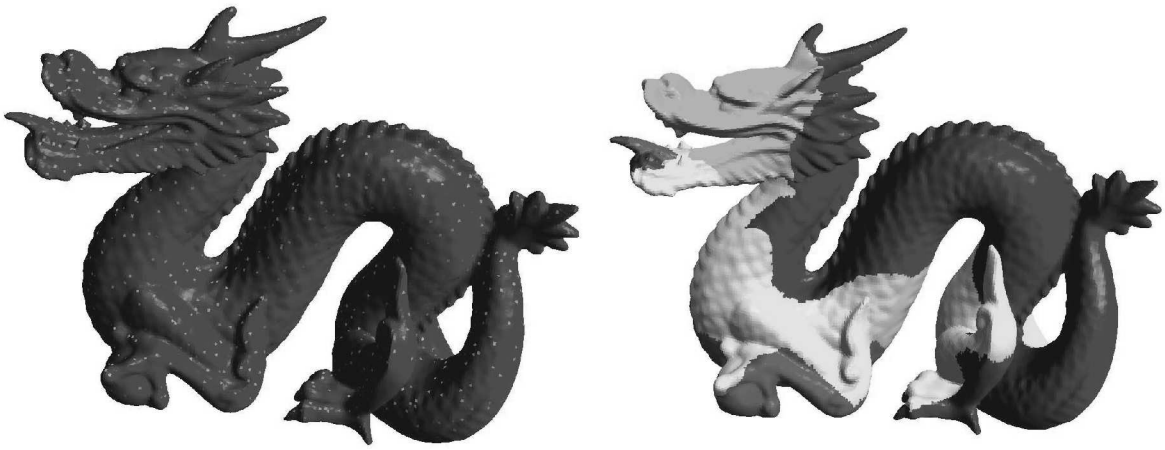
FIG. 6.11 – Accélération de l'application MPU selon l'environnement d'exécution utilisé.



(a) Parcours de la surface à travers la grille.



(b) Résultat.



(c) Ensemble des cubes graines et distribution sur les processeurs.

FIG. 6.12 – Construction d'un ensemble de polygones à l'aide de l'algorithme *Marching-Cube*. Le principe est que chaque processeur part d'un point de l'objet (une graine), et suit la surface implicite au travers d'une grille uniforme. Au sein de chaque cube de la grille, on approxime la surface à l'aide de polygones. Un protocole léger s'assure que deux processeurs ne se gênent pas lorsqu'ils traitent de cubes voisins, produisant ainsi une *frontière* entre les domaines traités par chaque processeur. Lorsqu'un processeur ne peut plus progresser, il pioche une autre graine.

Chapitre 7

Conclusion et perspectives

Les technologies permettant d'intégrer à faible coût de nombreux processeurs au sein d'une même machine (architecture NUMA, multicœur, SMT) se sont démocratisées, comme le montre le choix d'AMD d'une architecture combinant nœuds NUMA et puces multicœurs pour ses machines OPTERON, et le choix d'INTEL d'une combinaison des technologies multicœur et SMT (*HyperThreading*) et bientôt NUMA. Il est ainsi devenu essentiel, pour exploiter au mieux ces ressources de calcul, de prendre en compte cette structuration hiérarchique. Bien sûr, il n'est pas envisageable de faire remonter cette complexité au niveau du programmeur final d'application, il est nécessaire de passer par des abstractions à la fois de la structure du parallélisme de l'application et de l'architecture de la machine cible, ne serait-ce que pour des raisons de portabilité. Cela est d'autant plus vrai que les applications utilisent de plus en plus souvent des schémas irréguliers, pour affiner les simulations sans faire exploser le temps de calcul par exemple. Pour pouvoir être pris en compte lors de l'ordonnancement, il est nécessaire que de tels raffinements puissent être exprimés d'une manière rationnelle.

De telles abstractions font assez cruellement défaut dans le standard le plus courant, l'interface de threads POSIX. Les ordonnanceurs génériques des systèmes d'exploitation n'ont ainsi que peu d'informations fiables (ou précises) sur les applications de calcul scientifique pour pouvoir les exécuter de manière réellement efficace sur machine hiérarchique. D'autres environnements de programmation plus spécialisés tels que le standard OPENMP ou les *Thread Building Blocks* (TBB) d'INTEL mettent en jeu des abstractions évoluées, mais aucune implémentation actuelle ne les utilise réellement en rapport avec la structure de la machine exécutant l'application.

Ainsi, il ressort un manque de lien entre l'*environnement de programmation* qui dispose souvent de concepts parallèles de haut niveau (sections parallèles pour OPENMP, espaces d'itération pour les TBB), et l'*environnement d'exécution* utilisé, que celui-ci soit directement embarqué au sein de l'application produite par le compilateur, ou que ce soit le système d'exploitation qui est chargé de l'ordonnancement (or lui a connaissance de la machine cible utilisée).

7.1 Contributions

Les travaux effectués durant cette thèse se proposent de combler ce manque à l'aide d'un concept au départ simple mais finalement puissant, la notion de *bulle*, qui permet de regrouper de manière récursive les tâches d'une application et d'être un support d'informations utiles à un ordonnanceur, telles qu'une estimation relative de la charge de calcul. Les environnements de programmation sont à même de générer des hiérarchies de bulles et d'apporter ces informations. D'autre part, les machines hiérarchiques sont modélisées au sein de l'environnement d'exécution à l'aide d'une hiérarchie de listes de tâches, chaque liste étant associée à un élément matériel de la machine (nœud NUMA, puce, cœur, etc.), si bien que placer une tâche sur une liste revient à indiquer qu'elle pourra être exécutée par n'importe quel processeur appartenant à l'élément matériel correspondant. Ces deux modélisations permettent alors d'abstraire également la notion d'ordonnanceur à un haut niveau : loin de parler d'aspects techniques tels que le changement de contexte, il s'agit désormais de distribuer bulles et tâches sur la hiérarchie de listes, ce qui fait ainsi le lien entre la structure de l'application et la structure de la machine.

Pour ce faire, il est fourni aux experts en ordonnancement une interface de programmation qui permet d'implémenter des stratégies de distribution indépendantes de la structure précise de la machine, ce qui leur permet donc d'être *portables*. Plusieurs ordonnanceurs variés ont été développés et montrent qu'il est ainsi possible d'exprimer différentes approches d'ordonnancement : distribution opportuniste, ordonnancement par lot, équilibrage de charge, vol de travail, respect des affinités entre différentes tâches et entre tâches et données. Certains de ces ordonnanceurs ont en fait été écrits par des stagiaires encadrés au sein de l'équipe. Enfin, sont fournis des outils de débogage sous forme d'animations reproduisant de manière graphique l'ordonnancement effectué durant une exécution réelle, permettant ainsi de déterminer les raisons de contre-performances.

Par ailleurs, bien que l'on puisse construire une hiérarchie de bulles manuellement et y attacher des informations depuis l'application, nous fournissons en plus un moyen de construction automatique selon l'environnement de programmation utilisé : à partir des relations père-fils dans le cas des threads POSIX, ou selon les sections parallèles emboîtées dans le cas d'OpenMP. Une application peut ainsi être ordonnancée avec des bulles sans aucune modification.

Enfin, d'une part des tests synthétiques permettent de vérifier que l'implémentation de ces concepts au sein de la bibliothèque de threads utilisateur Marcel n'apporte qu'un surcoût réduit. D'autre part, des mesures de performances de différents types d'applications montrent que les différents algorithmes à bulles implémentés peuvent répondre à différents besoins. Le gain de performances obtenu est alors substantiel, de l'ordre de 20 à 40%.

Il est à noter que nous avons volontairement choisi de limiter les modélisations des applications et machines à une structuration en arbre plutôt que d'utiliser la généricité d'une structure de graphe. Ce choix permet en effet d'une part d'obtenir une implémentation plus simple, mais aussi de conserver en général des algorithmes polynomiaux. Au besoin, il est possible d'approximer un graphe par un arbre couvrant.

7.2 Perspectives

Ces travaux, en proposant un modèle original pour ordonnancer l'exécution d'une application sur une machine hiérarchique, ouvrent de nombreuses perspectives.

7.2.1 Toujours plus d'informations

À court terme, il sera utile d'enrichir les informations associées aux bulles.

En effet, un compilateur OpenMP serait à même de donner par exemple des informations sur les accès aux données : quels threads accèdent à quels tableaux et à quelle fréquence. Cela permettrait aux ordonnanceurs à bulles d'avoir des informations précises sur les zones mémoire utilisées, et de décider alors quels threads sont les moins coûteux à faire migrer. Au besoin, il serait possible d'enrichir la syntaxe d'OpenMP pour que ce soit le programmeur lui-même qui puisse fournir de telles informations, de manière similaire au quatrième paramètre de la construction `forall` du langage UPC qui permet de préciser quelle donnée est *a priori* la plus importante pour chaque thread. Cette information n'est pas nécessairement statique, elle peut très bien être calculée dynamiquement lors de l'exécution. Tout cela est également valable pour de nombreux autres paramètres : estimation de la charge de calcul, quantité de mémoire allouée, utilisation de la bande passante mémoire, etc.

Par ailleurs, il serait possible d'utiliser les compteurs de performances intégrés aux processeurs pour obtenir des informations sur la qualité de l'exécution actuelle de l'application. Un ordonnanceur à bulles pourrait ainsi par exemple détecter un taux anormal de défauts de cache, et décider alors de distribuer les threads en cours d'exécution sur un plus grand nombre de processeurs, pour que ces threads ne se gênent pas mutuellement dans des caches partagés.

L'interface utilisateur des verrous pourrait également être enrichie, pour que l'ordonnanceur puisse savoir si un verrou est utilisé simplement de manière courte pour protéger l'accès à une ressource partagée, ou si un thread qui essaie de le prendre devra attendre potentiellement longtemps. En effet, lorsqu'un verrou n'est utilisé que pour protéger une ressource, si celle-ci n'est pas disponible immédiatement et que le thread est donc obligé d'attendre, il n'est sans doute pas nécessaire d'effectuer un rééquilibrage de charge pour autant : bien souvent, il est même plus intéressant de laisser le processeur inactif, ou de lui faire exécuter un thread attribué à un autre processeur, en attendant que la ressource se libère (ce qui survient rapidement), et continuer alors comme si le processeur avait effectué un travail utile, plutôt que de chambouler l'équilibrage actuel au risque de briser les affinités de cache existantes.

7.2.2 Des ordonnanceurs variés

À plus long terme, il sera bien sûr utile de continuer à développer différents ordonnanceurs spécialisés dans différents aspects de l'ordonnancement.

Il sera par exemple intéressant d'étendre les algorithmes décrits à la section 3.2 pour qu'ils puissent notamment prendre en compte les informations décrites ci-dessus pour affiner leurs prises de décision de migration de threads et de données.

D'autre part, il serait par exemple utile de développer un ordonnancement à bulles adapté aux applications mettant en œuvre la technique de raffinement de maillage (AMR). Dans ce cadre, il sera utile d'exiger que l'application indique précisément à quel moment elle raffine son domaine pour que l'ordonnanceur sache quel est le bon moment pour modifier la répartition de threads et bulles, ainsi que prendre éventuellement le temps de migrer des données.

Pour pouvoir faciliter le développement d'ordonnanceurs, il serait sans doute bon de se détacher du langage actuellement utilisé, C. Les concepts de haut niveau mis en jeu se prêtent en effet *a priori* assez bien à être manipulés avec des langages de plus haut niveau. On peut songer par exemple à des langages interprétés, éventuellement ML. La lenteur éventuelle des implémentations de tels langages n'est pas nécessairement une contrainte, si les ordonnanceurs développés n'ont pas besoin d'intervenir très souvent durant l'exécution. On pourrait même imaginer d'écrire un ordonnancement sous la forme d'un ensemble de contraintes, de manière similaire à Prolog, et d'utiliser un mini-ordonnanceur, écrit dans un langage de plus bas niveau, qui essaierait en permanence de tendre au mieux vers le respect de ces contraintes. Cela pourrait permettre d'exprimer par exemple « distribuer tel ensemble de threads, mais séparer le plus possible ces deux threads ».

Enfin, il sera opportun d'essayer de combiner ces ordonnanceurs de différentes manières selon les applications. Tel qu'expliqué à la section 3.3.1 (page 58), il est au moins possible de les combiner dans le temps, dans l'espace ou de manière hiérarchisée. Il pourra être approprié de pouvoir exprimer cela depuis l'environnement de programmation, éventuellement de manière explicite depuis l'application. Ainsi, une routine de calcul pourra exprimer, lorsqu'elle crée une équipe de threads, quel type d'ordonnancement il est préférable de lui appliquer. Dans le cas de l'application MPU, nous avons vu qu'elle était clairement composée de deux étapes dont les besoins d'ordonnancement étaient très différents, il devrait pouvoir être possible de l'exprimer.

7.2.3 Au-delà du modèle hiérarchique de processeurs

La modélisation que nous avons choisie pour représenter les machines, une hiérarchie de listes associées aux nœuds NUMA, caches partagés, etc. est bien sûr une simplification de la réalité.

En premier lieu, il serait utile d'étendre légèrement la modélisation des machines hiérarchiques. En effet, certaines machines telles que les SGI ALTIX ont un réseau d'interconnexion en forme de *tore 2D*, et le prototype de puce embarquant 80 cœurs présenté par INTEL utilise un réseau d'interconnexion en forme de grille. Il serait ainsi utile d'étendre au moins à 2 dimensions la hiérarchie de listes de tâches pour modéliser plus fidèlement de telles machines. Cela ne pose pas de réel problème technique, mais il faudrait alors adapter les algorithmes à bulles pour qu'ils profitent de cette nouvelle information de topologie.

D'autre part, les machines à architectures NUMA exhibent en général aussi des caractéristiques NUIOA (*Non-Uniform Input/Ouput Access*), c'est-à-dire que, par exemple, la latence de réception de données depuis une carte réseau dépend du processeur qui effectue la réception. Lorsque plusieurs processeurs utilisent en même temps plusieurs cartes réseaux, les débits des communications peuvent également se retrouver limités par la bande passante du

réseau d'interconnexion des processeurs. Ainsi, il pourrait être utile de prendre garde au placement des threads de communication par rapport aux cartes réseaux de la machine. Il apparaît donc ici important de savoir quels threads utiliseront le plus des ressources réseau. De nouveau, cette information pourrait être fournie par le programmeur ou automatiquement par le compilateur. Elle pourrait également être collectée automatiquement par la bibliothèque de communication utilisée, en observant simplement quels threads effectuent le plus d'appel à cette bibliothèque. Le placement des threads concernés pourrait alors lui-même apporter des contraintes de placement des threads de calcul, pour que les communications au sein même de la machine par mémoire partagée s'effectuent aussi sans interférence, etc. En quelque sorte, ces contraintes représentent des « élastiques » pour la hiérarchie de bulles.

De plus, il est de plus en plus courant de confier des tâches de calcul à des co-processeurs. Ceux-ci peuvent être spécialisés comme ceux des cartes accélératrices CLEARSPEED, plus ou moins génériques tels que les SPUs du processeur CELL/BE d'IBM, ou encore *a priori* prévus pour d'autres usages, tels que ceux des cartes graphiques. Tous ces cas soulèvent cependant de nouveau des problèmes de localité. En effet, les effets NUIOA décrits au paragraphe précédent ont également un impact sur le transfert des données qui seront traitées par les co-processeurs. Une répartition de charge sur plusieurs cartes devra ainsi prendre en compte le positionnement actuel des données sur les différents nœuds de la machine pour limiter les transferts coûteux. Par ailleurs, lorsque des données ont été transférées, il est souvent possible de leur appliquer plusieurs traitements sans avoir à effectuer d'autres transferts. De telles situations sont bien sûr à privilégier, en rassemblant par exemple les tâches qui sont liées les unes aux autres par des dépendances de données. À l'inverse, il se peut qu'en raison d'une certaine distribution de charge et d'un certain placement de données, il soit plus intéressant d'exécuter simplement la tâche sur le processeur courant, plutôt que de prendre le temps d'envoyer des données à un co-processeur lointain !

Par ailleurs, les très grands calculateurs sont assez souvent utilisés par plusieurs utilisateurs à la fois. Le système d'exploitation effectue alors un partage équitable des ressources de calcul pour ces différents utilisateurs. D'une part, il pourrait être intéressant pour effectuer ce partage d'utiliser des algorithmes à bulles, les utilisateurs et leurs processus représentant une famille de hiérarchies de bulles, que l'on pourrait donc préférer ordonnancer par lot, distribuer sur la machine, etc. D'autre part, du point de vue d'une application qui embarque son propre environnement d'exécution, il serait souhaitable d'enrichir le retour d'information du système d'exploitation. De manière similaire au mécanisme de *Scheduler Activations*, celui-ci pourrait ainsi prévenir l'application qu'un des processeurs qu'elle était en train d'utiliser a été attribué (pour longtemps) à une autre application. L'environnement d'exécution pourrait alors réagir en redistribuant les threads qu'il avait *a priori* attribué à ce processeur. Disposer d'une information aussi précise permettrait en effet à un ordonnanceur à bulles d'effectuer cette redistribution de manière cohérente entre l'organisation hiérarchique de la machine et la hiérarchie de bulles et de threads de l'application.

Enfin, des systèmes à image unique (SSI) tels que KERRIGHED permettent d'exploiter de manière transparente plusieurs machines connectées en réseau comme si elles formaient une seule machine avec cohérence mémoire. Techniquement parlant, il suffit d'ajouter un niveau « réseau » à la hiérarchie de listes de tâches, mais du point de vue des ordonnanceurs, il sera essentiel de prendre en compte le coût *a priori* énorme que représente la migration d'un thread ou de portions de mémoire d'une machine à une autre.

7.2.4 Diffusion du concept

Enfin, nombreux sont les environnements de programmation et environnements d'exécution qui pourraient s'appuyer avec profit sur des abstractions similaires à celles que nous avons développées.

Cette thèse a montré une ébauche d'intégration au sein d'un environnement de programmation de l'extension OPENMP, GOMP, ce qui a permis d'utiliser l'environnement d'exécution MARCEL pour ordonnancer quelques applications et obtenir des gains significatifs. Il serait donc intéressant d'utiliser cette même approche pour utiliser MARCEL comme support d'exécution d'autres environnements de programmation, tels que les TBB. Par ailleurs, il serait souhaitable de combler réellement le manque d'abstraction dans la norme POSIX, en proposant une extension de l'interface de threads standard pour que les nombreux environnements de programmation reposant dessus puissent exprimer un minimum d'informations sur les affinités entre threads, par exemple. Il serait alors possible d'améliorer les ordonnanceurs génériques actuels des systèmes d'exploitation pour qu'ils combinent ces informations avec la structure hiérarchique de la machine pour améliorer les performances obtenues.

Il serait également possible d'essayer d'ajouter des abstractions telles que les bulles au sein d'autres types d'environnements d'exécution, non nécessairement basés sur l'ordonnancement de threads, en effectuant des regroupements dans le traitement de requêtes sur une base de données par exemple.

À long terme, on voit pointer à l'horizon ce qui est souvent appelé des « mers de cœurs ». La société INTEL a effectivement annoncé la donne en Septembre 2006 en dévoilant un prototype de processeur 80 cœurs qu'elle espère mettre en production en 2011 ! Une manière d'exploiter des puces embarquant autant de cœurs est évidemment de lancer autant d'applications, mais on pourra préférer utiliser tous ces cœurs au sein d'une même application pour voir son exécution accélérée en conséquence. Cela semble cependant ardu, car si certains problèmes se parallélisent plutôt bien et l'accélération est effectivement au rendez-vous, la plupart du temps la parallélisation n'offre qu'une accélération limitée. Nous avons vu, dans le cas de l'application BT-MZ, qu'utiliser le parallélisme imbriqué permettait de combiner différents niveaux de parallélisme pour dépasser de telles limites. Il me semble que c'est là une approche qu'il faut envisager le plus possible si l'on veut pouvoir inonder efficacement une mer de cœurs par une « mer de threads ». Par exemple, les fonctions de bibliothèques de calcul scientifique devraient idéalement toujours exhiber du parallélisme, même lorsqu'elles sont elles-mêmes utilisées au sein de sections parallèles, qui elles-mêmes peuvent être imbriquées, etc. On devrait ainsi, à tous les niveaux de briques logicielles, s'efforcer d'exhiber un parallélisme sans avoir à se soucier de l'architecture de la machine cible, puisque ces briques seront potentiellement assemblées de manière dynamique et irrégulière. Bien sûr, une telle profusion de parallélisme doit être gérée de manière efficace, et rares sont les environnements d'exécution courants actuels qui savent réellement faire cela. Cette thèse a cependant montré qu'à l'aide notamment de graines de threads et d'ordonnanceurs à bulles il était possible de réaliser un ordonnancement efficace s'adaptant automatiquement à la structure de la machine cible tout en n'exigeant de l'application que d'exprimer sa structure. Il me semble donc que des développements adéquats autour du support du parallélisme imbriqué sont à privilégier.

Annexe A

Interface de programmation des bulles

Sommaire

A.1 Interface de programmation utilisateur des bulles 112
A.2 Interface de programmation d'ordonnancement des bulles 114
A.3 Interface d'accès aux statistiques 115
A.4 Conteneurs et entités 117

Cette annexe regroupe un extrait de la documentation de l'interface de programmation des bulles, extraite automatiquement du code source. La dernière version est disponible sur `http://pm2.gforge.inria.fr/marcel/html/`.

A.1 Interface de programmation utilisateur des bulles

Cette section regroupe les fonctions fournies pour que l'application puisse décrire sa structure à l'aide de bulles. La dernière version est disponible sur `http://pm2.gforge.inria.fr/marcel/html/group__marcel__bubble__user.html`.

Typedefs

- typedef struct **marcel_bubble marcel_bubble_t**
  Type of a bubble.

Functions

- int **marcel_bubble_init** (**marcel_bubble_t** *bubble)
  Initializes bubble bubble.
- int **marcel_bubble_destroy** (**marcel_bubble_t** *bubble)
  Destroys resources associated with bubble bubble.
- int **marcel_bubble_setid** (**marcel_bubble_t** *bubble, int id)
  Sets an id for the bubble, useful for traces.
- int **marcel_bubble_insertentity** (**marcel_bubble_t** *bubble, **marcel_entity_t** *entity)
  Inserts entity entity *into bubble* bubble.
- int **marcel_bubble_insertbubble** (**marcel_bubble_t** *bubble, **marcel_bubble_t** *little_bubble)
  Inserts bubble little_bubble *into bubble* bubble.
- int **marcel_bubble_insertthread** (**marcel_bubble_t** *bubble, marcel_task_t *task)
  Inserts thread task *into bubble* bubble.
- int **marcel_bubble_removeentity** (**marcel_bubble_t** *bubble, **marcel_entity_t** *entity)
  Removes entity entity *from bubble* bubble.
- int **marcel_bubble_removebubble** (**marcel_bubble_t** *bubble, **marcel_bubble_t** *little_bubble)
  Removes bubble little_bubble *from bubble* bubble.
- int **marcel_bubble_removethread** (**marcel_bubble_t** *bubble, marcel_task_t *task)
  Removes thread task *from bubble* bubble.
- int **marcel_entity_setschedlevel** (**marcel_entity_t** *entity, int level)
  Sets the "scheduling level" of entity entity *to* level. *This information is used by the Burst Scheduler.*
- int **marcel_bubble_setprio** (**marcel_bubble_t** *bubble, int prio)
  Sets the priority of bubble bubble *to* prio.

- int **marcel_bubble_getprio** (__const **marcel_bubble_t** *bubble, int *prio)
  Gets the priority of bubble bubble, *put into* *prio.

- int **marcel_bubble_setinitrq** (**marcel_bubble_t** *bubble, **ma_runqueue_t** *rq)
  Sets the initial runqueue of bubble bubble *to* rq. *When woken up (see* ***marcel_wake_up_bubble()*** (p. *113)),* bubble *will be put on runqueue* rq.

- int **marcel_bubble_setinitlevel** (**marcel_bubble_t** *bubble, **marcel_topo_level_t** *level)
  Sets the initial topology level of bubble bubble *to* level. *When woken up (see* ***marcel_wake_up_bubble()*** (p. *113)),* bubble *will be put on the runqueue of topology level* level.

- int **marcel_bubble_setinithere** (**marcel_bubble_t** *bubble)
  Sets the initial holder of bubble bubble *to the calling thread's current scheduling holder. When woken up (see* ***marcel_wake_up_bubble()*** (p. *113)),* bubble *will be put on the corresponding runqueue.*

- void **marcel_wake_up_bubble** (**marcel_bubble_t** *bubble)
  Wakes bubble bubble *up, i.e. puts it on its initial runqueue, and let the currently running bubble scheduler and the basic self scheduler schedule the threads it holds.*

- void **marcel_bubble_join** (**marcel_bubble_t** *bubble)
  Joins bubble bubble, *i.e. waits for all of its sub-bubbles and threads to terminate.*

- int **marcel_bubble_barrier** (**marcel_bubble_t** *bubble)
  Executes the barrier of bubble bubble, *which synchronizes all threads of the bubble.*

- **marcel_bubble_t** * **marcel_bubble_holding_task** (marcel_task_t *task)
  Returns the bubble that holds thread task.

- **marcel_bubble_t** * **marcel_bubble_holding_bubble** (**marcel_bubble_t** *bubble)
  Returns the bubble that holds bubble bubble.

- **marcel_bubble_t** * **marcel_bubble_holding_entity** (**marcel_entity_t** *entity)
  Returns the bubble that holds entity entity.

- **marcel_bubble_sched_t** * **marcel_bubble_change_sched** (**marcel_bubble_sched_t** *new_sched)
  Changes the current bubble scheduler, returns the old one.

Variables

- **marcel_bubble_sched_t** * **current_sched**
  The current bubble scheduler.

- **marcel_bubble_t marcel_root_bubble**
  The root bubble (default holding bubble).

A.2 Interface de programmation d'ordonnancement des bulles

Cette section regroupe les fonctions fournies pour implémenter des ordonnanceurs à bulles. La dernière version est disponible sur `http://pm2.gforge.inria.fr/marcel/html/group__marcel__bubble__sched.html`.

Data Structures

- struct **marcel_bubble**

Defines

- #define **ma_bubble_stats_get**(b, offset)
  Gets a statistic value of a bubble.

- #define **ma_bubble_hold_stats_get**(b, offset)
  Gets a synthesis of a statistic value of the content of a bubble, as collected during the last call to ***ma_bubble_synthesize_stats()*** *(p. 114).*

Functions

- void **ma_bubble_synthesize_stats** (**marcel_bubble_t** *bubble)
  Synthesizes statistics of the entities contained in bubble bubble*. The synthesis of statistics can be read thanks to* ***ma_bubble_hold_stats_get()*** *(p. 114).*

- int **ma_bubble_detach** (**marcel_bubble_t** *bubble)
  Detaches bubbles bubble *from its holding bubble, i.e. put* bubble *on the first runqueue encountering when following holding bubbles.*

- void **ma_bubble_gather** (**marcel_bubble_t** *b)
  Recursively gathers bubbles contained in b *back into it, i.e. get them from their holding runqueues and put them back into their holders.*

- void **ma_bubble_lock_all** (**marcel_bubble_t** *b, struct **marcel_topo_level** *level)
  Locks a whole bubble hierarchy. Also locks the whole level hierarchy.

- void **ma_bubble_unlock_all** (**marcel_bubble_t** *b, struct **marcel_topo_level** *level)
  Unlocks a whole bubble hierarchy. Also unlocks the whole level hierarchy.

- void **ma_bubble_lock** (**marcel_bubble_t** *b)
  Locks a bubble hierarchy. Stops at bubbles held on other runqueues.

- void **ma_bubble_unlock** (**marcel_bubble_t** *b)
  Unlocks a bubble hierarchy. Stops at bubbles held on other runqueues.

A.3 Interface d'accès aux statistiques

Cette section regroupe les fonctions fournies pour manipuler les statistiques attachées aux bulles et threads. La dernière version est disponible sur `http://pm2.gforge.inria.fr/marcel/html/group__marcel__stats.html`.

Defines

- #define **ma_stats_get**(object, offset)
  Gets the statistical value at the given offset in the stats *member of the given object.*

- #define **ma_stats_reset**(object)
  Resets all statistics for the given object.

- #define **ma_stats_synthesize**(dest, src)
  Synthesizes all statistics for the given objects.

- #define **marcel_stats_get**(t, kind)
  Application-level function for accessing statistics of a given thread (or the current one if t *is NULL).*

- #define **marcel_bubble_stats_get**(b, kind)
  Application-level function for accessing statistics of a given bubble.

Typedefs

- typedef char **ma_stats_t** [MARCEL_STATS_ROOM]
  Type of a statistics buffer

- typedef void **ma_stats_synthesis_t** (void *__restrict dest, const void *__restrict src)
  Type of a synthesizing function.

- typedef void **ma_stats_reset_t** (void *dest)
  Type of a reset function.

Functions

- **ma_stats_reset_t** * **ma_stats_reset_func** (unsigned long offset)
  Returns the reset function for the statistics at the given offset.

- **ma_stats_synthesis_t** * **ma_stats_synthesis_func** (unsigned long offset)
  Returns the synthesis function for the statistics at the given offset.

- size_t * **ma_stats_size** (unsigned long offset)
  Returns the size of the statistics at the given offset.

– unsigned long **ma_stats_alloc** (**ma_stats_reset_t** *reset_function, **ma_stats_synthesis_t** *synthesis_function, size_t size)
Declares a new statistics.

Variables

– unsigned long **marcel_stats_load_offset**
Offset of the "load" statistics (arbitrary user-provided).

– unsigned long **ma_stats_nbthreads_offset**
Offset of the "number of thread" statistics.

– unsigned long **ma_stats_nbghostthreads_offset**
Offset of the "number of ghost thread" statistics.

– unsigned long **ma_stats_nbrunning_offset**
Offset of the "number of running thread" statistics.

– unsigned long **ma_stats_last_ran_offset**
Offset of the "last ran time" statistics.

– unsigned long **ma_stats_memory_offset**
Offset of the "memory usage" statistics.

– **ma_stats_synthesis_t ma_stats_long_sum_synthesis**
"Sum" synthesis function for longs

– **ma_stats_reset_t ma_stats_long_sum_reset**
"Sum" reset function for longs (set to 0)

– **ma_stats_synthesis_t ma_stats_long_max_synthesis**
"Max" synthesis function for longs

– **ma_stats_reset_t ma_stats_long_max_reset**
"Max" reset function for longs (set to 0)

A.4 Conteneurs et entités

Cette section regroupe les outils pour travailler sur les relations entre conteneurs et entités La dernière version est disponible sur `http://pm2.gforge.inria.fr/marcel/html/group__marcel__holders.html`.

Enumerations

- enum **marcel_holder** { **MA_RUNQUEUE_HOLDER**, **MA_BUBBLE_HOLDER** }
  Holder types: runqueue or bubble.

- enum **marcel_entity** { **MA_BUBBLE_ENTITY**, **MA_THREAD_ENTITY**, **MA_THREAD_SEED** }
  Entity types: bubble or thread.

Functions

- enum **marcel_holder ma_holder_type** (**ma_holder_t** *h)
  Returns type of holder h.

- **marcel_bubble_t** * **ma_bubble_holder** (**ma_holder_t** *h)
  *Converts ma_holder_t *h into marcel_bubble_t * (assumes that* h *is a bubble).*

- **ma_runqueue_t** * **ma_rq_holder** (**ma_holder_t** *h)
  *Converts ma_holder_t *h into ma_runqueue_t * (assumes that* h *is a runqueue).*

- **ma_holder_t** * **ma_holder_bubble** (**marcel_bubble_t** *b)
  *Converts marcel_bubble_t *b into ma_holder_t *.*

- **ma_holder_t** * **ma_holder_rq** (**ma_runqueue_t** *rq)
  *Converts ma_runqueue_t *b into ma_holder_t *.*

- enum **marcel_entity ma_entity_type** (**marcel_entity_t** *e)
  Returns type of entity e.

- marcel_task_t * **ma_task_entity** (**marcel_entity_t** *e)
  *Converts ma_entity_t * e into marcel_task_t * (assumes that* e *is a thread).*

- **marcel_bubble_t** * **ma_bubble_entity** (**marcel_entity_t** *e)
  *Converts ma_entity_t * e into marcel_bubble_t * (assumes that* e *is a bubble).*

- **marcel_entity_t** * **ma_entity_bubble** (**marcel_bubble_t** *b)
  *Converts marcel_bubble_t *b into marcel_entity_t *.*

- **marcel_entity_t** * **ma_entity_task** (marcel_task_t *t)
  *Converts marcel_task_t *b into marcel_entity_t *.*

- void **ma_holder_lock_softirq** (**ma_holder_t** *h)
  Locks holder h.

- void **ma_holder_unlock_softirq** (**ma_holder_t** *h)
  Unlocks holder h.

- **ma_holder_t** * **ma_entity_holder_lock_softirq** (**marcel_entity_t** *e)
  Locks the holder of entity e *and return it.*

- **ma_holder_t** * **ma_task_holder_lock_softirq** (marcel_task_t *e)
  Locks the holder of task t *and return it.*

- void **ma_entity_holder_unlock_softirq** (**ma_holder_t** *h)
  Unlocks holder h.

- void **ma_task_holder_unlock_softirq** (**ma_holder_t** *h)
  Unlocks holder h.

- int **ma_get_entity** (**marcel_entity_t** *e)
  Gets entity e *out from its holder (which must be already locked), returns its state. If entity is a bubble, its hierarchy is supposed to be already locked.*

- void **ma_put_entity** (**marcel_entity_t** *e, **ma_holder_t** *h, int state)
  Puts entity e *into holder* h *(which must be already locked) in state* state*. If entity is a bubble, its hierarchy is supposed to be already locked.*

- void **ma_move_entity** (**marcel_entity_t** *e, **ma_holder_t** *h)
  Moves entity e *out from its holder (which must be already locked) into holder* h *(which must be also already locked). If entity is a bubble, its hierarchy is supposed to be already locked.*

Bibliographie

[AAD+02] Appavoo (J.), Auslander (M.), DaSilva (D.), Edelsohn (D.), Krieger (O.), Ostrowski (M.), Rosenburg (B.), Wisniewski (R. W.) et Xenidis (J.), « Scheduling in K42 ». Rapport technique, IBM Reseach, 2002.

[ABB02] Acar (U. A.), Blelloch (G. E.) et Blumofe (R. D.), « The Data Locality of Work Stealing », *Theory of Computing Systems*, vol. 35, n° 3, 05 2002, p. 321–347.

[ACD+07] Ayguade (E.), Copty (N.), Duranl (A.), Hoeflinger (J.), Lin (Y.), Massaioli (F.), Su (E.), Unnikrishnan (P.) et Zhang (G.), « A proposal for task parallelism in OpenMP », dans *Third International Workshop on OpenMP (IWOMP 2007)*, Beijing, China, 2007.

[AFK+06] Anselmi (G.), Filhol (B.), Kim (S.), Linzmeier (G.) et Plachy (O.), « IBM System p5 Quad Core Module Based On POWER5+ Technology Technical Overview and Introduction ». Rapport technique n° redp4150, IBM, février 2006.

[AGMJ04] Ayguade (E.), Gonzalez (M.), Martorell (X.) et Jost (G.), « Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications », dans *18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[ANP03] Antonopoulos (C.), Nikolopoulos (D.) et Papatheodorou (T.), « Scheduling algorithms with bus bandwidth considerations for SMPs », dans *2003 International Conference on Parallel Processing*, p. 547–554. IEEE, octobre 2003.

[BCZ+03] von Behren (R.), Condit (J.), Zhou (F.), Necula (G. C.) et Brewer (E.), « Capriccio: Scalable Threads for Internet Services », *Symposium on Operating Systems Principles*, octobre 2003, p. 19–22.

[BFS89] Bolosky (W. J.), Fitzgerald (R. P.) et Scott (M. L.), « Simple but effective techniques for numa memory management », dans *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, vol. 23, p. 19–31, 1989.

[BM02] Barreto (L. P.) et Muller (G.), « Bossa: une approche langage à la conception d'ordonnanceurs de processus », dans *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 14)*, Hammamet, Tunisie, avril 2002.

[BP04] Bulpin (J. R.) et Pratt (I. A.), « Multiprogramming Performance of the Pentium 4 with Hyper-Threading », dans *Third Annual Workshop on Duplicating, Deconstructing and Debunking (WDDD2004) (at ISCA'04)*, p. 53–62, juin 2004.

[Bro07] Broquedis (F.), « De l'exécution structurée de programmes OpenMP sur architectures hiérarchiques ». Mémoire de Master Recherche, juin 2007.

[BRS04] Boubekeur (T.), Reuter (P.) et Schlick (C.), « Reconstruction locale et visualisation de nuages de points par surfaces de subdivision », dans *AFIG '04*, 2004.

[BS05] Blikberg (R.) et Sørevik (T.), « Load balancing and OpenMP implementation of nested parallelism », *Parallel Computing*, vol. 31, n° 10-12, octobre 2005, p. 984–998.

[Bul] Bull, *Bull NovaScale servers*. `http://www.bull.com/novascale/`.

[Bul04] Bulpin (J. R.), *Operating system support for simultaneous multithreaded processors*. PhD thesis, University of Cambridge, Computer Laboratory, septembre 2004.

[CDC+99] Carlson (W.), Draper (J.), Culler (D.), Yelick (K.), Brooks (E.) et Warren (K.), « Introduction to UPC and Language Specification ». Rapport technique n° CCS-TR-99-157, George Mason University, mai 1999.

[CK01] Chang (M.-S.) et Kim (H.-J.), « Performance Analysis of a CC-NUMA Operating System », dans *15th International Parallel and Distributed Processing Symposium*. IEEE, 2001.

[Com] Compaq, *NUMA Scheduling Groups (NSG)*. `http://modman.unixdev.net/?sektion=4&page=numa_scheduling_groups&manpath=OSF1-V5.1-alpha`.

[Dan98] 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.

[Dan04] Danjean (V.), *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. PhD thesis, École Normale Supérieure de Lyon, décembre 2004.

[DC97] 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.

[DGC05] Duran (A.), Gonzàles (M.) et Corbalán (J.), « Automatic Thread Distribution for Nested Parallelism in OpenMP », dans *19th ACM International Conference on Supercomputing*, p. 121–130, Cambridge, MA, USA, juin 2005.

[DGL99] Demmel (J. W.), Gilbert (J. R.) et Li (X. S.), « An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination », *SIAM J. Matrix Analysis and Applications*, vol. 20, n° 4, 1999, p. 915–952.

[Dia07] Diakhaté (F.), « Reconstruction parallèle de surfaces par partition de l'unité hiérarchique ». Mémoire de Master Recherche, juin 2007.

[DMKJ96] Durand (D.), Montaut (T.), Kervella (L.) et Jalby (W.), « Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors », dans *Int. Conf. on Parallel and Distributed Systems*, vol. 7. IEEE, novembre 1996.

[DNR00a] 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.

[Dre03] Drepper (U.), « ELF Handling For Thread-Local Storage », février 2003. `http://people.redhat.com/drepper/tls.pdf`.

[DSCL04] Duran (A.), Silvera (R.), Corbalán (J.) et Labarta (J.), « Runtime adjustment of parallel nested loops », dans *International Workshop on OpenMP Applications and Tools (WOMPAT '04)*, Houston, TX, USA, mai 2004.

[DW05] Danjean (V.) et Wacrenier (P.-A.), « Mécanismes de traces efficaces pour programmes multithreadés », *Technique et science informatiques*, 2005.

[EAL93] Engler (D. R.), Andrews (G. R.) et Lowenthal (D. K.), « Filaments: Efficient Support for Fine-Grain Parallelism ». Rapport technique n° 93-13, University of Arizona, 1993.

[Edl95] Edler (J.), *Practical Structures for Parallel Operating Systems*. PhD thesis, New York University, mai 1995.

[EKO95] Engler (D. R.), Kaashoek (M. F.) et O'Toole (J.), « Exokernel: An Operating System Architecture for Application-Level Resource Management », *OPERATING SYSTEMS REVIEW*, vol. 29, n° 5, 1995, p. 251–266.

[EMGAD06] El-Moursy (A.), Garg (R.), Albonesi (D. H.) et Dwarkadas (S.), « Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors », dans *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[Fed06] Fedorova (A.), *Operating System Scheduling for Chip Multithreaded Processors*. PhD thesis, Harvard University, Cambridge, Massachusetts, 9 2006.

[FLR98] Frigo (M.), Leiserson (C. E.) et Randall (K. H.), « The Implementation of the Cilk-5 Multithreaded Language », dans *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, juin 1998.

[Fos95] Foster (I.), *Designing and building parallel programs : concepts and tools for parallel software engineering*. Addison-Wesley Publishing Compagny, Reading, MA, 1995.

[FR96] 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.

[GOM+00] Gonzalez (M.), Oliver (J.), Martorell (X.), Ayguade (E.), Labarta (J.) et Navarro (N.), « OpenMP Extensions for Thread Groups and Their Run-Time Support », dans *Languages and Compilers for Parallel Computing*. Springer Verlag, 2000.

[GSS+06] Gao (G. R.), Sterling (T.), Stevens (R.), Hereld (M.) et Zhu (W.), « Hierarchical multithreading: programming model and system software », dans *20th International Parallel and Distributed Processing Symposium (IPDPS)*, avril 2006.

[GSW+06] Gerndt (A.), Sarholz (S.), Wolter (M.), an Mey (D.), Bischof (C.) et Kuhlen (T.), « Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets », dans *Super Computing*, novembre 2006.

[HD07] Hadjidoukas (P. E.) et Dimakopoulos (V. V.), « Nested Parallelism in the OMPi OpenMP/C compiler », dans *EuroPar*, Rennes,France, juillet 2007. ACM.

[HIM+05] Hsu (L.), Iyer (R.), Makineni (S.), Reinhardt (S.) et Newell (D.), « Exploring the cache design space for large scale CMPs », *SIGARCH Comput. Archit. News*, vol. 33, n° 4, 2005, p. 24–33.

[HK99] Hagersten (E.) et Koster (M.), « WildFire: A Scalable Path for SMPs », dans *The Fifth International Symposium on High Performance Computer Architecture*, janvier 1999. Sun Microsystems, Inc.

[HP03] Hennessy (J. L.) et Patterson (D. A.), *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3e édition, 2003.

[HRR00] Hénon (P.), Ramet (P.) et Roman (J.), « PaStiX: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions », dans *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, p. 519–527. Springer-Verlag, janvier 2000.

[Jeu06] Jeuland (S.), « Documentation : Compatibilité posix de marcel ». Mémoire de Stage, septembre 2006.

[Jeu07] Jeuland (S.), « Ordonnancement de threads dirigé par la mémoire sur architecture numa ». Mémoire de Master Recherche, juin 2007.

[JHA02] Jain (R.), Hughes (C. J.) et Adve (S. V.), « Soft Real-Time Scheduling on Simultaneous Multithreaded Processors », 2002.

[KLS+94] Koelbel (C.), Loveman (D.), Schreiber (R.), Steele (G.) et Zosel (M.), *The High Performance Fortran Handbook*, 1994.

[LC96a] Lai (G.-J.) et Chen (C.), « A New Scheduling Strategy for NUMA Multiprocessor Systems », juin 1996, p. 222–229.

[LCS96] Lovett (T. D.), Clapp (R. M.) et Safranek (R. J.), « NUMA-Q: An SCI-Based Enterprise Server ». Rapport technique, Sequent Computer Systems Inc., 1996.

[LH05] Löf (H.) et Holmgren (S.), « affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system », dans *19th ACM International Conference on Supercomputing*, p. 387–392, Cambridge, MA, USA, juin 2005.

[Lin] Linux, *Linux Scalability Effort*. `http://lse.sourceforge.net/`.

[LL97] Laudon (J.) et Lenoski (D.), « The SGI Origin: A ccNUMA Highly Scalable Server », dans *24th International Symposium on Computer Architecture*, p. 241–251, juin 1997. Silicon Graphics, Inc.

[LTSS93] Li (H.), Tandri (S.), Stumm (M.) et Sevcik (K. C.), « Locality and Loop Scheduling on NUMA Multiprocessors », dans *International Conference on Parallel Processing*, vol. II, p. 140–127, août 1993.

[MAN+99] Martorell (X.), Ayguadé (E.), Navarro (N.), Corbalán (J.), González (M.) et Labarta (J.), « Thread Fork/Join Techniques for Multi-Level Parallelism Exploitation in NUMA Multiprocessors », dans *International Conference on SuperComputing*, p. 294–301. ACM, 1999.

[MAN05] McGregor (R. L.), Antonopoulos (C. D.) et Nikolopoulos (D. S.), « Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors », dans *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, p. 28.1, Washington, DC, USA, 2005. IEEE Computer Society.

[MBH+02] Marr (D. T.), Binns (F.), Hill (D. L.), Hinton (G.), Koufaty (D. A.), Miller (J. A.) et Upton (M.), « Hyper-Threading Technology Architecture and Microarchitecture », *Intel Technology*, vol. 6 issue 01, 2002.

[ME99] Mattson (T.) et EigenMann (R.), « OpenMP: An API for Writing Portable SMP Application Software », dans *SuperComputing 99 Conference*, novembre 1999.

[Men07] Menage (P. B.), « Adding Generic Process Containers to the Linux Kernel », dans *Linux Symposium*, Ottawa, Ontario, Canada, juin 2007.

[min] « Ming ». `http://ming.sourceforge.net/`.

[ML94] Markatos (E.) et Leblanc (T.), « Using processor affinity in loop scheduling on shared-memory multiprocessors », *Parallel and Distributed Systems*, vol. 5, n° 4, avril 1994, p. 379–400.

[MLV+03] Morin (C.), Lottiaux (R.), Vallée (G.), Gallard (P.), Utard (G.), Badrinath (R.) et Rilling (L.), « Kerrighed: A Single System Image Cluster Operating System for High Performance Computing », dans *EuroPar*, Klagenfurt, Austria, août 2003.

[MM06] Marathe (J.) et Mueller (F.), « Hardware Profile-guided Automatic Page Placement for ccNUMA Systems », dans *Sixth Symposium on Principles and Practice of Parallel Programming*, mars 2006.

[MS98] McKenney (P. E.) et Slingwine (J. D.), « Read-copy update: Using execution history to solve concurrency problems », dans *Parallel and Distributed Computing and Systems*, p. 509–518, Las Vegas, NV, October 1998.

[MST07] an Mey (D.), Sarholz (S.) et Terboven (C.), « Nested Parallelization with OpenMP », *Parallel Computing*, vol. 35, n° 5, octobre 2007, p. 459–476.

[Nam97] Namyst (R.), *PM2: un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Univ. de Lille 1, janvier 1997.

[Nar02] Narlikar (G. J.), « Scheduling Threads for Low Space Requirement and Good Locality », *Theory of Computing Systems*, vol. 35, n° 2, 1 2002, p. 151–187.

[Nit06] Nitsure (A.), « Implementation and Optimization of a Cache Oblivious Lattice Boltzmann Algorithm ». Master's thesis, FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG, 2006.

[NPP98] Nikolopoulos (D. S.), Polychronopoulos (E. D.) et Papatheodorou (T. S.), « Efficient Runtime Thread Management for the Nano-Threads Programming Model », dans *Second IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming*, vol. 1388, p. 183–194, Orlando, FL, USA, avril 1998.

[NPP+00] Nikolopoulos (D. S.), Papatheodorou (T. S.), Polychronopoulos (C. D.), Labarta (J.) et Ayguadé (E.), « User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors », dans *International Conference on Parallel Processing*, p. 95–103. IEEE, septembre 2000.

[NPP+02] Nikolopoulos (D. S.), Polychronopoulos (C. D.), Papatheodorou (T. S.), Labarta (J.) et Ayguadé (E.), « Scheduler-Activated Dynamic Page Migration for Multiprogrammed DSM Multiprocessors », *Parallel and Distributed Computing*, vol. 62, décembre 2002, p. 1069–1103.

[ON01] Oguma (H.) et Nakayama (Y.), « A Scheduling Mechanism for Lock-free Operation of a Lightweight Process Library for SMP Computers », dans *Conference on Parallel and Distributed Systems*, p. 235–242. IEEE, juillet 2001.

[ope] « OpenMP ». `http://www.openmp.org/`.

[Ous82] Ousterhout (J. K.), « Scheduling techniques for concurrent systems », dans *Third International Conference on Distributed Computing Systems*, p. 22–30, octobre 1982.

[Pér05] Pérache (M.), « Nouveaux mécanismes au sein des ordonnanceurs de threads pour une implantation efficace des opérations collectives sur machines multiprocesseurs », dans *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 16)*, Le Croisic, France, mars 2005.

[Pér06] Pérache (M.), *Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances*. PhD thesis, Univ. de Bordeaux, octobre 2006.

[Rat] Rattner (J.), « Platform 2015 ». `http://www.intel.com/technology/magazine/computing/platform-2015-0305.htm`.

[RF97] Reed (D.) et Fairbairns (R.), « The nemesis kerneloverview », 1997.

[RO05] Roper (M. D.) et Olsson (R. A.), « Developing Embedded Multi-threaded Applications with CATAPULTS, a Domain-specic Language for Generating Thread Schedulers », dans *CASES'05*, p. 24–27, septembre 2005.

[Rob03] Roberson (J.), « ULE: A Modern Scheduler For FreeBSD ». Rapport technique, The FreeBSD Project, `jeff@FreeBSD.org`, 2003.

[Rus97] Russinovich (M.), « Inside the Windows NT Scheduler, Part 2 », *Windows IT Pro*, vol. 303, juillet 1997. `http://www.windowsitpro.com/Article/ArticleID/303/303.html`.

[SB95] Steckermeier (M.) et Bellosa (F.), « Using Locality Information in Userlevel Scheduling ». Rapport technique n° TR-95-14, University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstraße 1, 91058 Erlangen, Germany, décembre 1995. `http://www4.informatik.uni-erlangen.de/Projects/FORTWIHR/ELiTE/`.

[Sch96] Schreiber (R.), « An Introduction to HPF », dans *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, p. 27–44. Springer-Verlag, 1996.

[SGDA05] Shen (X.), Gao (Y.), Ding (C.) et Archambault (R.), « Lightweight Reference Affinity Analysis », dans *19th ACM International Conference on Supercomputing*, p. 131–140, Cambridge, MA, USA, juin 2005.

[sgi] sgi, *Process Aggregates*. `http://oss.sgi.com/projects/pagg/`.

[Sol96] Solutions (D. I.), *The Dolphin SCI Interconnect*, février 1996.

[SPE] « SPEC OMP ». `http://www.spec.org/omp/`.

[Sun] Sun microsystems, *Solaris Memory Placement Optimization (MPO)*. `http://partneradvantage.sun.com/protected/solaris10/adoptionkit/tech/mpo/mpo_man.html`.

[tbb] « Thread Building Blocks ». `http://www.intel.com/software/products/tbb/`.

[TGBS05] Tian (X.), Girkar (M.), Bik (A.) et Saito (H.), « Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs », *Comput. J.*, vol. 48, n° 5, 2005, p. 588–601.

[TGS+03] Tian (X.), Girkar (M.), Shah (S.), Armstrong (D.), Su (E.) et Petersen (P.), « Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures », dans *Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, p. 47–55, avril 2003.

[THH+05] Tian (X.), Hoeflinger (J. P.), Haab (G.), Chen (Y.-K.), Girkar (M.) et Shah (S.), « A compiler for exploiting nested parallelism in OpenMP programs », *Parallel Comput.*, vol. 31, n° 10-12, 2005, p. 960–983.

[top] « TOP500 Supercomputing Sites ». `http://www.top500.org/`.

[TTSY00] Tanaka (Y.), Taura (K.), Sato (M.) et Yonezawa (A.), « Performance evaluation of openmp applications with nested parallelism », dans *Languages, Compilers, and Run-Time Systems for Scalable Computers*, p. 100–112, 2000.

[TY86] Tang (P.) et Yew (P.-C.), « Processor Self-Scheduling for Multiple Nested Parallel Loops », dans *Proceedings 1986 International Conference on Parallel Processing*, p. 528–535, août 1986.

[WA01] 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.

[WJ03] der Wijngaart (R. F. V.) et Jin (H.), « NAS Parallel Benchmarks, Multi-Zone Versions ». Rapport technique n° NAS-03-010, NASA Advanced Supercomputing (NAS) Division, 2003.

[WMB+97] Whitney (S.), McCalpin (J.), Bitar (N.), Richardson (J. L.) et Stevens (L.), « The SGI Origin Software Environment and Application Performance », dans *COMPCON 97*, p. 165–170, San Jose, California, 1997. IEEE.

[WRRF03] Woodacre (M.), Robb (D.), Roe (D.) et Feind (K.), « The SGI Altix 3000 Global Shared-Memory Architecture ». Rapport technique n° 3474 J14186, Silicon Graphics, Inc., octobre 2003.

[WWC97] 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.

[WWC00] 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.

[YAM+03] Y. (O.), A. (B.), M. (A.), G. (T.) et H-P. (S.), « Multi-level Partition of Unity Implicits », dans *SIGGRAPH 2003*, 2003.

[ZOSD04] Zhong (Y.), Orlovich (M.), Shen (X.) et Ding (C.), « Array regrouping and structure splitting using whole-program reference affinity », dans *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, p. 255–266, New York, NY, USA, 2004. ACM Press.

Liste des publications

Conférences internationales avec publication des actes et comité de lecture

[1] THIBAULT (S.), NAMYST (R.) et WACRENIER (P.-A.), « Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework », dans *EuroPar*, Rennes,France, juillet 2007. ACM.

Colloques internationaux avec publication des actes et comité de lecture

[2] THIBAULT (S.), « A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines », dans *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge / USA, juin 2005. ICS / ACM / IRISA.

[3] THIBAULT (S.), BROQUEDIS (F.), GOGLIN (B.) *et al.*, « An Efficient OpenMP Runtime System for Hierarchical Architectures », dans *International Workshop on OpenMP (IWOMP)*, p. 148–159, Beijing,China, juin 2007.

Revues nationales avec comité de lecture

[4] THIBAULT (S.), NAMYST (R.) et WACRENIER (P.-A.), « Bubblesched: une plate-forme pour la conception d'ordonnanceurs de threads portables sur machines multiprocesseurs hiérarchiques », *Technique et Science Informatiques*, 2007. En cours de publication.

Conférences nationales avec publication des actes et comité de lecture

[5] THIBAULT (S.), « Un ordonnanceur flexible pour machines multiprocesseurs hiérarchiques », dans *16ème Rencontres Francophones du Parallélisme*, Le Croisic / France, avril 2005. ACM/ASF - École des Mines de Nantes.

[6] THIBAULT (S.), « Bubblesched : construire son propre ordonnanceur de threads pour machines multiprocesseurs hirarchiques », dans *17ème Rencontres Francophones du Parallélisme*, Canet en Roussillon / France, octobre 2006. ACM/ASF - Université de Perpignan.

Rapports de recherche

[7] THIBAULT (S.), « Un ordonnanceur flexible pour machines multiprocesseurs hiérarchisées ». Rapport de stage DEA, juillet 2004. http://enslyon.free.fr/rapports/info/Samuel_Thibault_3.ps.gz.

Manuels

[8] THIBAULT (S.). « BubbleSched ». http://runtime.futurs.inria.fr/marcel/bubblesched.php.

[9] THIBAULT (S.). « BubbleSched API ». http://runtime.futurs.inria.fr/marcel/doc/.

-=-=-=-