HAL
open science

De l'exécution structurée d'applications scientifiques OpenMP sur les architectures hiérarchiques.

François Broquedis

To cite this version:

François Broquedis. De l'exécution structurée d'applications scientifiques OpenMP sur les architectures hiérarchiques.. Calcul parallèle, distribué et partagé [cs.DC]. Université Sciences et Technologies - Bordeaux I, 2010. Français. NNT : . tel-00793472

HAL Id: tel-00793472

https://theses.hal.science/tel-00793472v1

Submitted on 22 Feb 2013

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 : 4190

THÈSE

présentée à

L'Université Bordeaux 1

École Doctorale de Mathématiques et Informatique

par Monsieur François BROQUEDIS

pour obtenir le grade de

Docteur

Spécialité : Informatique

De l'exécution structurée d'applications scientifiques OpenMP sur architectures hiérarchiques

Soutenue le : 9 Décembre 2010

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.	Denis	BARTHOU	Professeur des Universités	Président
M.	Jean-François	MÉHAUT	Professeur des Universités	Rapporteur
M.	Raymond	NAMYST	Professeur des Universités	
M.	Thierry	PRIOL	Directeur de Recherche INRIA	Rapporteur
Mme.	Frédérique	SILBER-CHAUSSUMIER	Maître de Conférences	Examinatrice
M.	Pierre-André	WACRENIER	Maître de Conférences	

Remerciements

Mes premiers remerciements se destinent naturellement à mes directeurs de thèse, Raymond Namyst et Pierre-André Wacrenier, qui ont constitué à eux deux une source intarissable d'inspiration et de motivation au cours de ces trois années de thèse, qui ont suivi un stage de DEA et quelques années d'études à Bordeaux au cours desquelles ils ont su me transmettre leur passion pour la recherche. J'aimerais les remercier particulièrement pour leur encadrement sans faille et leur complémentarité. Je souhaite à tout futur doctorant d'être encadré par un duo de cette qualité.

J'adresse aussi mes remerciements aux membres de mon jury, Denis Barthou, Frédérique Silber-Chaussumier et tout particulièrement Thierry Priol et Jean-François Méhaut pour le temps qu'ils ont consacré à la relecture de mon manuscrit et la pertinence des corrections qu'ils m'ont proposées. Un très grand merci à Jean-François (un deuxième!) pour son excellent accueil à Grenoble et pour m'avoir dégagé suffisamment de temps pour me permettre d'achever la rédaction de ce manuscrit.

Je souhaite ensuite remercier toute l'équipe Runtime pour m'avoir accompagné et soutenu au cours de cette thèse. Ils ont contribué à entretenir une excellente ambiance créant un climat idéal pour la croissance du petit Forest. Travailler avec les gens qu'on apprécie aide qui plus est énormément à se lever du bon pied le matin! Je remercie particulièrement Samuel qui m'a passé le flambeau et pris sous sa coupe en DEA, Olivier pour sa disponibilité et le temps passé à débugger avec moi, Brice pour les innombrables coups de main (et quelques coups de pieds parfois!) notamment sur les aspects mémoire, Nathalie ma MaMI préférée, Cédric pour les Nutty Bars et les *a capella* de Guy Marchand, mais surtout pour son pep's capable de redynamiser n'importe quel doctorant travaillant sur ForestGOMP, Alexandre pour PukABI que je n'aurais pas aimé écrire, Guillaume pour MegaShark Versus Crocosaurus que j'aurais aimé réaliser, Marie-Christine pour sa bienveillance et ses conseils avisés sur l'enseignement, Stéphanie pour son gateau à la carotte, Emmanuel pour la chouette balade lisboète, Louis-Claude pour son scepticisme attachant, les ingénieux Ludovic et Yannick, Sylvie pour m'avoir sauvé la vie plus d'une fois en mission, et Sylvain parce qu'il rédigera sa thèse après moi et je compte bien figurer dans ses remerciements. Je n'oublie pas non plus les doctorants ayant soutenu avant moi ou les étudiants de DEA qui sont partis faire leur thèse hors de Bordeaux avec qui j'ai passé de bons moments, à savoir Elisabeth, Sylvain, Mathieu et François.

J'adresse des remerciements particuliers à Jérôme, qui non content d'avoir été un excellent camarade de fac pendant de nombreuses années, est depuis devenu un véritable ami sur lequel je peux compter, ayant même survécu à trois années de cohabitation avec moi dans les locaux inflammables de l'INRIA Bordeaux. Je lui dois beaucoup pour ma réussite estudiantine et pour ce qu'il m'apporte en dehors.

Je souhaite aussi remercier particulièrement Ludovic, d'un part pour le travail incommensurable qu'il a effectué sur ForestGOMP au cours de son passage dans l'équipe Runtime, mais aussi et surtout pour ce qu'il est, à savoir une personne aussi sympathique que compétente (j'espère qu'il ne le prendra pas mal!). C'est un plaisir de travailler avec lui, mais aussi de s'attabler autour d'une bière. J'ai vraiment vécu son arrivée dans l'équipe comme une grosse bouffée d'oxygène, le petit Forest ayant du mal à grandir en cette 2e année de thèse, et je tenais à le remercier aussi pour ça.

Je remercie aussi tous les membres de ma famille, qui me soutiennent quoique j'entreprenne, et qui ont répondu présent le jour de ma soutenance, avec entre autres l'organisation d'un pot de thèse qui en a séduit plus d'un. Je souhaite remercier tout particulièrement mon père, jusqu'alors seul universitaire de la famille, pour me servir d'exemple tant au niveau professionnel, en m'ayant fait découvrir son métier, que personnel en m'aidant à grandir et à lui ressembler un peu plus. Je remercie aussi ma mère, à qui je dois certainement mes facilités de prise de parole, d'être à jamais la présidente de mon fan-club malgré mes travers de "petit dernier". Je remercie ma grande petite sœur qui n'est jamais très loin quand il s'agit de me prouver qu'elle tient à moi. Je remercie enfin Odette, Henri, Mirentxu, Alain, Cécile et Guillaume de s'être déplacés et de m'avoir soutenu en ce jour si particulier.

Je ne peux finir ces remerciements sans remercier mes amis, Stéphane, Leslie, Jérôme (encore!), Mathilde, Damien, Laurent, Cédric, Stéphane (encore, mais pas le même), pour supporter mes âneries et mes vannes pas drôles depuis de nombreuses années déjà. Un remerciement particulier à Stéphane, qui comme Jérôme a grandi avec moi sur les bancs de la fac. Je lui réserve une place toute particulière dans mon (multi)cœur.

Enfin, cerise sur le gateau basque, je remercie Cathy qui partage ma vie depuis maintenant cinq années, au cours desquelles elle m'a toujours soutenu, parfois supporté, pure source de bonheur, d'amour et de stabilité. Elle n'est certainement pas étrangère à l'aboutissement de cette thèse.

Résumé : Le domaine applicatif de la simulation numérique requiert toujours plus de puissance de calcul. La technologie multicœur aide à satisfaire ces besoins mais impose toutefois de nouvelles contraintes aux programmeurs d'applications scientifiques qu'ils devront respecter s'ils souhaitent en tirer la quintessence. En particulier, il devient plus que jamais nécessaire de structurer le parallélisme des applications pour s'adapter au relief imposé par la hiérarchie mémoire des architectures multicœurs. Les approches existantes pour les programmer ne tiennent pas compte de cette caractéristique, et le respect de la structure du parallélisme reste à la charge du programmeur. Il reste de ce fait très difficile de développer une application qui soit à la fois performante et portable.
La contribution de cette thèse s'articule en trois axes. Il s'agit dans un premier temps de s'appuyer sur le langage OpenMP pour générer du parallélisme structuré, et de permettre au programmeur de transmettre cette structure au support exécutif ForestGOMP. L'exécution structurée de ces flots de calcul est ensuite laissée aux ordonnanceurs *Cache* et *Memory* développés au cours de cette thèse, permettant respectivement de maximiser la réutilisation des caches partagés et de maximiser la bande passante mémoire accessible par les programmes OpenMP. Enfin, nous avons étudié la composition de ces ordonnanceurs, et plus généralement de bibliothèques parallèles, en considérant cette voie comme une piste sérieuse pour exploiter efficacement les multiples unités de calcul des architectures multicœurs.
Les gains obtenus sur des applications scientifiques montrent l'intérêt d'une communication forte entre l'application et le support exécutif, permettant l'ordonnancement dynamique et portable de parallélisme structuré sur les architectures hiérarchiques.

Mots-clés : Calcul hautes performances, support d'exécution, OpenMP, multicœur, NUMA

Table des matières

Introduction **1**

1 Contexte: L'ère pré-multicœur **5**
1.1 Évolutions récentes des architectures parallèles à mémoire commune du 20e siècle . . . 6
1.1.1 Du parallélisme depuis la conception interne des processeurs... . . 6
1.1.1.1 Le parallélisme d'instruction . . . 6
1.1.1.2 Le parallélisme de données . . . 7
1.1.2 ... jusqu'au niveau des cartes mères . . . 8
1.1.2.1 Architectures multiprocesseurs symétriques (SMP) . . . 8
1.1.2.2 Accès mémoire non-uniformes (NUMA) . . . 9
1.2 Programmation concurrente des architectures à mémoire commune des années 2000 . . . 10
1.2.1 Programmation par mémoire partagée . . . 10
1.2.1.1 Bibliothèques de processus légers . . . 10
1.2.1.2 Langages de programmation parallèle en mémoire partagée . . . 12
1.2.1.3 Techniques algorithmiques pour une gestion efficace du cache . . . 13
1.2.2 Programmation par passage de messages . . . 14
1.2.3 Programmation à parallélisme de données . . . 15
1.2.4 Programmation fonctionnelle . . . 15
1.2.5 Programmation à base de machines virtuellement partagées . . . 16
1.3 Discussion . . . 17

2 Motivation et état de l'art : la révolution du multicœur **19**
2.1 L'avènement du multicœur . . . 19
2.1.1 Des processeurs multithreads aux architectures multicœurs . . . 20
2.1.1.1 Masquer la latence d'accès à la mémoire . . . 20
2.1.1.2 Optimiser l'utilisation du pipeline d'instructions . . . 21
2.1.2 Plusieurs cœurs sur une même puce . . . 21
2.1.3 Des machines multicœurs aux accès mémoire non-uniformes . . . 22
2.2 Programmation des architectures multicœurs . . . 24
2.2.1 Outils et supports exécutifs spécifiques . . . 24
2.2.1.1 Support logiciel pour la programmation des multicœurs . 24

2.2.1.2 Supports exécutifs conçus pour les architectures hiérarchiques . . . 25
2.2.2 Support des compilateurs . . . 26
2.2.3 Langages de programmation . . . 27
2.3 Discussion . . . 29

3 Contribution: Structurer pour régner! 31
3.1 Capturer l'information et la pérenniser . . . 32
3.1.1 Quel parallélisme pour les architectures multicœurs? . . . 32
3.1.2 Structurer le parallélisme de façon pérenne . . . 33
3.1.3 OpenMP, un langage de programmation parallèle structurant . . . 34
3.1.4 Des indications pérennes pour une exécution structurée des programmes OpenMP . . . 35
3.2 Projeter dynamiquement le parallélisme sur la topologie . . . 35
3.2.1 Expression générique des architectures hiérarchiques . . . 36
3.2.2 Ordonnancement dynamique de parallélisme structuré . . . 36
3.2.2.1 Différents besoins, différents ordonnanceurs . . . 37
3.2.3 Cache: un ordonnanceur qui respecte les affinités entre threads . . 38
3.2.3.1 Répartir les groupes de threads de façon compacte . . . 38
3.2.3.2 Equilibrer la charge quand une unité de calcul devient inactive . . . 40
3.2.4 Memory: un ordonnanceur qui tient compte des affinités mémoire 44
3.2.4.1 Débit mémoire et localité . . . 44
3.2.4.2 Support logiciel pour la gestion des données sur les architectures NUMA . . . 46
3.2.4.3 Transmission des affinités mémoire au support exécutif . 46
3.2.4.4 Ordonnancement conjoint de threads et de données . . . 49
3.3 Composer, confiner, dimensionner . . . 51
3.3.1 Composition de parallélisme . . . 53
3.3.2 Confinement de parallélisme . . . 54
3.3.2.1 Partitionnement de l'ensemble des ressources de calcul . 55
3.3.2.2 Gestion de la surcharge en environnement confiné . . . 56
3.3.3 Dimensionnement de régions parallèles OpenMP . . . 59
3.4 Synthèse . . . 59

4 Éléments d'implémentation 61
4.1 Au niveau de ForestGOMP . . . 61
4.1.1 Une implémentation de barrière pour les architectures hiérarchiques 61
4.1.2 Un effort de compatibilité . . . 65
4.2 Au niveau de BubbleSched . . . 66
4.2.1 Des ordonnanceurs instanciables et composables . . . 66
4.2.2 Un module d'aide au développement d'algorithmes gloutons . . . 68
4.3 Au niveau de GotoBLAS . . . 70
4.4 Synthèse et discussion . . . 72

5 Evaluation 73
5.1 Plateforme expérimentale . . . 74

5.1.1 Hagrid 74
5.1.2 Kwak 74
5.2 Performances brutes de ForestGOMP 76
5.2.1 EPCC 76
5.2.2 Nested EPCC 77
5.2.3 La suite de benchmarks de la NASA 77
5.3 Ordonnancement d'applications au parallélisme irrégulier 79
5.3.1 BT-MZ, une application à deux niveaux de parallélisme aux comportements différents 79
5.3.2 MPU, une application au parallélisme imbriqué irrégulier dans le temps 80
5.4 Ordonnancement d'applications aux accès mémoire intensifs 83
5.4.1 STREAM 83
5.4.2 Nested-STREAM 84
5.4.3 Twisted-STREAM 85
5.4.3.1 100% de données distantes 85
5.4.3.2 66% de données distantes 86
5.4.4 Imbalanced-STREAM 87
5.4.5 LU 89
5.5 Composabilité, dimensionnement 91

6 Conclusion et perspectives 93

A Interface de programmation de ForestGOMP 97
A.1 Variables globales et structures de données 97
A.2 Primitives de gestion de données et d'affinités mémoire 98
A.3 Primitives pour définir des charges de travail 98
A.4 Primitives de débogage 99

B Instrumentation du benchmark Nested-STREAM 101
B.1 Allocation et libération de zones mémoire 101
B.2 Attachement des zones mémoire aux équipes OpenMP 102

Bibliographie 105

Liste des publications 109

Introduction

La simulation numérique joue aujourd'hui un rôle prépondérant dans la démarche scientifique et nécessite toujours plus de puissance de calcul pour traiter des problèmes plus gros, augmenter la précision des simulations ou encore en réduire les temps de traitement. Par le passé, l'augmentation de la puissance de calcul d'une machine se traduisait à la fois par l'augmentation de la fréquence de ses processeurs mais aussi par la complexification de leurs composants. Aujourd'hui, la fréquence n'évolue plus en raison de problèmes de dissipation de chaleur et les architectes ont épuisé les solutions matérielles permettant de faire évoluer leurs processeurs. Au lieu de cela, ils s'appuient désormais sur la miniaturisation galopante des unités de calcul et proposent d'en graver plusieurs sur une même puce, donnant ainsi naissance aux processeurs multicœurs.

Ces architectures se démarquent des machines de calcul à processeurs symétriques, dans lequelles un ensemble de processeurs identiques accèdent de façon uniforme à une mémoire commune, par l'organisation hiérarchique des différents niveaux de mémoire qui les composent. Certaines unités de calcul partagent ainsi plusieurs niveaux de cache mémoire qu'elles pourront exploiter pour communiquer efficacement. La mémoire vive est elle morcelée et répartie en bancs sur l'ensemble de l'architecture, cassant ainsi le modèle des architectures à accès mémoires uniformes présentes dans les calculateurs à processeurs symétriques. Ces spécifités architecturales confrontent le programmeur d'applications parallèles à de nouvelles contraintes, comme le respect des affinités entre les traitements et les données auxquelles ils accèdent.

Les architectures multicœurs sont aujourd'hui partout: on les retrouve depuis les nœuds qui composent les grappes de calcul jusque dans les ordinateurs et terminaux mobiles du grand public. Les modèles de programmation de type SPMD sont ainsi eux aussi confrontés à cette révolution, puisque les nœuds qui composent les grappes de calcul actuelles sont multicœurs. Les approches de type langage de programmation parallèle reviennent naturellement à la mode, puisqu'il permet au plus grand nombre, du grand public à l'expert en calcul haute performance, d'exploiter les multiples unités de calcul des machines multicœurs. Cependant, la plupart de ces approches ont été pensées pour tirer parti des architectures à processeurs symétriques. Quoi de plus naturel alors que d'exposer un parallélisme à un seul niveau pour occuper chacun des processeurs de ces architectures *planes*. Un tel parallélisme peine en revanche à s'adapter de façon harmonieuse au relief des architectures multicœurs.

Pour continuer à être performants, les langages de programmation parallèle doivent entamer une mutation qui leur permettra de s'adapter à de telles architectures. Cette évolution passe par la facilité d'exprimer toujours plus de parallélisme, de paralléliser

à grain plus fin. Le langage OpenMP, qui bénéficie d'un regain d'intérêt avec l'arrivée des architectures multicœurs, a su évoluer dans ce sens, en proposant le mécanisme de tâche qui, s'ajoutant à la possibilité d'imbriquer des régions parallèles, offre d'autant plus de moyens au programmeur pour occuper tous les cœurs de sa machine. Sa simplicité d'utilisation lui permet d'être aujourd'hui encore très utilisé par la communauté du calcul hautes performances. Il porte de plus la majorité des recherches en programmation hybride des grappes de calcul. En revanche, le parallélisme qu'il expose, bien que massif, peine à s'adapter aux hiérarchies mémoire des architectures multicœurs. Il semble en effet naturel de confier l'exécution de traitements connexes à des unités de calcul partageant des ressources comme des niveaux de cache ou des bancs mémoires.

Objectifs de contributions de la thèse

De ce constat émerge la nécessité, pour le programmeur, d'exprimer les relations d'affinité entre les traitements parallèles de son application. Par sa nature structurante, notamment grâce à la possibilité d'imbriquer des régions parallèles, OpenMP se trouve être un bon candidat pour tirer parti des architectures hiérarchiques en exprimant de façon simple ces relations d'affinité. En particulier, en invoquant une région parallèle, le programmeur OpenMP exprime de façon indirecte que les traitements parallèles qui lui sont associés sont en étroite relation. Cette information reste difficilement exploitable de façon portable au niveau applicatif, d'autant plus qu'elle doit être associée à une connaissance précise de l'architecture ciblée. C'est pourquoi, pour atteindre la portabilité des performances, il devient nécessaire de confier l'orchestration de l'exécution des applications parallèles au support exécutif. Cette approche repense le rôle du programmeur. Il n'a en effet plus à penser son application pour un type précis d'architecture, ni adapter explicitement son programme aux spécificités matérielles des machines multicœurs. En revanche, le support exécutif attend de lui qu'il transmette des informations sur l'application exécutée, comme les relations d'affinités entre traitements parallèles ou les motifs d'accès aux données, qui lui serviront à parfaire son ordonnancement mais aussi à l'adapter aux besoins de l'application.

Les développements de cette thèse s'inscrivent dans ce cadre. Il s'agit d'abord de proposer au programmeur des moyens d'exprimer ces informations, en s'appuyant sur la structuration naturelle du langage OpenMP pour, dans un premier temps, transmettre les informations d'affinité entre traitements parallèles au support exécutif. Ces informations guident l'ordonnancement et doivent donc pouvoir être consultables à tout moment de l'exécution du programme. Nous assurons cette pérennité grâce aux *bulles* qui permettent au support exécutif de définir et d'ordonnancer des groupes de traitements parallèles en relation. Une fois la structure des applications OpenMP capturée dans des bulles, le support exécutif projette ce parallélisme sur l'architecture cible, et s'appuie pour cela sur une expression générique de la topologie lui permettant de définir des politiques d'ordonnancement pouvant s'appliquer à n'importe quelle architecture. En particulier, deux politiques d'ordonnancement ont été développées dans le but d'améliorer les performances des applications OpenMP sur les architectures hiérarchiques: l'une favorise l'utilisation des différents niveaux de mémoire cache d'un processeur multicœur, l'autre s'attache à maximiser la bande passante mémoire accessible en limitant

les effets de la contention sur les architectures à accès mémoire non-uniformes. Nous étudions dans un deuxième temps la possibilité de composer OpenMP avec d'autres bibliothèques parallèles, en particulier la bibliothèque d'algèbre linéaire GotoBLAS, et mettons en évidence les caractéristiques nécessaires pour obtenir un environnement parallèle composable.

Organisation du document

Le chapitre 1 présente un historique des évolutions architecturales de l'ère pré-multicœur et des paradigmes de programmation employés pour l'exploitation des machines à mémoire commune. Le chapitre 2 introduit les architectures multicœur et présente les façons de les programmer, certaines issues d'évolutions d'approches existantes, d'autres ayant été spécifiquement développées pour ces architectures. Le chapitre 3 développe la contribution de cette thèse en exposant de manière théorique les solutions apportées pour capturer la structure du parallélisme OpenMP, orchestrer son ordonnancement de manière dynamique et portable et enfin garantir sa composabilité avec d'autres sources de parallélisme. Le chapitre 4 fournit des détails sur l'implémentation de notre support exécutif, des ordonnanceurs développés et de l'adaptation de la bibliothèque GotoBLAS pour permettre son utilisation en contexte multiprogrammé. Le chapitre 5 permet de valider les développements effectués au cours de cette thèse sur des benchmarks et applications de la vie réelle. Nous tirons enfin les conclusions de ces travaux et abordons les perspectives de recherche qu'ils ouvrent.

Chapitre 1

Contexte: L'ère pré-multicœur

Sommaire

1.1 Évolutions récentes des architectures parallèles à mémoire commune du 20e siècle . . . 6

1.1.1 Du parallélisme depuis la conception interne des processeurs... 6

1.1.1.1 Le parallélisme d'instruction . . . 6

1.1.1.2 Le parallélisme de données . . . 7

1.1.2 ... jusqu'au niveau des cartes mères . . . 8

1.1.2.1 Architectures multiprocesseurs symétriques (SMP) . . . 8

1.1.2.2 Accès mémoire non-uniformes (NUMA) . . . 9

1.2 Programmation concurrente des architectures à mémoire commune des années 2000 . . . 10

1.2.1 Programmation par mémoire partagée . . . 10

1.2.1.1 Bibliothèques de processus légers . . . 10

1.2.1.2 Langages de programmation parallèle en mémoire partagée . . . 12

1.2.1.3 Techniques algorithmiques pour une gestion efficace du cache . . . 13

1.2.2 Programmation par passage de messages . . . 14

1.2.3 Programmation à parallélisme de données . . . 15

1.2.4 Programmation fonctionnelle . . . 15

1.2.5 Programmation à base de machines virtuellement partagées . . . 16

1.3 Discussion . . . 17

Le dernier quart du siècle dernier aura vu l'apparition progressive du parallélisme au sein des architectures des supercalculateurs à mémoire commune: au niveau des unités fonctionnelles qui composent les processeurs tout d'abord, jusqu'au niveau architectural avec l'apparition de machines multiprocesseurs. Ce chapitre présente la conception et la programmation de ces architectures parallèles.

1.1 Évolutions récentes des architectures parallèles à mémoire commune du 20e siècle

Hennessy et Patterson [HP03] rangent en deux groupes les efforts des architectes pour augmenter la puissance des ordinateurs contemporains. Ils distinguent d'une part les optimisations pour faire en sorte qu'une suite d'instructions s'exécute le plus rapidement possible (optimisation de la vitesse de calcul) et d'autre part celles qui visent à favoriser l'exécution de multiples flux de calcul (optimisation du débit). Nous rappelons ici les principales techniques d'optimisation introduites par les architectes au cours des années 1980-2000.

1.1.1 Du parallélisme depuis la conception interne des processeurs...

Les architectes ont d'abord eu pour objectif de traduire la *loi de Moore*, qui prévoit la multiplication par deux du nombre de circuits tous les 18 mois, en une augmentation comparable de la fréquence des processeurs. Pour atteindre cet objectif, ils ont progressivement intégré du parallélisme au sein même des processeurs. Cette section rappelle comment l'industrie du microprocesseur a réussi à paralléliser l'exécution de programmes séquentiels.

1.1.1.1 Le parallélisme d'instruction

Depuis le milieu des années 80, toutes les architectures de processeurs s'appuient sur le principe de *pipeline* qui consiste à diviser l'exécution d'une instruction en plusieurs étapes successives. Chacune de ces étapes fait intervenir une unité fonctionnelle dédiée et matériellement indépendante des autres unités. À l'image du travail à la chaîne, il est donc possible de traiter en parallèle plusieurs instructions issues d'un même flot d'exécution, en faisant travailler de façon concurrente différentes unités fonctionnelles sur différentes instructions. La figure 1.1 illustre ce fonctionnement. Dans cette situation, jusqu'à trois instructions peuvent être en cours de traitement dans le pipeline, qualifié ainsi de pipeline à trois étages. Notons que la fréquence de l'horloge est limitée par la latence de l'étage le plus lent. Aussi certaines étapes étant plus complexe que d'autres, ou certaines instructions réclamant des traitements spécifiques comme des opérations flottantes par exemple, on a multiplié certaines unités fonctionnelles pour résorber des goulets d'étranglement en permettant l'exécution simultanées de plusieurs instructions. On parle alors de processeurs *superscalaires*.

Il arrive souvent que les unités fonctionnelles d'un processeur soient forcées de retarder l'exécution du programme. C'est le cas par exemple lorsque des instructions qui se suivent dépendent les unes des autres, ou encore lorsque le programme exécuté contient des branchements conditionnels pour lesquels il peut s'avérer difficile de décider assez tôt quelle sera la prochaine instruction à exécuter. L'introduction de techniques telles que le réordonnancement dynamique d'instructions, le renommage de registres ou la prédiction de branchements améliorent l'efficacité des pipelines. Elles ne résolvent cependant en rien les problèmes liés aux dépendances entre instructions, ou encore ceux provoqués par l'apparition de défauts de cache.

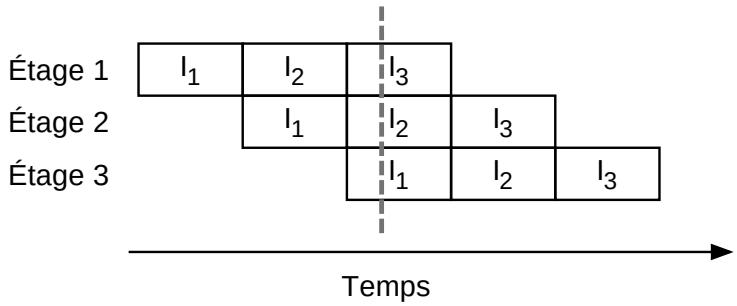


FIG. 1.1 – Progression de trois instructions I_1, I_2, I_3, dans un pipeline à trois étages

1.1.1.2 Le parallélisme de données

Une architecture à parallélisme de données comporte des unités de calcul qui exécutent le même code sur des données différentes (SIMD). Le parallélisme est ici induit par la distribution des données sur les différents processeurs.

Parmi les architectures à parallélisme de données, on retrouve les architectures systoliques composées de processeurs spécialisés tous identiques, connectés par un réseau d'interconnexion organisé le plus souvent en anneau ou en grille, auquel on adjoint un processeur généraliste. Chaque processeur élémentaire, appelé cellule, communique avec ses voisins directs. En pratique, il reçoit des données de ses voisins, effectue un calcul et leur transmet le résultat à chaque cycle de la machine systolique. Les cellules calculent en parallèle, de façon synchrone, au rythme d'une horloge globale. Ces architectures sont encore exploitées dans le domaine de la génomique et du traitement du signal via la conception de circuits FPGA ou ASIC adaptés.

Les ordinateurs massivement parallèles forment une deuxième famille d'architecture SIMD. On retrouve parmi eux la Goodyear Massively Parallel Processor (1980), les Connection Machine CM 1 (1983), CM 2 (1987) et CM 5 (1991) ou encore les MasPar MP1 et MP2. Fondée en 1987, la société *MasPar* produit jusqu'en 1996 des machines massivement parallèles de plusieurs centaines, voire plusieurs milliers de processeurs élémentaires. Chacun de ces processeurs comporte 16ko de mémoire vive et un jeu de 16 registres 32 bits. Ils sont organisés en *clusters*, ou *grappes* en français, de 4x4 processeurs élémentaires, qui exécutent tous le même code. Chaque processeur est directement connecté à ses huit voisins par un bus. Un routeur autorise les processeurs élémentaires à établir des communications globales de type *broadcast* ou *multicast* qui peuvent être invoquées par le programmeur à l'aide d'un langage spécifique baptisé *MPL* [Ver]. Cette extension du langage C introduit de plus la notion de variable parallèle qui facilite la distribution des données sur les différents processeurs élémentaires. Il est intéressant de remarquer que les architectures vectorielles comme celle des machines *MasPar* sont aujourd'hui remises au goût du jour par les constructeurs de cartes graphiques. En particulier, la programmation d'un *GPU* fait appel à des techniques similaires à celles employées pour programmer une machine à parallélisme de données des années 90.

Un troisième type d'architecture SIMD est celle des processeurs vectoriels conçue pour optimiser l'exécution d'une même opération sur des vecteurs de nombres. Il s'agit d'optimiser le traitement de vecteurs de bout en bout : de l'accès aux opérandes jusqu'au rangement du résultat et ce en évitant toute latence mémoire ou dépendance de données. Ainsi, les constructeurs *Cray* puis *Nec* ont contribué au développement des processeurs vectoriels parmi les supercalculateurs par exemple l'*Earth Simulator*, basé sur le processeur vectoriel NEC SX-6, a dominé le TOP 500 de 2002 à 2004.

On trouve aussi des instructions dites vectorielles dans le jeu d'instruction Intel à base de technologie *MMX* [PW96](ou plus tard *SSE*) et de la technologie *AltiVec* chez Motorola et IBM. Il s'agit ici d'appliquer une opération à des grands registres, on parle de technologie SWAR (SIMD Within A Register). Les processeurs SSE offrent par exemple des registres de 128 bits, qui peuvent donc accueillir jusqu'à 4 entiers 32 bits. Le parallélisme de données induit par de telles technologies reste donc modeste comparé à celui exposé par les machines vectorielles présentées précédemment, et l'on peut assimiler la technologie SSE à une extension vectorielle pour processeur scalaire. Les instructions SSE sont souvent introduites pour accélérer des traitements particuliers comme la compression et la décompression vidéo par exemple.

1.1.2 ... jusqu'au niveau des cartes mères

En marge des évolutions internes aux processeurs, le parallélisme s'est peu à peu imposé au niveau architectural grâce à la possibilité de connecter plusieurs processeurs entre eux, donnant ainsi naissance à des architectures multiprocesseurs que l'on peut classer en deux grandes familles: les architectures multiprocesseurs symétriques et les architectures multiprocesseurs à accès mémoire non-uniformes.

1.1.2.1 Architectures multiprocesseurs symétriques (SMP)

Les architectures multiprocesseurs symétriques se composent de plusieurs processeurs identiques reliés entre eux par un bus ou un commutateur parfait. Chaque processeur dispose de sa propre mémoire cache et accèdent de façon symétrique à une mémoire commune. La figure 1.2 représente une architecture SMP à quatre processeurs.

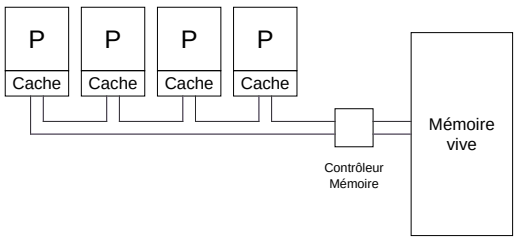


FIG. 1.2 – Architecture SMP à quatre processeurs.

La cohérence de cache est assurée par la technologie de *bus snooping* : chaque contrôleur de cache *surveille* sur le bus le trafic lié aux adresses qui concernent des données stockées par le cache dont il a la charge. Il est ainsi capable de reconnaître une requête qui concerne une ligne de son cache et déclencher des actions d'écriture sur le bus ou d'invalidation.

Bien que relativement simples à concevoir, ces architectures peinent cependant à passer à l'échelle. La bande passante du bus mémoire, en partie utilisée par le mécanisme de *bus snooping*, ou la complexité du contrôleur sont autant de facteurs limitants qui empêchent les constructeurs d'implanter des architectures SMP à plusieurs dizaines de processeurs. Les architectes se sont alors tournés vers les machines à accès mémoire non-uniformes pour lever cette limitation.

1.1.2.2 Accès mémoire non-uniformes (NUMA)

Les architectures à accès mémoire non-uniformes se démarquent des architectures SMP par leur organisation mémoire. Ici, au lieu d'accéder à une mémoire commune, les processeurs accèdent à une mémoire répartie en différents bancs distribués en plusieurs endroits de l'architecture, comme représenté sur la figure 1.3. Les processeurs regroupés autour d'un banc mémoire forment un nœud NUMA qui ressemble en tout point à une architecture SMP classique. Les processeurs d'un même nœud accèdent ainsi de façon symétrique au banc mémoire qui lui est attaché. En revanche, accéder au banc mémoire d'un nœud NUMA distant prendra plus de temps qu'accéder au banc mémoire local. On définit le *facteur NUMA* d'une telle architecture comme le rapport entre la latence d'accès à une donnée distante et celle nécessaire pour accéder à une donnée locale. Plus ce rapport est proche de 1, plus l'architecture se comportera comme une machine SMP. L'interconnexion de plusieurs nœuds permet donc la construction d'architectures parallèles à plus grande échelle, mais la multiplication du nombre de nœuds au sein d'une même machine peut provoquer une augmentation significative du facteur NUMA. De plus, bien que cette forme de conception lève les limites des architectures SMP liées à la bande passante mémoire, une mauvaise répartition des données peut faire réapparaitre des problèmes de contention.

La cohérence de cache s'appuie sur plusieurs *répertoires*, composants matériels attachés à chacun des nœuds NUMA de la machine, qui arbitrent l'accès aux données partagées entre plusieurs processeurs. Un processeur qui souhaite charger une donnée depuis la mémoire de son nœud NUMA vers son cache doit demander l'autorisation au répertoire afin que ce dernier puisse maintenir la liste des processeurs accédant aux différentes données partagées. Quand une entrée du répertoire est modifiée, il invalide les copies correspondantes dans les caches des processeurs concernés. Ce mécanisme met donc en œuvre des communications point-à-point, à la différence du *cache snooping* qui s'appuie sur les communications globales, et passe donc mieux à l'échelle.

Depuis leur apparition dans le TOP 500 vers 1995 (IBM SP, SGI Origin), les architectures NUMA ont peu à peu perdu du terrain sur le marché des supercalculateurs au profit des grappes de PC, beaucoup moins chères à construire. Cependant, nous verrons dans la suite que l'introduction par les constructeurs de processeurs multicœurs de technologies d'interconnexion replace ces architectures au devant de la scène du calcul hautes performances.

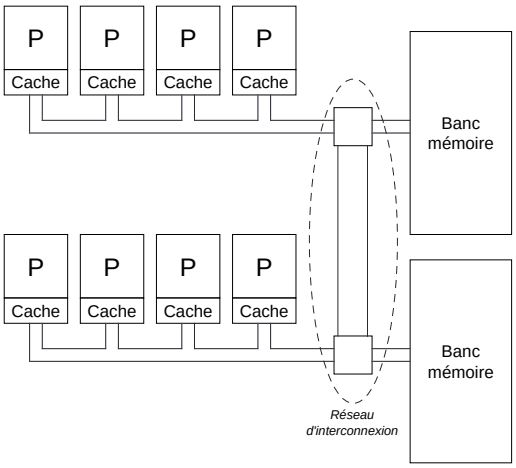


FIG. 1.3 – Architecture NUMA à deux nœuds de quatre processeurs.

1.2 Programmation concurrente des architectures à mémoire commune des années 2000

Les modèles de programmation parallèle sont aujourd'hui relativement nombreux. Certains d'entre eux ont été pensés avant l'apparition du parallélisme, comme les paradigmes de programmation impérative et fonctionnelle, et ont dû évoluer pour s'adapter aux architectures parallèles. Leurs évolutions se posent alors comme autant de façons de programmer une machine parallèle, comme le présente cette section.

1.2.1 Programmation par mémoire partagée

La programmation parallèle par mémoire partagée consiste à créer des traitements, exécutés de manière concurrente, évoluant dans le même espace d'adressage. Ils sont de ce fait capables de communiquer entre eux par l'intermédiaire de la mémoire de la machine. La création de ces traitements peut se faire en s'appuyant sur une bibliothèque de processus légers ou par le biais d'un langage de programmation parallèle.

1.2.1.1 Bibliothèques de processus légers

Les bibliothèques de processus légers fournissent les primitives de création, gestion et destruction des threads. Elles permettent la gestion explicite du parallélisme depuis l'application. Cette approche de bas niveau reste très technique mais offre à l'expert le plus grand panel d'expressivité et d'optimisations possibles quant au parallélisme de son application. Cependant, ces optimisations nécessitent, pour être portables, un réel investissement en temps de développement. De plus, ces bibliothèques se basent sur différents modes d'ordonnancement pour orchestrer l'exécution du parallélisme sur

l'architecture. On distingue trois grands types de bibliothèques de processus légers, selon si ces derniers sont entièrement gérés en espace utilisateur, au sein du noyau ou de façon mixte.

Bibliothèques de niveau utilisateur Le parallélisme issu de ces bibliothèques est intégralement géré en espace utilisateur. Les primitives de gestion de threads sont donc très efficaces puisqu'elles ne nécessitent aucun appel système. Cette approche permet d'adapter le parallélisme et l'ordonnancement aux besoins de l'application. En effet, il est ici possible d'implémenter des fonctionnalités supplémentaires, comme la migration de threads utile pour équilibrer la charge. Les coûts de création et de gestion des threads étant dans ces conditions minimes, la production d'un grand nombre de processus légers n'engendre pas l'écroulement de la machine. En revanche, les threads générés par ces bibliothèques ne sont pas visibles par l'ordonnanceur du système d'exploitation. De ce fait, ils ne pourront pas être exécutés par plusieurs processeurs de la machine. De plus, l'exécution d'une primitive potentiellement bloquante comme une écriture sur disque par exemple peut provoquer le blocage du processus entier qui embarque l'ordonnanceur et tous les threads de la bibliothèque. On retrouve parmi les bibliothèques de niveau utilisateur les *Green Threads* [Ber96] de Sun, permettant d'obtenir le même ordonnancement quel que soit le système d'exploitation sur lequel tourne la machine virtuelle Java, par exemple.

Bibliothèques de niveau noyau Les threads issus d'une bibliothèque de niveau noyau sont intégralement gérés par l'ordonnanceur du système. Cette approche permet donc une exécution concurrente des traitements parallèles générés, au prix cependant de primitives de gestion bien plus coûteuses que l'approche utilisateur. Les threads d'une telle bibliothèque peuvent en effet s'apparenter à des processus à mémoire partagée. On les appelle parfois en français *threads lourds*, en amusante opposition à la terminologie anglo-saxonne *light-weight processes*. Cette approche autorise par essence l'exécution concurrente des threads créés, puisqu'ils sont ordonnancés par le système d'exploitation. Elle ignore aussi les problèmes liés aux entrées/sorties, puisque l'exécution d'un appel bloquant aura pour effet de seulement bloquer le thread concerné. Le processeur rendu inactif par cette attente pourra exécuter un autre thread prêt. La gestion d'un tel parallélisme est en revanche bien plus coûteuse, et ces threads peuvent s'apparenter à de véritables processus. Parmi ces bibliothèques, on retrouve par exemple les *Linux Threads* de Xavier Leroy, qui furent plus tard remplacés par les threads de la *Native POSIX Threads Library* [DM05], encore utilisée par Linux.

Bibliothèques hybrides Les bibliothèques hybrides créent un certain nombre de threads noyau capables d'exécuter des threads de niveau utilisateur. Les threads noyau, visibles par l'ordonnanceur du système d'exploitation, peuvent être exécutés de manière concurrente sur différents processeurs. On parle d'ordonnanceur N:M lorsque N threads utilisateurs sont exécutés par l'intermédiaire de M threads noyau, que l'on peut considérer comme autant de rampes de lancement pour le parallélisme exprimé au niveau utilisateur. Cette approche offre la flexibilité des bibliothèques en espace utilisateur tout en étant capable d'exploiter plusieurs processeurs. Elle affiche aussi une

meilleure réactivité vis-à-vis des entrées/sorties. Les bibliothèques hybrides sont cependant complexes à développer et à maintenir. La bibliothèque hybride *Next Generation POSIX Threading* [IBM] a été proposée comme alternative aux POSIX threads de Linux, avant de céder sa place à la *NPTL*. La bibliothèque de threads *Marcel* de la suite logicielle *PM2* [Nam97] fonctionne aussi en ordonnanceur N:M.

1.2.1.2 Langages de programmation parallèle en mémoire partagée

Si elle permet d'obtenir les meilleures performances, la programmation à parallélisme explicite peut s'avérer technique, complexe voire fastidieuse. Avec les architectures parallèles sont apparus des langages qui visent à masquer certains détails techniques liés à la parallélisation d'un programme à l'utilisateur, comme la création des threads et leur terminaison par exemple. Cette section présente quelques unes de ces approches.

OpenMP Le standard OpenMP [Ope], défini par un consortium d'industriels et d'universitaires, voit le jour en octobre 1997. OpenMP définit un ensemble d'annotations pour la parallélisation de programmes séquentiels écrits en C, C++ ou Fortran. Ces annotations permettent la gestion automatique du parallélisme, que ce soit la création d'un nombre concerté de threads en relation avec le nombre de processeurs présents sur la machine, leur synchronisation ou encore leur terminaison. OpenMP séduit surtout parce qu'il permet de facilement paralléliser des boucles et de décider de la répartition des indices à traiter par chaque thread. L'utilisateur dispose en effet d'un certain nombre de mots-clés qu'il peut passer en paramètre des directives OpenMP. Ces mots-clés définissent par exemple la visibilité (privée ou partagée) de variables, la façon de répartir les indices d'une boucle entre les threads, ou encore le nombre de threads associés à une région parallèle. Le standard définit aussi des primitives pour la synchronisation et l'exclusion mutuelle des threads OpenMP. La figure 1.4 présente le code d'une parallélisation de boucle C à l'aide d'OpenMP. La directive `parallel for` indique que la boucle qui suit sera exécutée en parallèle. Le mot-clé `schedule` définit la politique de partage de travail entre les threads: on choisit ici la politique `static` selon laquelle chaque thread traite un bloc contigu d'itérations de taille `N` divisée par le nombre de threads participants.

Bien que performant sur les architectures SMP, OpenMP peine à s'imposer sur les architectures NUMA. En effet, rien n'existe dans le standard pour exprimer le fait qu'une région parallèle accède à une certaine zone mémoire, et la gestion des données était totalement laissée à la charge du programmeur, voire du système d'exploitation.

Cilk Cilk [FLR98] est une extension du langage C dont le but est de permettre l'écriture de programmes parallèles performants quel que soit le nombre de processeurs d'une architecture SMP. Il s'appuie pour cela sur une gestion de tâches parallèles extrêmement efficace. Le coût de lancement d'une tâche ne représente que 2 à 6 fois le coût d'appel d'une fonction C classique. Le modèle d'exécution d'un programme Cilk est toujours le même: quelques processus légers, créés au démarrage du programme, ont la charge d'exécuter des tâches Cilk, que l'on peut grossièrement assimiler à une structure

```
#define N 1000
static int a[N], b[N], c[N];
void sum () {
  int i;
#pragma omp parallel for schedule (static)
  for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
}
```

FIG. 1.4 – Parallélisation d'une boucle à l'aide d'OpenMP.

de données contenant le pointeur de la fonction à exécuter et ses arguments. Chaque thread propulseur possède sa propre file de travaux pouvant accueillir des tâches Cilk. Quand un propulseur n'a plus de travail à effectuer, il peut voler une tâche depuis la queue de la file de travaux d'un autre propulseur. Les tâches Cilk étant ajoutées en tête de liste, l'exécution du programme se déroule sans aucune synchronisation entre les threads propulseurs. Le langage Cilk permet donc une parallélisation très efficace et pratiquement sans surcoût sur une architecture monoprocesseur, mais ne s'applique qu'à des algorithmes récursifs de type *"Diviser pour régner"*. Les programmes Cilk peuvent aussi se comporter de façon décevantes sur les architectures NUMA, puisque le vol de travail s'effectue de façon aléatoire, sans tenir compte de la localité mémoire.

1.2.1.3 Techniques algorithmiques pour une gestion efficace du cache

L'exploitation efficace des caches des processeurs est obligatoire pour tirer parti des architectures parallèles. Lorsqu'il connaît les caractéristiques matérielles des caches des processeurs de la machine sur laquelle s'exécute son application, comme leur taille totale et la taille d'une ligne de cache, le programmeur est capable de :

1. dimensionner ses données afin qu'elles puissent être stockées en cache
2. aligner les données accédées fréquemment pour éviter les phénomènes de *faux partage*

Les algorithmes qu'il développe sont alors qualifiés de *cache conscious*, et peuvent allier performance et portabilité à condition de détecter les caractéristique des caches des processeurs de façon automatique.

D'autres techniques algorithmiques ont été pensées pour aider le programmeur à développer des codes portables qui tirent naturellement parti de la mémoire cache. Ces algorithmes, qualifiés de *cache oblivious* [FLPR99], ignorent tout des caractéristiques matérielles des caches des processeurs. Ils adaptent de manière récursive la granularité des données qu'ils manipulent, le problème à traiter étant subdivisé en plusieurs sous-problèmes plus petits qui finissent par tenir en cache. Ils s'attachent aussi à respecter la dimension temporelle d'un cache, de façon à limiter les évictions de lignes qui seront accédées dans le futur.

1.2.2 Programmation par passage de messages

La programmation par passage de messages implique la création de traitements exécutés en parallèle, qui échangent des informations au cours de l'exécution du programme en étant tour à tour émetteur puis récepteur. Ce modèle est avant tout utilisé pour la programmation des grappes de calcul, mais souvent aussi pour programmer une architecture à mémoire partagée, le problème du *faux partage* étant ici résolu par duplication des données.

PVM *Parallel Virtual Machine* [SGDM94] est une bibliothèque de communication développée en 1989 par Oak Ridge. Elle permet de bâtir une machine virtuelle unique à partir d'un réseau de machines potentiellement hétérogènes. Cette bibliothèque repose sur une architecture à deux niveaux: une couche de bas niveau comportant autant de composantes que d'environnements supportés (architectures de processeurs et systèmes d'exploitation) et une couche de plus haut niveau qui fournit l'interface de programmation générique de la machine PVM. Elle permet donc de faire communiquer ensemble des matériels supposés incompatibles. En pratique, chaque nœud qui compose la machine PVM lance un démon, chargé de l'authentification, de l'attribution des ressources, du routage des messages, et plus généralement des communications avec les autres démons PVM. L'interface de programmation permet la création de tâches PVM, leur synchronisation, mais aussi l'envoi et la réception de messages entre tâches. Apparu alors qu'il n'existait pas encore de standard pour la programmation parallèle, PVM s'est rapidement imposé auprès de la communauté scientifique. Le but de cette bibliothèque n'étant pas la performance mais la flexibilité et l'interopérabilité, l'implantation de fonctionnalités avancées comme le support de la dynamicité ou encore la gestion des appels non bloquants engendrent des surcoûts qui incitent les experts du calcul hautes performances à se tourner vers d'autres standards comme MPI.

MPI La norme MPI [SO98], pour *Message Passing Interface*, proposée en 1993 par un consortium regroupant des industriels et des équipes de recherche universitaires, définit une interface standard permettant de communiquer à travers n'importe quel type de réseau. Les applications MPI communiquent sans se soucier du matériel réseau sous-jacent, faisant de ce standard une approche portable pour la programmation des grilles de calcul. MPI s'appuie sur un modèle de programmation SPMD (*Single Program Multiple Data*: une application MPI créé un certain nombre de processus qui exécutent tous le même programme avec des paramètres d'entrée différents. Ce modèle de programmation s'adapte bien aux architectures logicielles de type *maitre-esclaves*, où un processus particulier distribue du travail aux autres processus. Les processus MPI s'échangent des messages au cours de l'exécution du programme. L'interface de programmation fournit les primitives nécessaires à différents modes de communication entre processus (synchrones, asynchrones, diffusion) ainsi qu'à leur synchronisation. Un sous ensemble de processus MPI peuvent être rattachés au même *communicateur*, une structure de données permettant de définir des sous-groupes de processus et ainsi de limiter la portée des messages de diffusion par exemple. Certaines implantations de ce standard obtiennent des performances proches des performances brutes du réseau sous-jacent. MPI

se distingue en cela de PVM. Il existe de plus des implémentations du standard spécialement optimisées pour communiquer en mémoire partagée. En revanche, il demeure impossible de faire communiquer deux implémentations de MPI différentes, contrairement à PVM qui prone l'interopérabilité.

1.2.3 Programmation à parallélisme de données

Contrairement aux approches multiprogrammées évoquées ci-dessus, les langages à parallélisme de données cherchent le plus possible à s'abstraire de l'architecture pour proposer des codes portables. Dans un langage à parallélisme de données, un flot d'exécution séquentiel peut contenir des expressions parallèles. L'approche proposée par ces langages est de dicter la génération des traitements parallèles en fonction de la distribution des données. Les données jouent donc un rôle prépondérant dans le parallélisme de ces langages. En particulier, la gestion des tableaux est mise en avant. Une addition de vecteurs s'écrit par exemple très simplement à l'aide d'un langage à parallélisme de données, puisqu'aucune boucle n'est nécessaire pour les parcourir. Parmi ces langages, *HPF* a fait partie des plus prometteurs.

HPF *High Performance Fortran* [Sch96] étend le langage Fortran en proposant un certain nombre de directives de compilation. Ces directives définissent la découpe et la distribution d'objets sur un ensemble de *processeurs abstraits*. La distribution des processeurs abstraits sur les processeurs physiques de la machine dépend de l'implémentation du support exécutif HPF. Les données doivent obligatoirement être organisées sous forme de tableaux pour être traitées par HPF. Une fois un tableau distribué par HPF, il est possible d'utiliser la directive *ALIGN* pour reproduire cette distribution sur des données différentes. La distribution des données se fait au moment de leur déclaration. Bien que prometteur, HPF a souffert de son ambition. Les fonctionnalités du langage ont été difficiles à intégrer aux compilateurs, contrariant fortement le démarrage de ce langage en 1993. La façon de programmer des langages à parallélisme de données diffère aussi grandement de celle poussée par les approches multiprogrammées ou par passage de messages qui commençaient déjà à se développer au moment de la sortie d'HPF 2.0 en 1997.

1.2.4 Programmation fonctionnelle

Toutes les approches citées jusqu'à présent s'appuient sur le paradigme de programmation impérative selon lequel un programme, par l'intermédiaire de variables dites *mutables*, modifie l'état global de la mémoire. Un code écrit en langage de programmation impérative peut être réduit à une suite d'instructions élémentaires exécutables par le processeur.

Apparu à la fin des années 50, le paradigme de programmation fonctionnelle s'appuie sur le lambda-calcul, système formel à qui l'on doit la définition et la caractérisation des fonctions récursives, pour se détacher de la programmation d'une machine à état global. Les seuls objets manipulés par les langages fonctionnels sont les fonctions et

les constantes. Ainsi, à chaque calcul correspond une ou plusieurs évaluations de fonctions mathématiques. Bien que ces langages soient pour la plupart interprétés, certains d'entre eux disposent de compilateurs.

Le paradigme de programmation fonctionnelle dispose d'un certain nombre d'atouts qui facilitent le développement d'applications parallèles. En particulier, les langages fonctionnels manipulent des fonctions *pures*, sans effets de bord, qui peuvent de ce fait être exécutées indépendamment des autres. On s'affranchit ainsi de l'utilisation de verrous, améliorant la concision et l'élégance du code produit tout en facilitant grandement son développement. En revanche, l'absence d'effets de bord implique la nécessité de recopier des données pour communiquer entre traitements parallèles, ce qui augmente considérablement le trafic sur le bus mémoire et dégrade les performances.

1.2.5 Programmation à base de machines virtuellement partagées

Le paradigme de programmation à base de machines virtuellement partagée met l'accent sur la facilité de développement en s'appuyant sur une abstraction, fournie par le système d'exploitation [MLV+04] ou une bibliothèque spécifique, permettant le partage de données entre nœuds de calcul ne possédant pas de mémoire commune. Il offre donc un moyen de programmer les grappes de machines à la façon d'une architecture à mémoire commune. C'est dans ce cadre que nous l'étudions.

Intel Cluster OpenMP *Cluster OpenMP* [Hoe06] est une machine à mémoire virtuellement partagée conçue par Intel pour l'exécution de programmes OpenMP sur des grappes de machines. Le langage OpenMP est étendu par l'ajout du mot-clé `sharable`, identifiant les variables référencées par plus d'un thread qui seront gérées par la machine virtuellement partagée. La cohérence mémoire entre les différents nœuds de calcul est assurée en protégeant les pages mémoire hébergeant les variables `sharable` à l'aide de l'appel système `mprotect` sur chaque nœud de calcul. A la modification d'une de ces pages, Cluster OpenMP interdit l'accès aux pages correspondantes sur les autres nœuds en lecture et écriture. Un nœud de calcul qui souhaite lire cette page recevra donc un signal `SIGSEGV` intercepté par Cluster OpenMP, qui réclamera alors la mise à jour de cette page au dernier nœud l'ayant modifié. Les constructions traditionnelles d'OpenMP, comme les barrières et les réductions, sont aussi repensées pour fonctionner efficacement en contexte réparti. En revanche, Cluster OpenMP ne s'appuie pas sur la mémoire virtuellement partagée pour attirer les traitements au plus près des données qu'ils accèdent, comme cela est possible avec le langage *UPC*.

UPC Le langage UPC [CDC+99], pour *Unified Parallel C*, est une extension du langage C pour la programmation des grappes de machines multiprocesseurs. L'originalité du langage UPC réside dans le fait qu'il s'appuie sur le modèle de mémoire virtuellement partagée pour distribuer dynamiquement les calculs en fonction de l'emplacement physique des données qu'ils utilisent. Le programmeur explicite pour cela les variables à partager. UPC propose aussi deux types de cohérence mémoire que le programmeur peut choisir d'appliquer aux variables partagées de son application: la cohérence sé-

quentielle, pour laquelle toute écriture ou lecture d'une donnée demande la synchronisation de la mémoire et la cohérence relachée, pour laquelle la mémoire n'est synchronisée qu'aux points de synchronisation du programme, comme les barrières par exemple. La répartition du calcul est faite par rapport aux données en s'appuyant sur des informations d'affinités exprimées par le programmeur au niveau des boucles à distribuer. Ainsi, dans l'exemple illustré par la figure 1.5, tout indice `k` de la boucle sera traité par le thread se trouvant à proximité de la variable partagée `a[k]`.

```
#define N 1000
relaxed shared int a[N], b[N], c[N];
void sum () {
  int i;
  upc_forall (i = 0; i < N; i++; &a[i])
    a[i] = b[i] + c[i];
}
```

FIG. 1.5 – Cas d'école de l'utilisation du langage UPC.

S'inspirant des techniques de programmation des machines à mémoire commune, les approches basées sur les machines à mémoire virtuellement partagée sont à la peine lorsqu'il s'agit d'exécuter des programmes où la localité des données par rapport aux traitements n'est pas assurée. En effet, les phénomènes comme le *faux partage* dégrade de façon amplifiée les performances: la mémoire virtuellement partagée est découpée à plus gros grain qu'un cache de processeur, ce qui augmente d'autant la probabilité de retrouver des données fréquemment accédées sur la même subdivision de mémoire virtuelle. D'une manière plus générale, les surcoûts engendrés par les mécanismes de cohérence mémoire sur architecture à mémoire commune, comme les défauts de cache et défauts de page, sont amplifiés sur une machine à mémoire virtuellement partagée, qui est donc plus difficile à programmer efficacement. Optimiser un programme pour une architecture à mémoire virtuellement partagée revient au final à expliciter les transferts de données entre threads. Des environnements de programmation comme *Fortran-S* [BKP93] ont été conçus dans cette optique.

1.3 Discussion

L'apparition d'architectures parallèles a impliqué de véritables bouleversements dans le processus de programmation des machines de calcul. Les programmeurs ont consenti beaucoup d'efforts pour maitriser les mécanismes de la programmation concurrente et savoir en éviter les pièges. Les modèles de programmation parallèles présentés dans ce chapitre aident à tirer parti de ces machines, bien qu'aucun d'entre eux ne se pose comme la solution providentielle au problème de l'exploitation directe des architectures parallèles. On constate en effet que même si certaines approches de programmation parallèle comme OpenMP se veulent incrémentales (les annotations permettent de paralléliser un code séquentiel préexistant), la plupart des algorithmes doivent être repensés et donc réécrits pour exploiter ce parallélisme. La façon de programmer a donc évolué

avec les architectures: les programmeurs se sont formés à la programmation parallèle, ont investi du temps pour maitriser de nouveaux langages et ont (re)développé une quantité innombrable de bibliothèques et d'applications exploitant ces machines.

Les architectures multicœurs sont aujourd'hui partout, et les applications et bibliothèques développées jusqu'alors peinent à en tirer pleinement parti. Elles imposent en effet de nouvelles contraintes au programmeur d'applications parallèles. Certains langages parmi ceux présentés dans ce chapitre bénéficient aujourd'hui d'un regain d'intérêt: la communauté les utilisait déjà pour programmer les architectures SMP et NUMA, et souhaite naturellement capitaliser cette expérience pour apprendre à maitriser les architectures multicœurs. Le chapitre 2 présente les spécificités architecturales des technologies multicœurs et les moyens dont on dispose pour les programmer, parmi lesquels on retrouve des évolutions de certains langages présentés précédemment.

Chapitre 2

Motivation et état de l'art : la révolution du multicœur

Sommaire

2.1 L'avènement du multicœur 19
2.1.1 Des processeurs multithreads aux architectures multicœurs . . 20
2.1.1.1 Masquer la latence d'accès à la mémoire 20
2.1.1.2 Optimiser l'utilisation du pipeline d'instructions . . . 21
2.1.2 Plusieurs cœurs sur une même puce 21
2.1.3 Des machines multicœurs aux accès mémoire non-uniformes . 22
2.2 Programmation des architectures multicœurs 24
2.2.1 Outils et supports exécutifs spécifiques 24
2.2.1.1 Support logiciel pour la programmation des multicœurs 24
2.2.1.2 Supports exécutifs conçus pour les architectures hiérarchiques 25
2.2.2 Support des compilateurs 26
2.2.3 Langages de programmation 27
2.3 Discussion 29

Les processeurs multicœurs sont aujourd'hui omniprésents, depuis les ordinateurs portables grand public jusqu'aux supercalculateurs des centres de recherche. Les programmeurs d'applications parallèles, contraints de s'adapter à ces nouvelles architectures, peinent à en tirer la quintessence. Pour comprendre ces difficultés, nous examinons dans ce chapitre les caractéristiques des architectures multicœurs avant d'aborder les moyens dont nous disposons pour les programmer.

2.1 L'avènement du multicœur

Cette section présente les avancées technologiques au sein du processeur qui ont conduit aux architectures multicœurs.

2.1.1 Des processeurs multithreads aux architectures multicœurs

Les progrès en finesse de gravure, offrant toujours plus de place et donc de composants aux architectes, ont été exploités pour augmenter la fréquence, principalement en allongeant et en complexifiant les pipelines. Ainsi lors de la conception de l'architecture *Netburst* [HSU+], Intel planifiait de produire un pipeline de 45 étages pour monter à la fréquence de 7Ghz. Ce projet ne put aboutir en raison de problèmes liés à la dissipation thermique. Mais alors comment traduire les progrès technologiques de la gravure en performance sans augmentation de fréquence? En améliorant encore la prédiction de branchement? La qualité des prédicteurs actuels est telle qu'il ne reste plus que quelques pourcents à gagner dans ce domaine. Augmenter la taille des caches? La latence d'accès à une mémoire cache est d'autant plus importante que cette mémoire est grande. Les architectes disposaient donc d'une marge de manœuvre limitée pour faire évoluer les mécanismes existants.

Une solution possible était de consacrer plus de silicium aux unités de calculs qu'aux coûteux mécanismes de réordonnancement ou d'allocation dynamique de registres et de laisser le compilateur conduire de façon statique l'évaluation spéculative et l'exécution parallèle des instructions au sein du processeur. Un retour en arrière en quelque sorte. C'est cet esprit qui a animé HP et Intel au milieu des années 1990 pour définir l'architecture EPIC, pour *Explicit Parallel Instruction Computer* [SRM00] qui a conduit aux processeurs Itanium des années 2000-2010. Cependant la combinatoire introduite par la souplesse du jeu d'instructions de l'Itanium est telle qu'il n'existe toujours pas de compilateur capable de bien exploiter ce processeur. Aussi, les architectes ont exploré une autre voie : la possibilité de juxtaposer plusieurs processeurs généralistes indépendants sur une même puce et donc de s'appuyer sur les threads (et les programmeurs) pour améliorer le rendement des processeurs. L'introduction de la technologie dite du *multithreading* est présentée dans cette section.

2.1.1.1 Masquer la latence d'accès à la mémoire

Le *multithreading*, abordé ici d'un point de vue matériel, autorise plusieurs threads à se partager les unités fonctionnelles d'un processeur. Les processeurs multithreads proposent la gestion de plusieurs contextes de threads en parallèle. Pour que le multithreading soit exploité de façon efficace, le processeur doit être capable de passer d'un thread à un autre en un temps très court. En particulier, un changement de contexte de threads doit être bien plus rapide qu'un changement de contexte de processus, qui lui peut prendre plusieurs centaines de cycles. On distingue deux technologies principales parmi les processeurs multi-threads: le *Coarse-grained Multithreading* (CMT) et le *Fine-grained Multithreading* (FMT).

Les processeurs FMT passent d'un thread prêt à un autre à chaque instruction, les threads en attente de données n'étant pas éligibles pour être exécutés. La technologie FMT masque ainsi le temps nécessaire pour récupérer une donnée en mémoire en passant la main à un autre thread prêt. Ce fonctionnement correspond à l'implantation, à l'intérieur du processeur, de l'ordonnancement préemptif des systèmes d'exploitation actuels. Les machines *HEP* [Smi81] et *Tera* [ACC+90] furent parmi les premières à disposer de processeurs FMT, et permettaient par exemple la gestion de 128 contextes de threads en parallèle.

La technologie CMT diffère du FMT dans le sens où le processeur change de contexte seulement en cas d'attente prolongée, par exemple celle induite par un défaut de cache L2. Le processeur *PULSAR* [BEKK00], développé par IBM dans les années 2000, permettait par exemple la gestion de deux contextes de threads en parallèle. Il était possible sur ce processeur de changer de contexte de threads en 3 cycles. La vitesse de changement de contexte est ici un facteur moins critique, simplifiant la conception de processeurs multithreads CMT, qui restent cependant moins réactifs que les processeurs FMT quand il s'agit de masquer des attentes courtes. On retrouve par exemple cette technologie au sein de processeurs *Intel Itanium*.

2.1.1.2 Optimiser l'utilisation du pipeline d'instructions

Pour améliorer l'occupation des unités fonctionnelles, les architectes ont rendu possible l'exécution concurrente de plusieurs flots d'instructions indépendants par un seul et même pipeline. Cette amélioration passe par la duplication du jeu de registres nécessaire à l'exécution d'un flot d'instruction. Cette duplication donne naissance à autant de *processeurs logiques* perçus comme de véritables processeurs physiques par le système d'exploitation. On parle ici d'architecture SMT, pour *Simultaneous Multi-Threading* [EEL+97], ou encore d'*Hyperthreading* si on parle de l'implantation de cette technique par Intel. Elle apparait pour la première fois dans la série *Northwood* du processeur *Intel Pentium 4*. Un processeur SMT capable de partager son pipeline entre deux flots d'instructions est qualifié de processeur SMT à deux voies.

Bien que perçus comme un bi-processeur par le système d'exploitation, les deux processeurs logiques se partagent la mémoire cache, la bande passante et les unités fonctionnelles du processeur physique. Les performances obtenues par un processeur SMT à deux voies sont de ce fait difficiles à prédire.

2.1.2 Plusieurs cœurs sur une même puce

Alors que la fréquence des processeurs n'augmente plus, la finesse de gravure des composants qui le constituent continue de progresser. Cette miniaturisation offre tellement d'espace sur le silicium qu'il est possible de graver plusieurs processeurs sur une même puce. On parle alors de processeur multicœur.

Les processeurs multicœurs ont depuis plusieurs années déjà envahi le marché grand public. Les principaux fondeurs de processeurs comme AMD, Intel, IBM ou encore SUN proposent leur propre gamme de processeurs multicœurs, qui se démarquent le plus souvent les uns des autres par l'agencement des cœurs et des mémoires cache sur une puce. Le principe de conception général reste le même : plusieurs cœurs, semblables[1] aux processeurs physiques de l'ère pré-multicœur, sont gravés sur la même puce. Ils communiquent ensemble via différents niveaux de cache partagés. L'organisation de ces caches dépend là encore du processeur, mais tous adoptent le même schéma de conception: un cœur dispose d'un cache de niveau 1 privé, très rapide mais de taille limitée. Il accède aussi à un ou plusieurs caches de niveaux supérieurs, qui sont moins

[1] La longueur de leurs pipelines est tout de même divisée par trois.

rapides mais de plus grande taille. Ces caches sont généralement partagés entre plusieurs cœurs, facilitant ainsi les communications entre ceux-ci. Le tableau 2.1 illustre ces différences de temps d'accès. Un programme synthétique créant deux threads qui incrémentent un compteur global de manière concurrente est exécuté sur une machine à deux processeurs quadri-cœurs *Opteron* [KMAC03]. À l'intérieur d'un processeur, chaque cœur dispose de sa propre mémoire cache de niveau 1, partage un cache de niveau 2 avec un de ses voisins, et accède à un cache de niveau 3 partagé entre les 4 cœurs du processeur. On observe ainsi que les communications entre les cœurs 0 et 1, partageant du cache de niveau 2, sont plus performantes que celles faisant intervenir les cœur 2 et 3, qui communiquent avec le cœur 0 via un cache plus lent.

Emplacement	Cœur 1	Cœur 2	Cœur 3	Cœur 4	Cœur 5	Cœur 6	Cœur 7
Cœur 0	16,6	20,0	20,0	25,2	25,8	26,0	26,2

TAB. 2.1 – Temps d'exécution, en microsecondes, d'un programme synthétique créant deux threads qui incrémentent de manière concurrente un compteur global, en fonction de l'emplacement des threads sur une machine à deux processeurs à quatre cœurs Opteron.

Les architectures actuelles sont de véritables poupées russes. Les constructeurs sont capables de produire des topologies fortement hiérarchiques, composées de blocs de processeurs agencés en parallèle, connectés ensemble via un réseau d'interconnexion propriétaire (Quickpath pour Intel, HyperTransport pour AMD). Les processeurs de ces machines peuvent être multicœurs et SMT. Il est donc possible de construire des architectures parallèles à mémoire partagée disposant d'un très grand nombre d'unités de calcul. L'interconnexion des processeurs multicœurs aggrave encore le relief des architectures parallèles contemporaines. En effet, on remarque déjà que certains cœurs d'un même processeur sont capables de communiquer de façon privilégiée via un cache partagé. La possibilité de connecter des puces multicœurs ensemble ressucite les modèles de conception d'architectures à accès mémoire non-uniformes. En effet, pour limiter les effets de la contention mémoire, les architectes en reviennent à distribuer la mémoire en bancs attachés à des blocs d'un ou plusieurs processeurs multicœurs. Les architectures multicœurs exposent donc aux programmeurs les contraintes liées aux architectures NUMA, auxquelles s'ajoutent les spécifités matérielles inhérentes aux différents niveaux de caches partagés. On reparle ainsi de facteur NUMA pour les machine de calcul contemporaines, et rien n'indique que cette tendance ne soit pas suivie par les constructeurs vis-à-vis du grand public.

2.1.3 Des machines multicœurs aux accès mémoire non-uniformes

En pratique, la mémoire vive de la machine est répartie en plusieurs bancs. Chaque banc se trouve physiquement proche d'un ensemble de processeurs multicœurs. On peut alors revenir à la terminologie des machines à accès mémoire non-uniforme et remarquer que ces processeurs font partie d'un même nœud NUMA. Ainsi, une machine multicœur peut exhiber des latences d'accès à la mémoire différentes selon qu'un cœur accède à des données allouées sur le banc mémoire de son nœud ou qu'il accède à des

bancs mémoire distants. Il est facile de mettre en évidence ce phénomène à l'aide d'une expérience synthétique faisant intervenir des lectures et écritures sur des données que l'on alloue tour à tour sur les différents nœuds d'une machine. L'architecture cible est présentée sur la figure 2.1. Elle se compose de quatre processeurs Opteron comportant quatre cœurs chacun. Un banc mémoire est attaché à chacun des processeurs: cette machine dispose donc de quatre nœuds NUMA interconnectés deux à deux de façon circulaire. Les latences mémoire mesurées à partir de cette machine sont présentées par le tableau 2.2. Quel que soit le type d'accès, l'accès à un banc mémoire distant est plus lent de 18 à 46%. De plus, accéder au banc mémoire opposé reste la pire des solutions sur cette architecture, puisqu'on doit alors traverser deux liens HyperTransport[2], et autant de contrôleurs mémoire. Il faut donc tenir compte de cette réalité architecturale lorsqu'on programme ce type de machines.

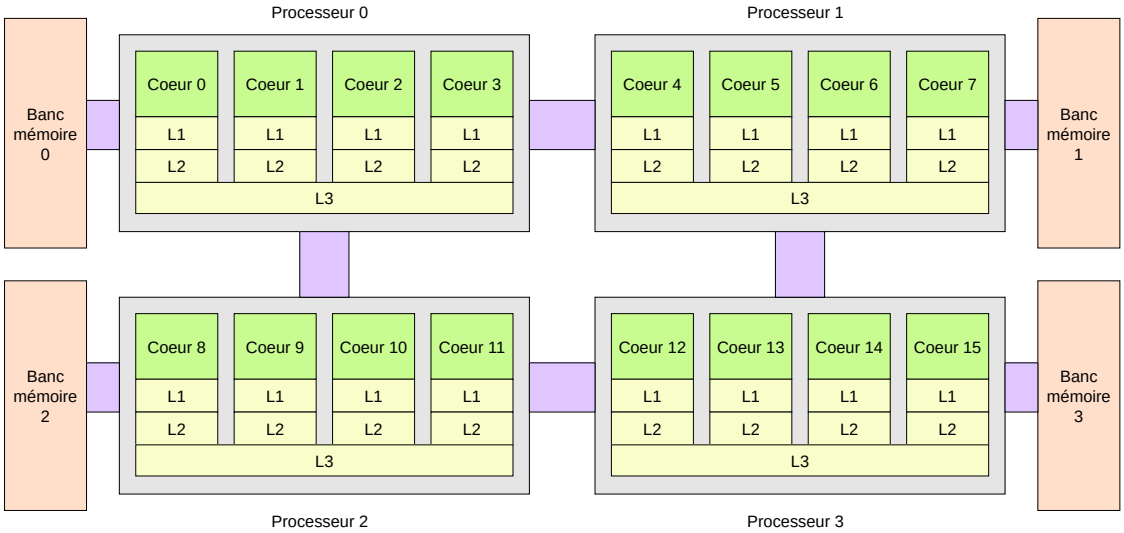


FIG. 2.1 – Architecture multicœur composée de 4 processeurs à 4 cœurs de type Opteron.

Type d'accès	Accès local	Accès au banc voisin	Accès au banc opposé
Lecture	83 ns	98 ns (× 1,18)	117 ns (× 1,41)
Ecriture	142 ns	177 ns (× 1,25)	208 ns (× 1,46)

TAB. 2.2 – Latences d'accès à la mémoire suivant l'emplacement physique des données sur une machine à 4 nœuds NUMA aux processeurs Opteron.

[2]Technologie d'interconnexion d'AMD

2.2 Programmation des architectures multicœurs

Le programmeur d'applications parallèles dispose de différents outils, techniques de programmation et extensions de langages pour exploiter les architectures multicœurs. Cette section présente certains d'entre eux parmi les plus représentatifs.

2.2.1 Outils et supports exécutifs spécifiques

Cette section présente les bibliothèques conçues pour aider le programmeur à tirer parti des architectures multicœurs.

2.2.1.1 Support logiciel pour la programmation des multicœurs

CPU affinity La bibliothèque standard C propose une interface de programmation pour le placement explicite des processus sur les unités de calcul de la machine. Le programmeur manipule des ensembles de processeurs, les *cpu_set*, sur lesquels il peut cantonner l'exécution d'un processus. La bibliothèque de threads *NPTL*, utilisée par le système d'exploitation Linux depuis sa version 2.6, implémente aussi cette fonctionnalité pour permettre au programmeur de placer explicitement les threads de son application sur les différents cœurs de la machine. Cette démarche, bien que peu portable, reste très utile pour expérimenter différents placements de threads et analyser les performances qui en découlent.

libNUMA *libNUMA* est une bibliothèque qui étend les fonctionnalités de placement des threads du système d'exploitation Linux aux données. Elle permet donc, en plus de pouvoir spécifier le ou les cœurs sur lesquels un thread s'exécutera de façon privilégiée, de définir le placement des données allouées dynamiquement. En pratique, le programmeur utilise une API particulière qui remplace les appels aux traditionnels `malloc` et `free` de la *libc* par des appels de fonctions de la bibliothèque libNUMA. Cette bibliothèque propose plusieurs politiques d'allocation, comme l'allocation explicite sur un nœud NUMA particulier, ou encore la distribution cyclique des pages mémoire associées à une donnée sur un ensemble de processeurs. Bien que très utile pour expérimenter différentes répartitions mémoire, l'utilisation de libNUMA demande un savoir faire certain pour l'être de façon portable et performante puisque certaines fonctionnalités nécessitent une connaissance précise de l'architecture sous-jacente pour être employées efficacement. Il est de plus difficile de s'adapter à des changements de motifs d'accès à la mémoire en ne s'appuyant que sur cette bibliothèque, étant donné qu'elle ne fait remonter aucune information à l'application sur l'occupation des bancs mémoire ou les statistiques d'accès aux différents nœuds par exemple.

hwloc Initialement développée au sein de l'équipe Runtime, la bibliothèque HWLOC [BOM+10] a pour but de construire une représentation générique de toute architecture. Cette modélisation intègre des informations détaillées sur les différents niveaux de mémoire de la machine, leur organisation, leur taille, mais aussi leur connexion avec

les unités de calcul, du processeur logique SMT jusqu'à l'interconnexion de puces multicœur que forment les nœuds NUMA. La figure 2.2 montre la sortie obtenue par l'utilitaire `lstopo` de cette bibliothèque, exécuté sur une machine bi-quadricœur Opteron. La bibliothèque fournit des fonctions de parcours de l'architecture, ainsi que des primitives pour fixer l'exécution de threads et de processus sur un élément de la topologie.

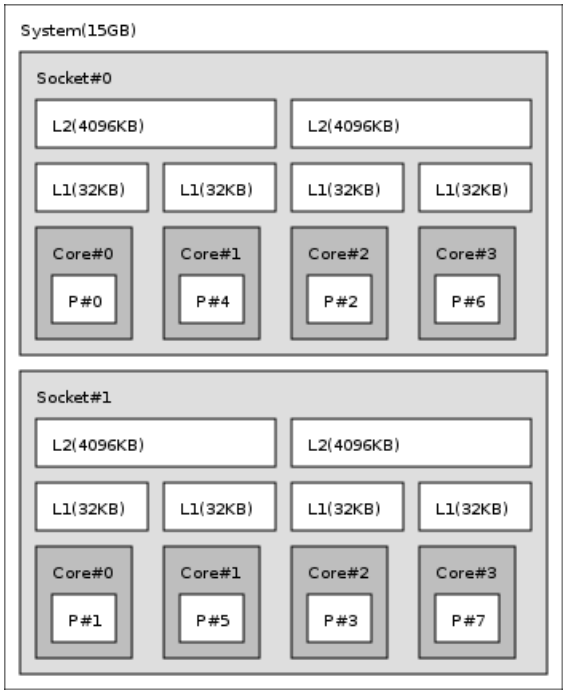


FIG. 2.2 – Architecture bi-quadricœur Opteron modélisée par la bibliothèque hwloc.

2.2.1.2 Supports exécutifs conçus pour les architectures hiérarchiques

McRT *Manycore RunTime* est un support exécutif développé par Intel pour aider le programmeur à exploiter les architectures multicœurs. Il implémente les primitives de création et de gestion de threads légers, intégralement conçus en espace utilisateur. Il propose aussi le mécanisme de *futures*, aussi appelé dans la littérature création paresseuse de threads: on peut voir un *future* comme un thread sans pile. Au moment de son exécution, il réutilise si possible une pile préallouée. Ce mécanisme est particulièrement efficace pour paralléliser les algorithmes de type *diviser pour régner*. En outre, l'extension *McRT-STM* [SAtH$^+$06] de ce support exécutif s'appuie sur le principe de mémoire transactionnelle, inspiré de la gestion des bases de données, pour simplifier le partage de données d'une application parallèle. Enfin, ce support exécutif donne la possibilité au programmeur de partitionner la machine, sous forme de *domaines d'ordonnancement*, pour exécuter plusieurs applications de manière concurrente sans perturbation.

BubbleSched La plateforme *BubbleSched* [Thi07], développée pendant la thèse de Samuel Thibault, permet de structurer le parallélisme des applications en regroupant les threads en relation dans une structure baptisée *bulle*. Une bulle peut en contenir plusieurs autres, créant ainsi un arbre de threads comme représenté sur la figure 2.3. La projection de ce parallélisme sur l'architecture est assuré par un ordonnanceur à bulles qui s'appuie sur la modélisation de la topologie offerte par `hwloc` pour placer les différents groupes de threads de l'application. BubbleSched fournit un certain nombre d'ordonnanceurs pour palier les besoins applicatifs les plus courants, comme l'équilibrage de charge sur la machine par exemple. La plateforme fournit aussi une interface de programmation spécifique sur laquelle l'expert pourra s'appuyer pour créer sa propre politique d'ordonnancement.

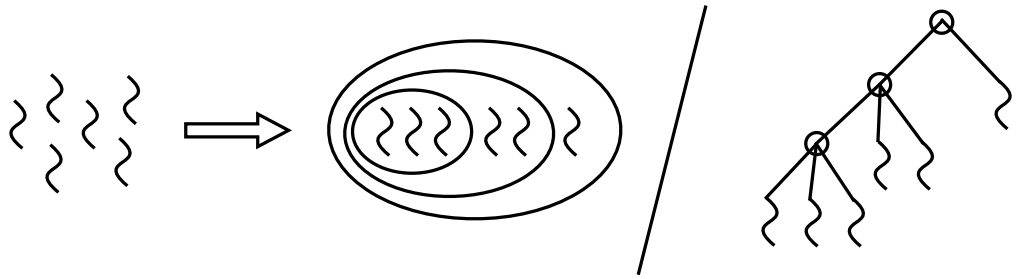


FIG. 2.3 – Structuration de parallélisme à l'aide des bulles.

2.2.2 Support des compilateurs

Compilation itérative La *compilation itérative* est un procédé souvent utilisé par la bibliothèques d'algèbre linéaire qui consiste à générer un grand nombre de binaires intermédiaires à la compilation d'un programme parallèle, qui diffèrent essentiellement par les optimisations qui leur sont appliquées. Ces binaires sont testés tour à tour en cours d'exécution du programme. L'analyse des résultats de ces tests dirige le compilateur pour les étapes suivantes de la compilation. En contrepartie d'un temps de compilation très supérieur à une compilation classique, la compilation itérative permet d'obtenir un binaire particulièrement optimisé pour l'architecture de la machine cible. Cette approche offre parfois des résultats très satisfaisants, comme le montrent les travaux [WCO+08] menés par Williams et al., où l'optimisation par compilation itérative d'un algorithme mettant en œuvre les techniques numériques de *Boltzmann* permet d'obtenir des performances 14 fois supérieures à celles obtenues par compilation classique.

Gestion mémoire et architectures NUMA Pousa et al. ont montré à plusieurs reprises [PCMC10, BCFP+09] l'intérêt de placer les données d'une application en accord avec la topologie mémoire de l'architecture sous-jacente. Ils proposent une plateforme de gestion des données, et plus particulièrement des affinités mémoire, pour les architectures NUMA. Cette plateforme, baptisée *Minas*, se décompose en trois principaux

modules. Le premier d'entre eux, *MAi*, offre une interface de programmation de haut niveau implémentant des mécanismes d'allocation et de placement de données. Le but de MAi est de proposer au programmeur une boîte à outils complète lui permettant de maitriser les accès mémoire de son application sur n'importe quelle architecture NUMA. Le deuxième module, *MAPP*, offre une solution automatique d'application de politiques mémoire à la compilation. Le préprocesseur MAPP est ainsi capable de détecter les tableaux de taille supérieure à la taille du cache de plus haut niveau de la machine et de leur appliquer une politique particulière d'allocation: allocation sur un nœud NUMA particulier, distribution des pages mémoire en tourniquet sur les différents nœuds, etc. Le dernier module, *numarch*, récupère les informations de topologie nécessaires au placement des données sur la machine.

2.2.3 Langages de programmation

OpenMP 3.0 La version 3.0 du standard OpenMP, publiée en mai 2008, étend l'expressivité du langage en intégrant la possibilité d'écrire des applications parallèles sous forme de tâches [ACD+09]. Ces tâches, créées grâce à une nouvelle directive de compilation, sont générées à l'intérieur d'une région parallèle. Elles sont ensuite exécutées, de façon indépendante, par les threads de la région parallèle dont elles sont issues. En pratique, le standard limite la possibilité de créer des tâches à un seul thread de l'équipe, les autres exécutant les tâches au fur et à mesure de leur création. Cette nouvelle fonctionnalité étend l'expressivité du langage: certains algorithmes s'écrivent naturellement sous forme de tâches, alors que leur écriture en OpenMP 2.5 pouvait s'avérer fastidieuse et inélégante. C'est le cas par exemple des algorithmes récursifs, comme le calcul d'un terme de la suite de Fibonacci présenté en figure 2.4. Bien que cette nouveauté étende encore un peu plus le spectre des applications OpenMP, elle ne résoud en rien les problèmes de performance que peut rencontrer ce langage sur les architectures multicœurs. En effet, rien dans la norme 3.0 ne permet de définir des affinités entre threads et données par exemple, ou d'expliciter qu'une zone mémoire doit être répartie sur la machine car utilisée par plusieurs tâches. La répartition des threads sur la machine est pour le moment elle aussi laissée à la discrétion du support exécutif, voire du système d'exploitation. De plus, il manque encore au parallélisme de tâches d'OpenMP la possibilité d'exprimer des dépendances entre les tâches, ce qui limite les possibilités d'optimisations, puisque seules les synchronisations explicites entre toutes les tâches d'une région parallèle sont pour l'instant disponibles.

Plusieurs extensions d'OpenMP ont été proposées pour améliorer les performances du langage sur les architectures multicœurs. Chapman et al. [CHJ+06] étendent les mécanismes de partage de travail d'OpenMP. Ils permettent ainsi au programmeur de définir des sous-groupes d'équipes, appelés *subteams*, sur lesquels il est possible de distribuer du travail à l'aide d'un nouveau mot-clé, `onthreads`. Cette distribution s'effectue à partir de la numérotation logique des threads OpenMP et non de leur emplacement sur la machine. Schmidl et al. [STMB10] proposent une extension du langage pour placer explicitement les équipes issues de régions parallèles imbriquées. Ils montrent une amélioration des performances de plusieurs benchmarks lorsqu'on dédide du placement des threads des régions parallèles imbriquées. Dimakopoulos et al. [HD07] s'inspirent

```
int
main (int argc, char **argv)
{
   [...]
#pragma omp parallel
  {
#pragma omp single
    fib(input);
  }
}
```

```
int
fib (int n)
{
   if (n < 2)
     return n;

   int x, y;

#pragma omp task shared(x)
   x = fib (n - 1);

#pragma omp task shared(y)
   y = fib (n - 2);

#pragma omp taskwait
   return x+y;
}
```

FIG. 2.4 – Calcul d'un terme de la suite de Fibonacci à l'aide de tâches OpenMP 3.0.

du langage Cilk pour la gestion du parallélisme imbriqué d'OpenMP. Ils reprennent le mécanisme de vol de travail mais l'adaptent de façon à voler depuis son voisinage en priorité, ce qui leur permet d'améliorer les performances d'applications OpenMP dans lesquelles les threads communiquent beaucoup.

TBB TBB, pour *Threads Building Blocks* [TBB], est une bibliothèque C++ générique développée par Intel, qui vise à étendre le langage C++ pour qu'il tire efficacement parti des architectures multicœurs. La bibliothèque TBB prone une approche de haut niveau pour générer du parallélisme à partir de programmes C++. Au lieu de s'appuyer sur une bibliothèque de processus légers, le programmeur se limite ici à exprimer les portions de code pouvant être exécutées de manière concurrente sous forme de tâche TBB. Ces tâches à la gestion très légère sont exécutées sur tout type d'architecture multicœur de manière transparente. A l'instar d'OpenMP, TBB détecte automatiquement le nombre de cœurs dont il dispose. La bibliothèque fournit un certain nombre de conteneurs, d'objets et d'itérateurs C++ parallèles qui permettent au programmeur C++ de ne pas avoir à modifier sa façon de programmer pour exploiter les différents cœurs de la machine. C'est un argument qui pèse en la faveur de TBB, par rapport à OpenMP, du côté de la communauté des programmeurs C++. Les programmeurs parallèles expérimentés peuvent aussi s'appuyer sur des interfaces de programmation de plus bas niveau, qui permettent par exemple de contrôler l'ordre d'exécution des tâches. TBB s'inspire du langage Cilk pour l'organisation de ses tâches en files de travail, et l'équilibrage de la charge par vol de tâches depuis ces files entre les différents cœurs de la machine.

Depuis la version 3.0 de la bibliothèque, tout thread a la possibilité de devenir maître d'une nouvelle arène (i.e. créer un groupe isolé de tâches) et de déclarer le nombre de

threads esclaves nécessaires. Un ordonnanceur gère l'affectation des threads esclaves aux arènes : à la terminaison d'une arène, les threads qui la propulsaient sont réaffectés en favorisant les groupes les plus déficitaires. Ce mécanisme de partionnement des threads fait de la bibliothèque TBB l'une des rares bibliothèques de programmation parallèle à avoir été pensée pour être *composable*. Par composabilité, on entend ici la faculté d'exécuter parallèlement différentes briques de calculs elles-même parallèles sans dégradation sensible des performances.

Outre sa complexité relative de mise en oeuvre TBB pèche par son ignorance de l'architecture sous-jacente, les affinités potentielles entre tâches d'une même arêne ne sont donc pas exploitées matériellement. De plus le principe des arênes n'est pas récursif, il serait de toute façon inutile sans traduction matérielle.

Manticore *Manticore* [FR] est une plateforme compilateur / support exécutif dédiée à un langage fonctionnel parallèle permettant l'expression de deux types de parallélismes, *implicite* et *explicite*. Le parallélisme *explicite* permet au programmeur de contrôler la création et la synchronisation des threads de son application. Cette synchronisation s'effectue par passages de messages synchrones, ce mécanisme étant mieux adapté aux langages fonctionnels que la communication par mémoire partagée. Le parallélisme *implicite* quant à lui autorise le programmeur d'applications à inciter le support exécutif, à l'aide d'annotations, à ce concentrer sur certaines parties du code pour en extraire automatiquement le parallélisme. En particulier, ce paradigme permet au programmeur d'écrire son application sous forme séquentielle. Certaines constructions, comme les boucles et les accès aux tableaux, seront parallélisées de façon automatique. Manticore s'appuie aussi sur le parallélisme implicite

Un aspect novateur de Manticore est permettre la définition d'ordonnanceurs à partir d'un ensemble de routines basé sur les continuations, la préemption et la migration entre processeurs virtuels. Une pile d'actions d'ordonnancement associée à chaque processeur virtuel est au coeur de ce dispositif. Lors de toute préemption, la fonction préemptée est sauvergardée sous forme de continuation, une action d'ordonnancement est alors dépilée et appliquée à la continuation. Cette technique permet de définir de façon élégante des ordonnanceurs et de les composer. Par exemple un ordonnanceur est défini pour optimiser de façon spéculative l'évaluation d'expressions (calcul de `ou` en parallèle, par exemple). Ici, aussi l'architecture de la machine n'est pas prise en compte.

2.3 Discussion

Afin d'être exploitées efficacement, les architectures multicœurs requièrent, de la part du programmeur, l'expression d'un parallélisme:

1. **massif**, pour suivre l'évolution rapide du nombre de cœurs d'une machine multicœur ;
2. **adapté** à l'organisation hiérarchique des unités de calcul et des différents niveaux de mémoire de ces architectures.

Des efforts ont été fournis à la fois pour encourager l'expression de parallélisme, avec par exemple la possibilité d'exprimer du parallélisme à grain fin depuis OpenMP 3.0, et faciliter le développement d'applications parallèles, grâce notamment au développement de techniques comme la mémoire transactionnelle pour faciliter la synchronisation. En revanche, l'optimisation des applications parallèles sur architecture multicœur reste une affaire de spécialistes. Pour améliorer les performances de son application, le programmeur en arrive à devoir *tout contrôler*, du placement des traitements parallèles sur les unités de calcul jusqu'à l'organisation de ses données sur les différents bancs mémoire. Ces optimisations nécessitent une bonne connaissance de l'achitecture sous-jacente et sont souvent très difficiles à concilier avec l'écriture d'un code portable. De son côté, le support exécutif dispose d'une vue générale de l'architecture et des threads de l'application. Malgré cela, il peine à prendre des décisions adéquates quant au placement de ces threads et des données qu'ils accèdent sans intervention du programmeur [NPP+00].

Chapitre 3

Contribution: Structurer pour régner!

Sommaire

3.1 Capturer l'information et la pérenniser 32
3.1.1 Quel parallélisme pour les architectures multicœurs? 32
3.1.2 Structurer le parallélisme de façon pérenne 33
3.1.3 OpenMP, un langage de programmation parallèle structurant . 34
3.1.4 Des indications pérennes pour une exécution structurée des programmes OpenMP 35
3.2 Projeter dynamiquement le parallélisme sur la topologie 35
3.2.1 Expression générique des architectures hiérarchiques 36
3.2.2 Ordonnancement dynamique de parallélisme structuré 36
3.2.2.1 Différents besoins, différents ordonnanceurs 37
3.2.3 Cache: un ordonnanceur qui respecte les affinités entre threads 38
3.2.3.1 Répartir les groupes de threads de façon compacte . . 38
3.2.3.2 Equilibrer la charge quand une unité de calcul devient inactive 40
3.2.4 Memory: un ordonnanceur qui tient compte des affinités mémoire 44
3.2.4.1 Débit mémoire et localité 44
3.2.4.2 Support logiciel pour la gestion des données sur les architectures NUMA 46
3.2.4.3 Transmission des affinités mémoire au support exécutif 46
3.2.4.4 Ordonnancement conjoint de threads et de données . 49
3.3 Composer, confiner, dimensionner 51
3.3.1 Composition de parallélisme 53
3.3.2 Confinement de parallélisme 54
3.3.2.1 Partitionnement de l'ensemble des ressources de calcul 55
3.3.2.2 Gestion de la surcharge en environnement confiné . . 56
3.3.3 Dimensionnement de régions parallèles OpenMP 59
3.4 Synthèse 59

Notre contribution s'articule en trois axes principaux :

1. **capturer** la structure du parallélisme exprimée par le programmeur d'applications OpenMP et la transmettre au support exécutif afin qu'il puisse la consulter à n'importe quel moment ;
2. **projeter** dynamiquement ce parallélisme structuré sur les architectures hiérarchiques ;
3. **composer** notre environnement de programmation avec d'autres bibliothèques parallèles.

Ce chapitre développe chacun de ces axes.

3.1 Capturer l'information et la pérenniser

L'obtention d'une solution portable et performante passe, de notre point de vue, par une coopération forte entre le programmeur d'applications et le support exécutif. En particulier, le programmeur doit être capable de transmettre des informations sur la structure du parallélisme de son application pour aider l'ordonnanceur à prendre les bonnes décisions. Cette section détaille les procédés nous permettant de capturer ces informations et de les pérenniser afin qu'elles puissent être consultées par le support exécutif.

3.1.1 Quel parallélisme pour les architectures multicœurs?

Les machines de calcul contemporaines exposent un grand nombre d'unités de calcul. Leur exploitation efficace passe par l'expression d'un parallélisme massif, adapté aux contraintes architecturales introduites par les processeurs multicœurs, comme le partage de mémoire cache entre plusieurs unités de calcul, ou encore le morcellement de la mémoire en bancs répartis à différents endroits de la topologie.

Une majorité de modèles de programmation pour les architectures à mémoire partagée ont été pensés pour des architectures de type SMP. Ils exposent le plus souvent un parallélisme *"à plat"* qui s'ordonnance naturellement sur les architectures multiprocesseurs symétriques. L'organisation de ce parallélisme, constitué de threads travaillant de façon identique, correspond en effet à l'agencement des processeurs d'une telle architecture. Les bonnes performances obtenues alors par ces modèles de programmation les ont rendu populaires auprès de la communauté du calcul hautes performances.

De nombreuses applications et bibliothèques s'appuyant sur ces modèles de programmation ont ainsi vu le jour, et les performances qu'elles obtiennent sur les architectures multicœurs d'aujourd'hui sont souvent décevantes. En effet, L'arrivée des processeurs multicœurs compléxifie encore un peu plus la tâche du programmeur d'applications parallèles. Le partage de plusieurs niveaux de cache, l'organisation hiérarchique des unités de calcul ou encore la réapparition de facteurs NUMA sont autant de réalités architecturales dont il faut tenir compte pour exploiter efficacement ces architectures. Un parallélisme *"à plat"*, comme celui exprimé par la majorité des modèles de programmation parallèle, reste peu adapté à l'expression de relations d'affinités entre les traite-

ments parallèles d'une application. Il est donc nécessaire de le structurer pour l'adapter aux architectures hiérarchiques. La bibliothèque BubbleSched, sur laquelle s'appuie notre solution, fournit les fonctionnalités nécessaires à la structuration du parallélisme et à son ordonnancement sur les architectures hiérarchiques.

3.1.2 Structurer le parallélisme de façon pérenne

Cette plateforme, développée au sein de l'équipe Runtime et issue des travaux de thèse de Samuel Thibault [Thi07], permet la création et le contrôle de l'exécution de groupes de threads, en introduisant le concept de *bulle*. Une bulle est une structure visible par l'ordonnanceur qui peut contenir des threads. Il est aussi possible d'inclure une bulle dans une autre, modélisant ainsi le parallélisme d'une application sous forme arborescente. Les threads d'une même bulle sont considérés par le support exécutif comme étant en relation. Une bulle peut par exemple exprimer que les threads qu'elle contient accèdent aux mêmes données ou encore effectuent fréquemment des opérations collectives. En d'autres termes, les bulles traduisent des besoins spécifiques en ordonnancement consultables par le support exécutif *tout au long de l'exécution du programme*. Ces bulles, ordonnançables au même titre qu'un thread, peuvent être réparties sur l'architecture à l'aide d'un ordonnanceur spécifique, ou ordonnanceur à bulles. La plateforme BubbleSched propose une interface de programmation pour la conception d'ordonnanceurs à bulles qui peut être utilisée pour répondre à des besoins d'ordonnancement très spécifiques. Pour répondre aux besoins les plus classiques, comme par exemple occuper tous les processeurs de la machine, BubbleSched fournit aussi quelques implémentations de politiques d'ordonnancement, sur lesquelles un expert peut s'appuyer pour en développer de nouvelles. Il est aussi possible d'intégrer de nouveaux critères, sous forme d'informations attachées aux bulles, aux politiques d'ordonnancement. Ces critères peuvent être explicités par le programmeur ou maintenus automatiquement par le support exécutif, comme par exemple le dernier cœur sur lequel un thread s'est exécuté, ou encore le dernier niveau de la topologie sur lequel il a été ordonnancé. Certains d'entre eux, comme la charge de travail associée à un groupe de threads ou encore la quantité de données auxquelles il accède, sont traités par BubbleSched comme des statistiques maintenues dynamiquement, facilement factorisables dans les bulles et consultables de façon efficace par l'ordonnanceur. Par exemple, spécifier la charge de travail d'un thread aura ainsi pour effet de mettre automatiquement à jour la charge de travail de la bulle à laquelle il appartient.

BubbleSched nous offre ainsi la possibilité de structurer le parallélisme des applications à l'aide de bulles et de décider de leur agencement sur la machine en créant son propre ordonnanceur. Comme on a pu l'observer par le passé sur les architectures SMP, les applications sont d'autant plus performantes que la structure du parallélisme qu'elles expriment s'adapte à l'architecture de la machine qui les exécute. Les architectures multicœurs exhibent plusieurs niveaux de mémoire. Une façon naturelle de les programmer consiste à exprimer plusieurs niveaux de parallélisme. A chacun de ces niveaux peut en effet correspondre une classe d'affinité, ainsi qu'une granularité. Les parallélismes les plus externes, à gros grain, définissent alors un découpage de l'application en traitements à répartir sur les différents bancs mémoire de la machine, et veillent ainsi

au respect des affinités entre les traitements et les données auxquelles ils accèdent. Les parallélismes les plus internes divisent eux ce premier découpage en tâches plus fines capables de communiquer efficacement via les différents niveaux de cache partagés.

Les applications et bibliothèques du domaine du calcul hautes performances nécessitent donc d'être repensées: rares sont celles exhibant plusieurs niveaux de parallélisme. BubbleSched peut être utilisée comme une bibliothèque de threads et être appelée directement depuis les applications parallèles. Cette approche permet parfois d'obtenir les meilleurs résultats en termes de performance, mais la gestion explicite des bulles et de leur ordonnancement reste très technique à mettre en œuvre. C'est pourquoi nous proposons de s'appuyer sur un langage de programmation parallèle pour générer automatiquement ces bulles. Parmi ces langages, OpenMP se dégage d'une part parce qu'il est très largement utilisé, et d'autre part parce qu'il est le mieux placé pour aider à structurer leur parallélisme, comme nous allons le voir dans la section suivante.

3.1.3 OpenMP, un langage de programmation parallèle structurant

Le modèle d'exécution d'OpenMP, de type *fork-join* ou graphe série-parallèle, pose déjà les bases d'une structuration forte de son parallélisme. Tout thread OpenMP est ainsi soumis à une relation de filiation: le thread entrant dans une région parallèle, ou *thread principal*, créé d'autres threads pour former une équipe qui exécute un bloc de code en parallèle. Les threads d'une même équipe travaillent ensemble, et sont donc en relation tant que l'équipe existe. L'utilisateur doit préciser les données partagées par chaque région parallèle. Il est donc possible d'identifier les threads amenés à accéder à certaines données du programme. La complétion d'une région parallèle OpenMP s'accompagne par la destruction des threads qui l'ont exécuté. Seul le thread principal y survit.

La gestion de plusieurs niveaux de parallélisme s'effectue de façon naturelle en OpenMP. Les threads d'une équipe OpenMP peuvent en effet eux aussi entrer dans une région parallèle, et créer à nouveau des threads. Chaque thread fait alors partie d'une nouvelle équipe qui traite généralement une portion plus fine du travail à accomplir en parallèle. Une fois ce traitement accompli, l'équipe est dissoute et son thread principal rejoint l'équipe de la région parallèle englobante. Il retrouve ainsi sa position initiale. Ainsi, bien qu'un thread n'appartient sémantiquement qu'à une seule équipe à la fois, il peut donc, au cours de sa vie, participer à l'exécution de plusieurs régions parallèles imbriquées.

On peut donc remarquer qu'OpenMP offre une structure naturelle de son parallélisme, aussi bien spatiale puisque qu'à tout moment de l'exécution du programme les threads OpenMP sont regroupés en équipes, que persistante puisqu'on peut facilement se rappeler de comment ont été générées ces équipes en maintenant une structure arborescente du parallélisme. Ces informations, correctement exploitées par le support exécutif, permettraient d'améliorer l'ordonnancement des équipes OpenMP sur les architectures hiérarchiques.

3.1.4 Des indications pérennes pour une exécution structurée des programmes OpenMP

La structure du parallélisme d'OpenMP exprime à elle seule des relations entre les traitements parallèles d'une application. En effet, l'entrée dans une région parallèle OpenMP provoque la création de threads qui s'exécuteront en parallèle en se partageant le travail associé au bloc de code correspondant. Ces threads sont donc en relation: ils sont d'une part sémantiquement associés, puisqu'ils sont issus d'une même région parallèle, et sont d'autre part amenés à partager des données et communiquer entre eux. Ces relations, facilement identifiables dans les programmes OpenMP, sont autant de précieux guides pour l'ordonnanceur du support exécutif. En effet, s'il lui était possible de consulter, à n'importe quel moment de l'exécution du programme, ces relations d'affinités, il disposerait de toutes les informations nécessaires pour par exemple regrouper les threads issus d'une même région parallèle sur les cœurs d'un même processeur multicœur. Il est donc nécessaire de mettre en place un mécanisme pour transmettre la structure du parallélisme depuis l'application OpenMP jusqu'au support exécutif sous-jacent. Notre solution s'appuie sur la structure de *bulle* de la plateforme d'ordonnancement *BubbleSched*.

BubbleSched, via son concept de bulle, permettrait de capturer la structure des programmes OpenMP en regroupant les threads d'une même région parallèle dans une bulle. Le support exécutif aurait alors les informations nécessaires pour orchestrer le parallélisme d'OpenMP sur des architectures à plusieurs niveaux de hiérarchies mémoire. Le support exécutif *ForestGOMP* est né de ce constat. Basé sur LIBGOMP, le support exécutif pour OpenMP de `GCC 4.2`, il opère à deux niveaux: il doit d'une part permettre la génération automatique de bulles depuis les programmes OpenMP, et d'autre part étendre BubbleSched en proposant des politiques d'ordonnancemement qui, dans un contexte OpenMP, s'adaptent bien aux architectures hiérarchiques.

Le modèle d'exécution d'OpenMP facilite grandement la génération automatique des bulles: à une équipe OpenMP correspond un groupe de threads en relation, donc une bulle. La structuration du parallélisme via BubbleSched singe ainsi en tout point l'organisation des threads OpenMP en équipes: les créations et destructions d'équipes impliquent des créations et destructions de bulles, les mouvements d'un thread d'une équipe à une autre engendre le même déplacement entre les bulles correspondantes. En particulier, l'entrée d'un thread dans une région parallèle imbriquée implique la création d'une bulle à l'intérieur d'une autre. Ainsi, à toute exécution de programme OpenMP correspond un arbre de bulles et de threads, dont la profondeur est définie par le niveau d'imbrication des régions parallèles qui le composent. La projection d'un tel arbre sur la topologie reste à la charge d'un ou plusieurs ordonnanceurs à bulles, comme explicité dans la section suivante.

3.2 Projeter dynamiquement le parallélisme sur la topologie

Maintenant que les programmes OpenMP génèrent des bulles de façon automatique, le support exécutif doit être capable d'orchestrer leur exécution de façon performante

et portable sur n'importe quelle architecture hiérarchique. Une façon d'y parvenir est pour lui de travailler sur une expression générique de la topologie de la machine.

3.2.1 Expression générique des architectures hiérarchiques

La connaissance de l'architecture par le support exécutif doit être d'autant plus poussée que les besoins exprimés par le programmeur sont précis. Par exemple, le support exécutif doit connaître les différents niveaux de cache partagés par les coeurs qui exécutent un groupe de threads en relation. Les informations que fournissent les systèmes d'exploitation sont souvent difficilement exploitables au niveau applicatif. Il n'est pas rare par exemple que la numérotation physique des processeurs ne reflète en rien la façon dont ils sont réellement connectés entre eux. De plus, pour que les stratégies d'ordonnancement implémentées soient portables, il est nécessaire de s'appuyer sur une vision générique de l'architecture.

La bibliothèque HWLOC propose un tel niveau d'abstraction. La plateforme BubbleSched s'appuie sur hwloc pour générer une représentation standardisée de la topologie. Elle associe des files d'ordonnancement à chaque élément de l'architecture, plus précisément à chacun d'entre eux disposant d'un niveau de mémoire (caches de niveaux L1, L2, L3, banc mémoire au sens NUMA, etc.). Ces files sont capable d'accueillir des threads et des bulles, et définissent des domaines d'ordonnancement spécifiques. Une file de niveau L1 contiendra des threads qui seront exécutés par le cœur correspondant à cette mémoire cache. De la même façon, ordonnancer une bulle sur une file de niveau nœud NUMA exprime que les threads qu'elle contient pourront être exécutés par n'importe quel cœur de ce nœud. L'organisation de ces files correspond exactement à la hiérarchie de l'architecture. A toute architecture hiérarchique correspond donc un arbre de files d'ordonnancement, dont la racine s'apparente au domaine d'ordonnancement le plus général (tous les cœurs de la machine), les feuilles caractérisant quant à elles les domaines d'ordonnancement les plus restreints, comme le processeur logique d'un cœur hyperthreadé par exemple. Cette modélisation de la topologie simplifie le problème de l'ordonnancement sur machines multicœurs en une projection dynamique d'un arbre de traitements sur un arbre de files d'ordonnancement.

3.2.2 Ordonnancement dynamique de parallélisme structuré

Cette projection est réalisée par le support exécutif à certains moments clés de l'exécution du programme. Elle peut être globale, auquel cas l'ensemble des traitements de l'application sont répartis, ou partielle, selon l'évènement qui la déclenche. Par exemple, l'entrée dans la première région parallèle du programme provoque une distribution sur la machine tout entière. A l'inverse, l'entrée dans des régions parallèles imbriquées donne lieu à des répartitions partielles, pour lesquelles seuls les traitements nouvellement créés seront répartis. Un cœur inactif peut aussi provoquer une nouvelle répartition, de façon automatique, en exécutant un algorithme de vol de travail. Cette fois, la répartition peut être partielle ou globale selon la physionomie du parallélisme, et le parcours de la topologie mis en œuvre par l'algorithme de vol. Une nouvelle répartition peut aussi être déclenchée par le programmeur. Ce peut être de façon explicite, directe-

ment dans le code du programme. Nous permettons ainsi au programmeur de jouer sur la fréquence d'intervention du support exécutif, ou encore d'indiquer les situations qui, selon lui, pourraient mettre en défaut la répartition actuelle. Notre support exécutif est aussi capable de déclencher de nouvelles répartitions de façon automatique. Les nouvelles informations que le programmeur lui transmet peut en effet soulever la nécessité de revoir la distribution des traitements ou des données. En particulier, nous offrons au programmeur la possibilité d'exprimer un besoin spécifique en ordonnancement pour chaque région parallèle.

Notre approche reconsidère ainsi le rôle du programmeur dans le processus de développement d'applications parallèles. Au lieu de le contraindre à investir du temps à la maitrise de mécanismes de bas niveau qui dépendent de l'architecture, comme le placement du parallélisme sur les unités de calcul ou encore la répartition physique des données, nous préférons lui permettre de transmettre son expertise à *ForestGOMP*. Les informations exprimées sont autant de guides sur lesquels s'appuient les politiques d'ordonnancement du support exécutif. Nous offrons de plus la possibilité au programmeur d'interroger à tout moment le support exécutif afin de consulter des statistiques d'exécution, comme les compteurs matériels représentant les taux de défauts de cache ou d'accès mémoire distants depuis le lancement du programme. L'analyse de ces statistiques peuvent en effet l'aider à estimer la pertinence des informations qu'il livre au support exécutif.

3.2.2.1 Différents besoins, différents ordonnanceurs

La politique d'ordonnancement ultime, s'adaptant à toutes les situations, n'existe pas. La répartition idéale des traitements parallèles peut changer selon les besoins de l'application. Certaines nécessitent par exemple un débit mémoire important alors que d'autres se comportent mieux si les threads qu'elles génèrent partagent de la mémoire cache. Ces besoins peuvent aussi évoluer selon l'architecture ciblée. L'affinité mémoire peut être négligée sur les architectures non-NUMA, par exemple. Même si une seule et même politique d'ordonnancement s'adaptant dynamiquement aux besoins évolutifs de toute application pouvait être implémentée, sa complexité la rendrait peu performante et très difficilement maintenable.

La plateforme BubbleSched, sur laquelle le support exécutif *ForestGOMP* s'appuie pour définir ses propres politiques, comme nous le verrons ultérieurement, facilite le développement d'ordonnanceurs à bulles. Un ordonnanceur à bulles peut être vu comme un ensemble d'algorithmes s'appliquant à tout parallélisme préalablement structuré par BubbleSched. Ces algorithmes sont appelés à certains moments clés de l'exécution du programme, comme par exemple l'entrée dans une nouvelle région parallèle, ou encore la détection de l'inactivité d'une unité de calcul. Le rôle d'un ordonnanceur est donc de répartir un certain nombre d'*entités*, terme générique pour définir une bulle ou un thread, sur les différents niveaux de l'architecture. Chaque ordonnanceur se concentre donc sur un besoin applicatif particulier, ce qui facilite grandement sa conception et sa maintenance.

Les ordonnanceurs à bulles sont instanciables sur différents éléments de l'architecture. On peut par exemple imaginer instancier un ordonnanceur par nœud NUMA de l'archi-

tecture qui pourront répartir, de manière concurrente, les traitements affectés à chaque nœud. Avec l'instanciation vient naturellement la composition d'ordonnanceurs. Il est ainsi possible d'imbriquer plusieurs ordonnanceurs s'occupant de la répartition des traitements à plusieurs niveaux de la topologie. Nous présenterons à ce propos dans la suite l'imbrication de deux ordonnanceurs à bulle, l'un dédié au respect des affinités mémoire opérant pour distribuer traitements et données sur les différents nœuds NUMA de l'architecture, l'autre chargé de répartir les traitements en fonction de leurs affinités à l'intérieur d'un nœud.

Le modèle d'exécution du langage OpenMP comporte un certain nombre de spécifités qu'il est possible d'exploiter du point de vue de l'ordonnanceur. En particulier, les threads créés à l'entrée d'une région parallèle exécutent le même code sur des jeux de données différents. Certaines familles d'algorithmes s'attachent à dimensionner ces données pour qu'elles puissent être stockées en cache. Sur une architecture multicœur, il revient alors au support exécutif de répartir les threads de façon compacte afin qu'ils puissent communiquer via l'un des différents niveaux de cache partagés. Nous avons développé l'ordonnanceur Cache de *ForestGOMP* dans ce but.

3.2.3 Cache: un ordonnanceur qui respecte les affinités entre threads

L'ordonnanceur Cache vise à respecter les relations d'affinité entre traitements, de telle sorte qu'ils puissent communiquer via les différents niveaux de cache partagés de l'architecture. Un tel ordonnancement s'adapte bien aux algorithmes de type *cache oblivious* et *cache conscious* qui veillent à dimensionner les données associées à chaque traitement parallèle pour exploiter au mieux le cache des processeurs. Les relations d'affinité sont exprimées par le regroupement des traitements dans un ou plusieurs niveaux de bulles. En d'autres termes, respecter ces relations passe par l'ordonnancement des groupes de threads de façon compacte sur la machine, afin qu'ils soient positionnés sur des cœurs partageant un ou plusieurs niveaux de mémoire cache.

Cache se compose de deux algorithmes de répartition, l'un appelé pour répartir des traitements nouvellement créés, l'autre appelé quand une unité de calcul devient inactive.

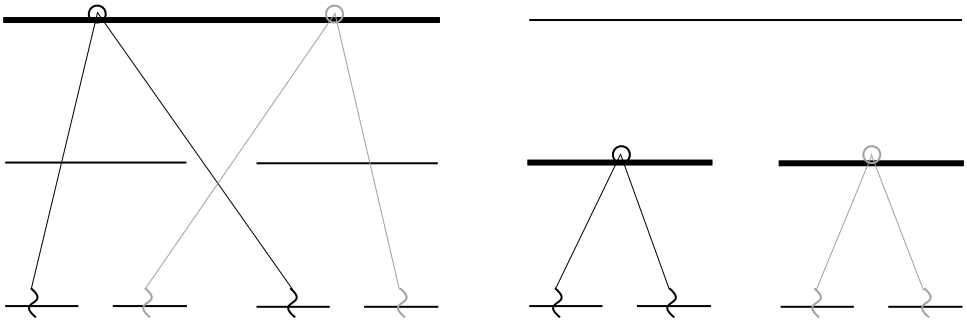
3.2.3.1 Répartir les groupes de threads de façon compacte

Quand il s'agit de répartir un ou plusieurs groupes de threads nouvellement créés, l'ordonnanceur Cache fait appel à un algorithme qui parcourt récursivement les listes de threads modélisant la topologie de la machine. Ce parcours débute par la liste la plus générale, qui recouvre tous les cœurs de l'architecture. L'algorithme est ensuite appelé récursivement sur les listes filles de la liste considérée, jusqu'à atteindre les feuilles de la topologie, représentant le plus souvent les cœurs ou, quand il y en a, les processeurs logiques (SMT) de l'architecture.

Le but de cet algorithme de répartition est de distribuer les bulles et threads de la liste considérée sur les différentes listes filles, en prenant soin de conserver la proximité des threads appartenant à une même bulle. Cet objectif est cependant souvent contrarié

par la nécessité d'occuper toutes les unités de calcul de la machine. Il arrive donc que l'ordonnanceur soit obligé d'extraire le contenu d'une bulle pour augmenter le nombre d'entités ordonnançables à répartir. On parle alors d'*éclatement* de bulle.

Dans les situations où l'éclatement d'une ou plusieurs bulles est inévitable, l'algorithme de répartition veille à retarder au maximum ce processus afin d'assurer que les threads de ces bulles soient exécutés *le plus près* possible les uns des autres. La figure 3.1 vient étayer cet argument. Elle présente deux façons d'ordonnancer une application générant deux équipes de deux threads sur un processeur à quatre cœurs, disposant de trois niveaux de mémoire cache: quatre caches privés de niveau 1, deux caches de niveau 2 partagés entre les cœurs 0 et 1 et les cœurs 2 et 3, et enfin un cache de niveau 3 partagé entre les quatre cœurs. Le parallélisme de cette exemple étant constitué de deux bulles pour occuper quatre cœurs, l'ordonnanceur va devoir procéder à l'éclatement de ces bulles pour répartir les quatre threads qu'elles contiennent. La figure 3.1(a) présente une répartition possible, obtenue en éclatant les bulles dès la liste d'ordonnancement la plus générale. Les threads d'une même équipe se retrouvent, dans cette situation, à communiquer via le cache de niveau 3 du processeur. La répartition présentée par la figure 3.1(b) est obtenue en retardant l'éclatement des bulles. Elles ne sont en effet éclatées qu'à partir des listes d'ordonnancement de niveau *cache L2*. De cette façon, les threads qu'elles contiennent communiquent via un cache plus rapide que la solution présentée précédemment. On peut ainsi reformuler l'objectif de l'ordonnanceur Cache: répartir les entités en retardant au maximum l'éclatement des bulles.



(a) Répartition obtenue en éclatant les bulles à partir du niveau *machine*.

(b) Répartition obtenue en retardant l'éclatement des bulles jusqu'au niveau *cache L2*.

FIG. 3.1 – Distribution de deux bulles de deux threads sur une machine à quatre cœurs, en fonction du niveau de la topologie à partir duquel l'ordonnanceur procède à l'éclatement des bulles.

Afin de mieux comprendre le fonctionnement de l'algorithme de répartition de Cache, nous nous appuierons sur la figure 3.2 qui présente les différentes étapes de l'ordonnancement de deux équipes de quatre threads sur l'architecture présentée dans le paragraphe précédent.

La figure 3.2(a) représente l'état initial dans lequel se trouvent les entités de l'application ordonnancée. L'algorithme de répartition est initialement appelé sur la liste d'ordonnancement la plus générale. Il commence par compter le nombre d'entités à répartir.

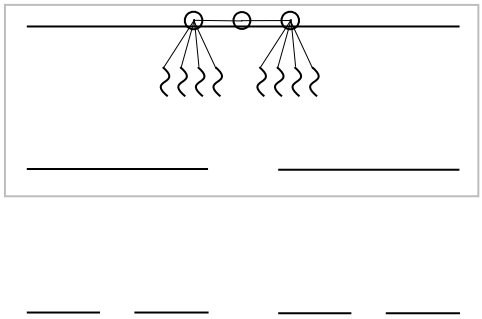
Ici, on dispose de deux bulles pour occuper quatre unités de calcul. Il sera donc nécessaire de les éclater pour augmenter le nombre d'entités ordonnançables. En analysant le contenu de ces bulles, on constate que chacune d'entre elles contient suffisamment de threads pour occuper toute une moitié de machine. L'ordonnanceur Cache décide donc de retarder l'éclatement des bulles, et de les répartir sur les listes sous-jacentes à l'aide d'un algorithme glouton, comme illustré par la figure 3.2(b).

L'algorithme de répartition est ensuite appelé récursivement sur chacune des listes du niveau *cache L2*. Cache ne trouve ici qu'une seule bulle pour occuper deux unités de calcul. Il n'a alors d'autre choix que d'éclater cette bulle sur le champ. Les threads que cette bulle contient sont extraits et placés sur la liste d'ordonnancement courante, comme représenté sur la figure 3.2(c). On rappelle alors l'algorithme de répartition sur cette même liste, puisque l'ensemble d'entités à répartir a changé. Cette fois-ci, Cache compte quatre entités sur cette liste et fait appel à un algorithme glouton pour les répartir sur les cœurs sous-jacents, comme illustré par la figure 3.2(d). La figure 3.2(e) présente la répartition finale calculée par l'ordonnanceur Cache sur cet exemple.

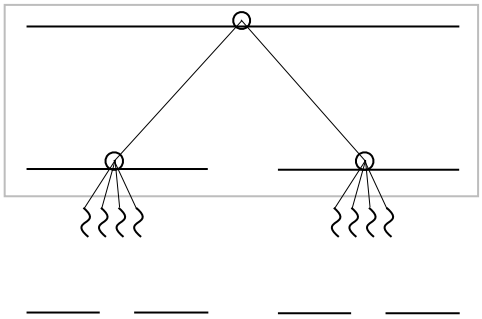
Par défaut, l'ordonnanceur Cache trie les bulles selon leur poids, c'est-à-dire le nombre de threads qu'elles contiennent, quand il s'agit de répartir les entités à l'aide de l'algorithme glouton. Nous offrons au programmeur la possibilité de définir la charge de travail associée à chaque région parallèle en s'appuyant sur une interface de programmation fournie par *ForestGOMP*. Ce critère aide l'ordonnanceur à différencier les entités de même poids. Cache consulte aussi les statistiques d'exécution des entités à répartir. Elles lui permettent d'attirer les threads vers des cœurs sur lesquels ils se sont déjà exécutés, espérant ainsi qu'ils réutilisent des données préalablement chargées dans le cache. Par défaut, ce critère d'affinité avec une unité de calcul prévaut sur les critères de charge de travail ou de poids d'une bulle, mais il est possible de redéfinir l'importance de chaque critère. Définir un nouveau critère pour guider l'ordonnancement passe par l'écriture d'une fonction spécifique de comparaison d'entités. L'injection de ces critères au sein de l'ordonnancement influe aussi sur le choix de la bulle à éclater en premier. De la même façon que Cache se souvient du dernier cœur sur lequel s'est exécuté une entité, il marque aussi les bulles ayant été éclatées au cours de répartitions précédentes. Le choix de la prochaine bulle à éclater se porte donc dans un premier temps sur celle disposant de la plus grande charge de travail. Si aucune information de charge n'a été fournie par le programmeur, Cache se tourne vers la bulle contenant le plus de threads, pour espérer limiter le nombre d'éclatements. Le fait qu'une bulle ait précédemment été éclatée entre en ligne de compte pour départager des bulles de même poids ou de même charge de travail.

3.2.3.2 Equilibrer la charge quand une unité de calcul devient inactive

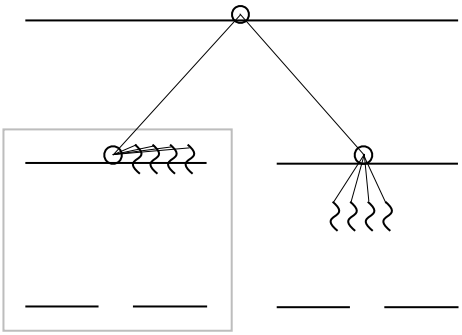
Au cours de l'exécution d'un programme OpenMP, il arrive parfois que certains threads terminent leur exécution de façon prématurée. C'est d'autant plus vrai pour les applications au parallélisme irrégulier, où la charge de travail associée à chaque thread varie d'un thread à l'autre. Ce déséquilibre dépend souvent du jeu de données passé à l'application. Il n'est dans ce cas pas possible de le prévoir a priori, c'est pourquoi nous choisissons d'armer le support exécutif pour être capable de rééquilibrer la charge dy-



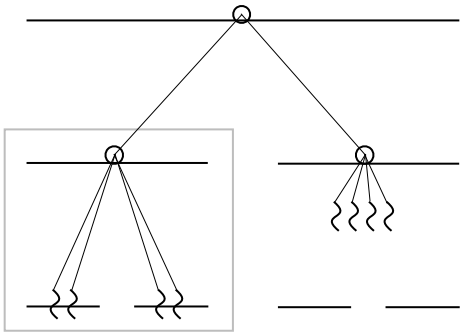
(a) Etat initial, après éclatement de la bulle racine.



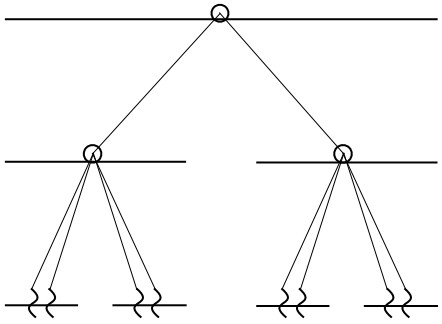
(b) Distribution gloutonne des bulles sur les listes de niveau cache L2.



(c) Eclatement de bulle par l'ordonnanceur.



(d) Distribution gloutonne des threads extraits sur les cœurs sous-jacents.



(e) Résultat de la répartition calculée par l'ordonnanceur Cache.

FIG. 3.2 – Ordonnancement de threads orchestré par l'ordonnanceur Cache sur une application OpenMP générant 2 équipes de 4 threads, exécutée sur une machine à quatre cœurs.

namiquement. L'ordonnanceur Cache dispose donc d'un algorithme de vol de travail appelé quand une unité de calcul devient inactive.

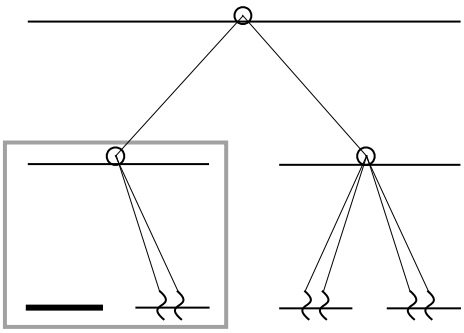
Il cherche ainsi à voler une ou plusieurs entités parmi les listes d'ordonnancement de la machine, en veillant à respecter au mieux les affinités entre threads. Pour ce faire, il s'appuie sur l'algorithme de répartition de Cache, qui comme on l'a vu précédemment cherche à occuper toutes les unités de calcul en respectant les affinités entre threads, pour redistribuer les entités sur une partie de la machine. Il revient à l'algorithme de vol d'identifier à partir de quelle liste d'ordonnancement il est nécessaire de redistribuer les entités sous-jacentes pour corriger le déséquilibre de charge.

La figure 3.3 illustre son comportement. L'inactivité du cœur 0 déclenche l'agorithme de vol, qui commence par rechercher à partir de quel niveau il faudra redistribuer les entités. Pour ce faire, il parcourt la topologie de proche en proche, élargissant petit à petit son domaine de recherche, représenté par le cadre gris sur la figure 3.3(a). Le parcours s'arrête lorsque la liste considérée couvre suffisamment d'entités pour occuper les unités de calcul sous-jacentes. La liste racine du domaine de recherche représenté sur la figure 3.3(a) couvre deux entités, pour deux cœurs à occuper. Cache remonte donc l'ensemble des entités contenues dans le domaine de recherche jusqu'à sa racine, comme illustré par la figure 3.3(b), avant d'appeler l'algorithme de répartition à partir de cette liste pour finalement obtenir le placement représenté par la figure 3.3(c).

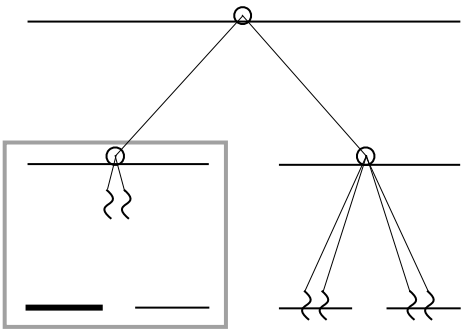
Supposons maintenant que les cœurs 0 et 1 deviennent tous deux inactifs quasiment au même instant. Dans ce cas, l'algorithme de vol est déclenché par l'un de ces cœurs, l'autre attendra que la répartition initiée par son voisin soit achevée. Ici, le domaine de recherche initial ne couvre pas suffisamment d'entités pour rééquilibrer la charge, comme le montre la figure 3.3(d). Il est donc étendu au domaine délimité par la liste père de la liste considérée jusqu'alors. Ce nouveau domaine couvre une bulle contenant quatre threads, que l'ordonnanceur redistribue pour obtenir la répartition présentée en figure 3.3(e).

Voler des entités depuis les listes d'ordonnancement voisines permet de mieux exploiter les différents niveaux de cache du processeur. Dans l'exemple illustré par la figure 3.3(c), le thread volé communique avec l'autre thread de sa bulle via un cache de niveau 2. Si nous avions choisi de voler un thread depuis les cœurs 2 ou 3, il aurait communiqué avec ses coéquipiers via le cache de niveau 3 du processeur, moins rapide.

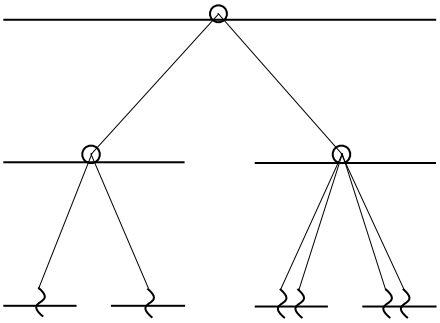
Même s'il obtient de très bons résultats sur les applications implémentant des algorithmes de types *cache-oblivious*, l'ordonnanceur Cache ne tient pas compte des données auxquelles les threads accèdent. Sur les architectures multicœurs à plusieurs nœuds NUMA, un placement efficace des données est primordial pour éviter les problèmes de performances liées à la contention sur le ou les bus mémoire. Nous avons pour cela développé pour *ForestGOMP* l'ordonnanceur Memory capable d'orchestrer de concert ordonnancement de threads et placement de leurs données de manière cohérente sur les architectures NUMA.



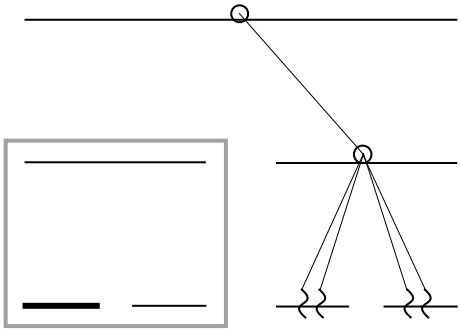
(a) Recherche d'une sous-partie de la topologie à partir de laquelle redistribuer les entités, après l'inactivité du cœur 0.



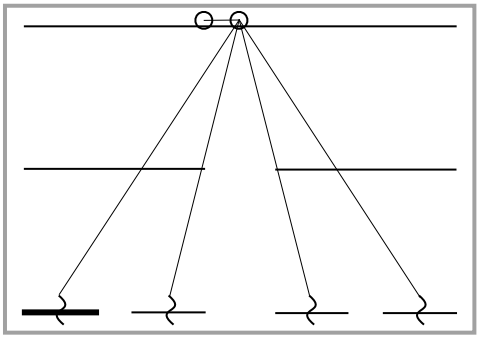
(b) Remontée des entités à répartir sur la liste racine du domaine de recherche.



(c) Répartition après l'appel à l'algorithme de vol de Cache.



(d) Vol déclenché par l'inactivité des cœurs 0 et 1: la recherche est ici infructueuse.



(e) Le domaine de recherche est étendu pour atteindre la bulle initialement exécutée par les cœurs 2 et 3.

FIG. 3.3 – Comportement de l'algorithme de vol de travail de l'ordonnanceur Cache selon différents scenari impliquant l'inactivité de certaines unités de calcul d'une machine à quatre cœurs.

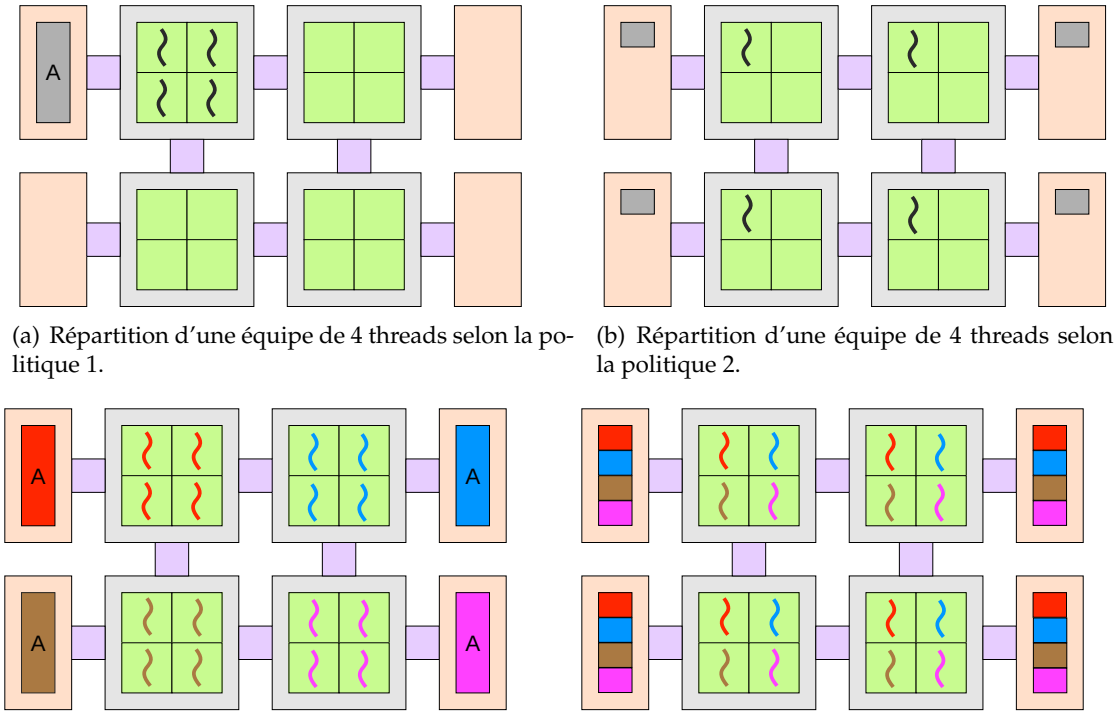
3.2.4 Memory: un ordonnanceur qui tient compte des affinités mémoire

Les processeurs multicœurs sont devenus, depuis les années 2000, les briques de base des architectures contemporaines, que ce soit dans un premier temps dans le domaine du calcul hautes performances, ou plus récemment jusque dans les machines du grand public. Leur interconnexion permet la construction de machines parallèles à grande échelle pour un coût toujours plus réduit. Les architectures qui en résultent proposent une organisation mémoire similaire à celle qu'on pouvait trouver dans les machines à accès mémoire non-uniforme de l'ère pré-multicœur. Une telle organisation doit être prise en compte par les applications et les supports exécutifs pour tirer réellement parti de ces architectures.

3.2.4.1 Débit mémoire et localité

Sur les architectures multicœurs à plusieurs bancs mémoire, le placement des traitements par rapport aux données auxquelles ils accèdent influe directement sur les performances des applications. C'est d'autant plus vrai que le programme exécuté accède intensivement à la mémoire. Les performances que ce type d'application obtient reste souvent étroitement liée à la bande passante mémoire disponible. Sur la machine présentée en figure 2.1, les quatre cœurs d'un même processeur se partagent la bande passante du lien HyperTransport pour accéder au banc mémoire de leur nœud NUMA. Ce phénomène de contention peut être mis en évidence à l'aide du benchmark mémoire STREAM [McC07]. Il vise à mesurer la bande passante soutenue d'une machine de calcul en effectuant des opérations arithmétiques simples sur des vecteurs d'entiers double précision. Il génère pour cela une équipe de plusieurs threads accédant en parallèle et de manière intensive à trois tableaux partagés. Dans un premier temps, nous cherchons à déterminer le meilleur placement pour une instance de STREAM à 4 threads, de façon à occuper un nœud NUMA de la machine cible. Nous expérimentons deux politiques de placement des threads et des données. La première politique, ou politique *locale*, place les quatre threads sur les cœurs d'un même processeur et le tableau sur le banc mémoire qui lui est attaché, comme présenté en figure 3.4(a). La seconde politique, ou politique *répartie*, distribue les threads et leurs données sur toute la machine. En pratique, on positionne un thread par nœud NUMA, et les pages mémoires qui correspondent au tableau partagé sont distribuées en tourniquet sur les différents bancs mémoire, comme représenté par la figure 3.4(b). Les performances obtenues par chacune des solutions sont présentées par le tableau 3.1. La seconde politique obtient les meilleures performances en terme de débit mémoire. La première politique oblige les threads à se partager la bande passante associée au lien HyperTransport du banc mémoire local. La répartition proposée par la seconde politique diminue grandement la probabilité qu'un lien HyperTransport soit traversé par quatre requêtes mémoire concurrentes. La bande passante offerte à chaque thread est donc potentiellement supérieure dans ce cas. Fort de ce constat, appliquons désormais ces deux politiques en prenant soin de charger complètement la machine cible. Pour ce faire, au lieu de ne créer qu'une équipe de quatre threads, nous allons lancer autant d'instances de STREAM que de nœuds NUMA de la machine (soit quatre, ici). Chaque instance travaillera sur son propre tableau, exclusivement accédé par les 4 threads qui la composent. Les figures 3.4(c) et 3.4(d) rappellent

le placement orchestré par chacune des politiques, considérant ce nouvel ensemble de threads. On observe ici que, contrairement aux résultats obtenus sur une machine non chargée, la politique locale obtient les meilleurs résultats. En effet, sur une machine non chargée, le surplus de bande passante mémoire offert aux threads de l'application compense des latences d'accès mémoire potentiellement supérieures du fait de l'emplacement de certaines pages mémoire sur des bancs mémoire distants. Dans le cas d'une machine chargée, la contention sur les différents liens HyperTransport est comparable entre les deux situations. En revanche, la politique locale assure que la majorité des accès mémoires effectués par les threads de l'application seront locaux.



(a) Répartition d'une équipe de 4 threads selon la politique 1.

(b) Répartition d'une équipe de 4 threads selon la politique 2.

(c) Répartition de 4 équipes de 4 threads selon la politique 1.

(d) Répartition de 4 équipes de 4 threads selon la politique 2.

FIG. 3.4 – Répartition des threads et des données selon deux politiques de placement.

Politique de placement	Politique locale (1)	Politique répartie (2)
4 threads sur le nœud 0	5151 Mo/s	5740 Mo/s
4 threads par nœud (16 au total)	4×3635 Mo/s	4×2257 Mo/s

TAB. 3.1 – Bande passante agrégée obtenue en s'appuyant sur différentes instances du benchmark STREAM sur une machine à 4 processeurs quad-core de type AMD Opteron en fonction de la charge de travail (1 ou 4 instances de STREAM à 4 threads) et de la politique de placement des threads et des données accédées.

3.2.4.2 Support logiciel pour la gestion des données sur les architectures NUMA

Des solutions logicielles existent pour aider le programmeur d'application à définir l'emplacement de ses données sur une architecture NUMA. Le système d'exploitation Linux implémente par exemple la politique d'allocation *first-touch*, mise en œuvre par défaut à l'allocation d'une zone mémoire de taille supérieure à 256 ko. Cette politique retarde l'allocation effective des pages mémoire jusqu'au moment où elles sont accédées pour la première fois. Elles sont alors allouées *au plus près* du premier thread qui y accède.

Pour être correctement exploitée, cette politique nécessite, durant la phase d'initialisation du programme, que chaque thread d'une région parallèle accède aux pages mémoire sur lesquelles il sera amené à travailler plus tard: on dit qu'ils *touchent* les pages mémoire. Un tel comportement peut grandement améliorer les performances des applications parallèles sur des architectures NUMA. En revanche, la politique *first-touch* suppose d'une part que le motif d'accès à la mémoire n'évolue pas au cours de l'exécution du programme, et d'autre part que les threads de l'application ne sont pas amenés à se déplacer. Une telle politique peine donc à s'adapter aux applications irrégulières, que ce soit en terme d'accès mémoire mais aussi en terme de parallélisme.

La politique *next-touch* a été conçue pour palier certains de ces manques. On peut la voir comme une évolution de la politique *first-touch* faisant intervenir des migrations mémoire. En pratique, le programmeur peut explicitement demander au système d'exploitation de marquer les pages d'une zone mémoire. Ces pages seront migrées au plus près du prochain thread qui y accèdera. Cette politique aide le programmeur à adapter le placement des données aux changements de comportement de son application. Cependant, elle est rarement implémentée dans les systèmes d'exploitation actuels et nécessite souvent une modification du noyau du système.

Plus généralement, on peut soulever un certain nombre de problèmes communs à l'utilisation de ces deux politiques. Tout d'abord, toutes les deux supposent que le thread qui touche une page mémoire le premier sera celui qui y accèdera le plus fréquemment ensuite. Si ce comportement s'adapte bien à certaines applications aux motifs d'accès mémoire réguliers, il est facilement mis en défaut dans le cas d'applications plus complexes, où il est difficile pour le programmeur de prédire l'ordre dans lequel les threads accèderont à des données partagées. De plus, aucune de ces deux politiques ne tient compte de la disponibilité des ressources sous-jacentes, en particulier de l'état des bancs mémoire de la machine. Il est tout à fait possible par exemple de demander au système d'exploitation la migration de pages mémoire vers un banc déjà plein, auquel cas le déplacement de ces pages peut ne pas avoir lieu.

3.2.4.3 Transmission des affinités mémoire au support exécutif

On vient de voir que, pour être réellement efficaces, les solutions logicielles pour la gestion mémoire sur les architectures NUMA ne peuvent être utilisées sans une connaissance précise de l'architecture sous-jacente. Pire, le programmeur devrait aussi pouvoir tenir compte de l'occupation des bancs mémoire de la machine, information rarement accessible au niveau applicatif. Aucune information ne lui remonte non plus quand un

thread de son application change de cœur après avoir été préempté par exemple, événement pouvant pourtant nécessiter une redistribution mémoire.

De son côté, le support exécutif a connaissance de l'agencement du parallélisme sur la machine *à tout instant*. Il est en effet responsable du placement des threads et perçoit de plus les spécifités architecturales de la machine cible. Il n'a en revanche aucune information sur les motifs d'accès à la mémoire au cours de l'exécution du programme. Cette information, très souvent connue par le programmeur, pourrait être transmise au support exécutif, permettant à ce dernier d'avoir une vue globale du placement des threads et des données. Dans un deuxième temps, le support exécutif doit être capable de migrer des zones mémoires, de la même façon qu'il est capable de déplacer des threads sur la machine.

Notre solution s'appuie sur la bibliothèque d'allocation mémoire MAMI, conçue spécifiquement pour tirer parti des architectures NUMA. Pour être exploitée par MAMI, une zone mémoire doit être *enregistrée* par la bibliothèque. Cet enregistrement se produit de façon implicite à l'appel d'une fonction d'allocation de MAMI, mais peut aussi être effectué à posteriori à l'aide d'un appel explicite. La bibliothèque implante les politiques d'allocation *first-touch* et *next-touch*, et permet la migration explicite de données, offrant ainsi au support exécutif le contrôle total du placement des zones mémoire d'une application. Elle s'intègre aussi parfaitement à la plateforme BubbleSched, puisqu'elle permet d'attacher des zones mémoires contiguës aux threads et bulles Marcel. Cet attachement, traité comme une statistique par BubbleSched, peut être perçu comme l'expression d'une relation d'affinité entre les threads d'une bulle et les données contenues dans la zone mémoire concernée. Les relations d'affinité mémoire sont ainsi consultables à tout instant de l'exécution du programme par le support exécutif, lui permettant de remettre en cause dynamiquement le placement des traitements et des données sur la machine cible. Il est aussi possible d'interroger MAMI quant à l'emplacement physique d'une zone mémoire dont elle a la charge.

La transmission des affinités mémoire au support exécutif passe par l'utilisation d'une interface de programmation fournie par ForestGOMP présentée sur le tableau 3.2. Les fonctionnalités proposées par cette interface peuvent être rangées en trois catégories qui correspondent à des niveaux d'implication différents de la part du programmeur:

1. **Le programmeur est capable d'exprimer avec précision quel thread accède à quelle donnée.** Il peut alors s'appuyer sur les fonctions d'attachement de zones mémoire à un thread ou une équipe OpenMP pour transmettre ces informations au support exécutif. Dans le cas d'un attachement à une équipe entière, la zone de données attachée est découpée en blocs par le support exécutif, afin d'attacher à chaque thread le bloc de donnée sur lequel il sera amené à travailler. Le programmeur peut régir ce découpage en définissant la taille des blocs et la façon de les attacher aux threads de l'équipe. Cette approche est celle qui nécessite le plus d'investissement du point de vue du programmeur, puisqu'il lui faut instrumenter fortement son application pour décrire de manière exhaustive les motifs d'accès à la mémoire et leur évolution. En revanche, cette approche lui permet de garder le contrôle sur l'expression des affinités et d'expérimenter avec précision l'impact que l'attachement d'une zone mémoire à une équipe de threads peut avoir sur la politique d'ordonnancement du support exécutif.

2. **Le programmeur sait identifier les changements de motifs d'accès à la mémoire dans son application, mais ces accès sont trop complexes pour être explicités avec précision.** Dans une telle situation, le programmeur peut s'appuyer sur la politique *next-touch* pour demander au système d'exploitation que les données soient déplacées entre chaque phase de son application. ForestGOMP va plus loin en proposant la politique *next-write* qui permet de marquer une zone mémoire comme étant à migrer la prochaine fois qu'un thread y accède en écriture. Cette politique est née de deux constats. Tout d'abord, nous avons observé, à distance équivalente, que les écritures coûtaient plus cher que les lectures. Le tableau 5.1 synthétise ces observations sur une machine à quatre nœuds NUMA Opteron. De plus, il n'est pas rare que le thread qui écrit une donnée soit celui qui y accède le plus souvent par la suite. Cette politique peut ainsi mieux se comporter que *next-touch*, qui peut être amenée à migrer des données auprès d'un thread qui n'y accède qu'occasionnellement en lecture.

3. **Le programmeur n'est pas capable d'identifier les changements de motifs d'accès à la mémoire.** Dans ce cas, il peut se reposer intégralement sur le support exécutif. L'interface de programmation de ForestGOMP propose en effet des fonctions pour organiser la surveillance d'une zone mémoire. Des compteurs matériels traduisant le nombre d'accès mémoire distants à partir de chaque processeur sont consultés puis remis à zéro à intervalles réguliers par le support exécutif. Quand ce nombre excède une valeur seuil, qui peut être définie par le programmeur, la zone mémoire surveillée est marquée comme étant à migrer. Le programmeur peut choisir la politique de migration, *next-touch* ou *next-write*, qui sera alors employée.

- void ***fgomp_malloc** (length);
  Alloue une zone mémoire et l'attache au thread courant.
- int **fgomp_set_current_thread_affinity** (buffer, length, shake_mode);
  Attache la zone mémoire passée en paramètre au thread appelant.
- int **fgomp_set_next_team_affinity** (buffer, chunk_length, shake_mode);
  Attache une zone mémoire à l'équipe OpenMP créée à l'entrée de la prochaine région parallèle.
- int **fgomp_attach_on_next_touch** (buffer, length);
  Attache une zone mémoire au prochain thread qui y accèdera.
- int **fgomp_migrate_on_next_touch** (buffer, length);
  Migre une zone mémoire au plus près du prochain thread qui y accèdera.
- int **fgomp_migrate_on_next_write** (buffer, length);
  Migre une zone mémoire au plus près du prochain thread qui y accèdera en écriture.
- int **fgomp_memory_watch** (buffer, length, threshold);
  Surveille une zone de données et la migre si les compteurs matériels consultés dépassent la valeur seuil passée en paramètre.

TAB. 3.2 – L'interface de ForestGOMP pour la gestion mémoire.

3.2.4.4 Ordonnancement conjoint de threads et de données

Le programmeur est désormais capable de transmettre les informations d'affinité entre les traitements et les données de son application. En s'appuyant sur la bibliothèque d'allocation MAMI, le support exécutif a donc la possibilité de consulter, à n'importe quel instant de l'exécution d'un programme, quel thread accède à quelles zones mémoires, et de déterminer l'emplacement physique des pages mémoire correspondantes. Etant en charge de la distribution des threads sur la topologie, il a également connaissance de leur placement tout au long de l'exécution.

L'ordonnanceur à bulles **Memory** s'appuie sur cette vue globale du placement des threads et des données pour proposer un ordonnancement respectant les affinités mémoire. Il fournit un algorithme de répartition portable dont le but est de calculer un placement s'inspirant de celui présenté en figure 3.4(c), quels que soient la machine cible, le parallélisme généré et le placement initial des données accédées. En pratique, cet algorithme vise à minimiser le nombre d'accès à un banc mémoire distant en faisant en sorte de regrouper les threads et les zones mémoire qu'ils accèdent sur le même nœud NUMA.

Ce regroupement peut s'effectuer soit en déplaçant les threads vers le nœud hébergeant leurs données, soit en migrant les pages mémoires au plus près des threads qui y accèdent. La migration de données est un procédé coûteux, comme l'a présenté le tableau 3.3 du chapitre précédent. La migration d'une seule page entre deux nœuds NUMA voisins dure déjà de l'ordre d'une centaine de microsecondes sur une machine à quatre nœuds NUMA Opteron. Modifier l'emplacement physique des pages mémoire engendre en effet des évictions dans les entrées du cache de traduction d'adresses des processeurs (TLB), ce qui peut expliquer ce surcoût conséquent. En comparaison, le déplacement d'un thread utilisateur BubbleSched entre deux cœurs de processeurs différents s'effectue bien plus rapidement, de l'ordre de quelques microsecondes sur cette même machine. Partant de ce constat, l'ordonnanceur **Memory** veille à migrer le moins de données possible quand il s'agit d'effectuer la répartition des threads et des données en fonction des affinités qui les lient.

L'algorithme de répartition de **Memory** comporte trois phases. Nous illustrons son comportement en ordonnançant une application synthétique créant deux équipes de 4 threads, chacune d'entre elles accédant à deux tableaux partagés, sur une machine à 2 nœuds NUMA de 4 cœurs. Ainsi, l'équipe 0 est attachée aux données A_0 et B_0 allouées respectivement sur les nœuds 0 et 1 de la machine. L'équipe 1 est quant à elle attachée aux données A_1 et B_1 allouées de façon similaire. La figure 3.5 représente cet état initial. L'application de chacune des phases de l'algorithme sur le parallélisme de cette application donne lieu aux répartitions successives représentées par la figure 3.6. Le code couleur employé ici permet de différencier les données distantes des données locales. Les données représentées en gris sont accédées par des threads exécutés par des cœurs du nœud NUMA local, celles représentées en noir sont accédées par des threads exécutés par des cœurs du nœud NUMA voisin.

Au cours de la première phase, chaque thread est attiré *de façon stricte* (sans tenir compte de l'équilibrage de charge sur la machine) vers le banc mémoire qui héberge la plus grande quantité de données qui lui sont attachées, comme illustré par la figure 3.6(a).

Cette première répartition peut provoquer un important déséquilibre de charge. On remarque en effet sur cet exemple que tous les threads sont attirés par le nœud 0, qui héberge les zones mémoire A_0 et A_1 de tailles supérieures aux zones mémoire B_0 et B_1, conduisant à l'inactivité des cœurs du nœud 1. C'est pourquoi, au cours de la deuxième phase de l'algorithme de répartition, la charge est rééquilibrée de telle sorte que tous les cœurs de la machine soient occupés. En pratique, l'ordonnanceur calcule pour chacune des bulles la quantité de données distantes, en fonction de l'emplacement des threads qui les composent. La ou les bulles déplacées entre les phases 1 et 2 de l'algorithme de répartition sont donc choisies de façon à minimiser la quantité de données accédées à distance, sur l'ensemble de l'application. Pour illustrer ce comportement, supposons les tailles de données suivantes: 48ko pour A_0, 64ko pour A_1, 16ko pour B_0 et 8ko pour B_1. Au terme de la distribution illustrée par la figure 3.6(a), l'application accède de manière distante à B_0 et B_1. La quantité de données accédées à distance est donc de 24ko. Afin de rééquilibrer la charge sur la machine, l'ordonnanceur doit choisir une bulle à déplacer vers le nœud 1. S'il choisit l'équipe 0, les données accédées à distance seront A_0 et B_1, pour une taille totale de 56ko. S'il choisit l'équipe 1, les données accédées à distance seront A_1 et B_0, pour une taille totale de 80ko. MEMORY choisit donc de déplacer l'équipe 0 vers le nœud 1 comme illustré par la figure 3.6(b). Au terme de la phase 2, la répartition des threads sur les différents nœuds NUMA de la machine est terminée. On assure que tous les cœurs sont occupés, tout en minimisant la quantité de données accédées à distance. La phase 3 parachève cette répartition en migrant, quand c'est possible, le reste des données distantes au plus proche du nouvel emplacement des threads qui y sont attachés. Sur notre exemple, l'équipe 1, placée sur le nœud 0 durant la phase 2, accède à la donnée B_1 allouée sur le nœud 1. Cette donnée va donc être migrée du nœud 1 vers le nœud 0 au cours de la troisième et dernière phase de l'algorithme de répartition de l'ordonnanceur MEMORY, comme indiqué sur la figure 3.6(c). Le résultat final de l'ordonnancement de notre programme synthétique par l'ordonnanceur MEMORY est présenté en figure 3.6(d).

Comme évoqué précédemment, le rôle de MEMORY se résume à distribuer conjointement les threads et leurs données sur les différents nœuds NUMA de la machine. La répartition des threads à l'intérieur d'un nœud doit tenir compte des différents niveaux de cache partagés entre les cœurs de ce nœud. Une façon simple d'y parvenir est de composer les ordonnanceurs MEMORY et CACHE, en les faisant intervenir à différents niveaux de l'architecture. En pratique, il est possible d'instancier des ordonnanceurs BubbleSched pour les affecter à des sous-arbres de la topologie. La répartition des threads et des données sur les différents nœuds NUMA peut donc être confiée à une instance d'ordonnanceur MEMORY qui appelera l'algorithme de répartition de l'ordonnanceur CACHE à l'intérieur de chacun des nœuds. La répartition correspondante est présentée sur la figure 3.6(e).

Certaines applications peuvent générer des threads aux durées de vie variables. On parle alors d'applications au parallélisme irrégulier. Le support exécutif est amené à revoir la distribution de threads dynamiquement pour éviter l'inactivité de certains cœurs de la machine. Comme l'ordonnanceur CACHE, MEMORY dispose d'un algorithme de vol de travail pour équilibrer la charge sur les différents nœuds NUMA, tout en migrant les données accédées si nécessaire. En pratique, l'inactivité d'un cœur déclenche l'algorithme de vol de travail de l'ordonnanceur CACHE associé au nœud NUMA cor-

respondant, comme illustré sur la figure 3.7(a) où après l'inactivité du cœur 3, un thread est volé sur la liste du cœur voisin. Il arrive parfois que l'algorithme de vol de CACHE ne trouve aucun thread à voler. Cette situation est illustrée par la figure 3.7(b) où les cœurs 2 et 3 sont devenus inactifs quasiment au même instant. L'ordonnanceur en charge de la répartition des threads dans le nœud 1 ne trouve alors rien à voler. Il contacte donc l'ordonnanceur MEMORY qui se charge de voler des threads depuis un nœud NUMA différent, en prenant soin de migrer les données accédées par ces threads.

Src / Dest.	Nœud 0	Nœud 1	Nœud 2	Nœud 3
Nœud 0	-	164 / 532 / 4251	165 / 532 / 4251	163 / 540 / 4253
Nœud 1	165 / 533 / 4382	-	165 / 533 / 4213	164 / 533 / 4221
Nœud 2	165 / 547 / 4282	165 / 537 / 4278	-	167 / 537 / 4280
Nœud 3	161 / 530 / 4325	165 / 546 / 4288	163 / 541 / 4279	-

TAB. 3.3 – Temps nécessaire, en microsecondes, pour effectuer la migration de 1 / 10 / 100 pages mémoires en fonction des nœuds source et destination, sur une machine à quatre nœuds NUMA, chacun d'entre eux comportant un processeur à quatre cœurs Opteron.

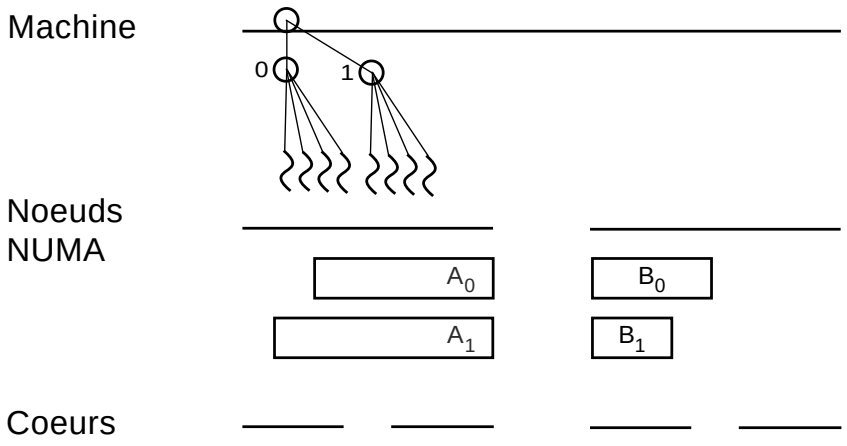
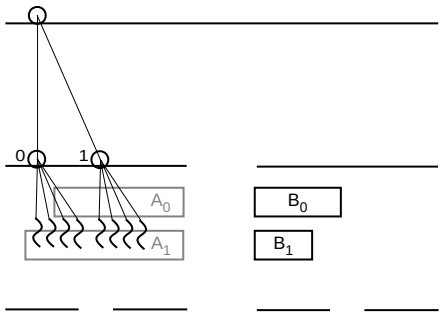


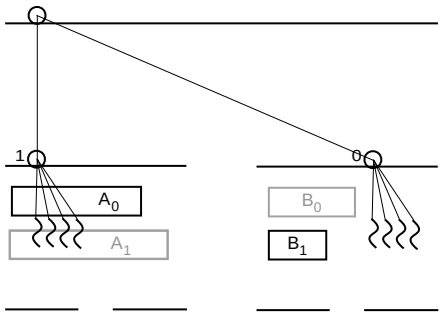
FIG. 3.5 – Répartition initiale des threads et des données du programme d'exemple sur lequel s'appuie la présentation de l'ordonnanceur **Memory**.

3.3 Composer, confiner, dimensionner

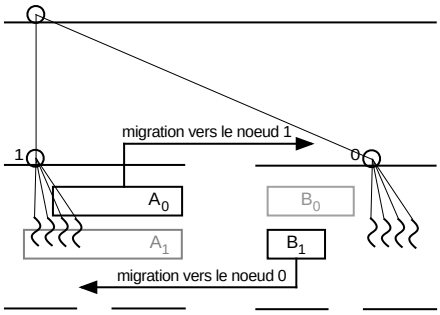
L'évolution des machines de calcul contemporaines passe par l'augmentation constante du nombre d'entités de calcul qui les composent. Cette réalité architecturale oblige les



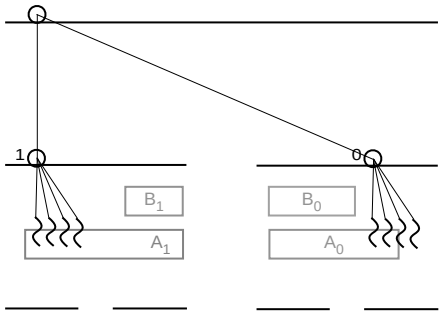
(a) Phase 1: Attirer chaque thread vers le banc mémoire contenant le plus de ses données.



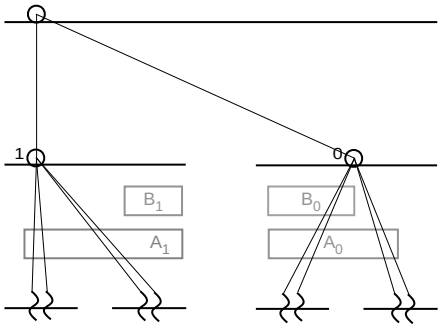
(b) Phase 2: Equilibrer la charge pour occuper tous les cœurs de la machine.



(c) Phase 3: Migrer le reste de données distantes vers l'emplacement final des threads.

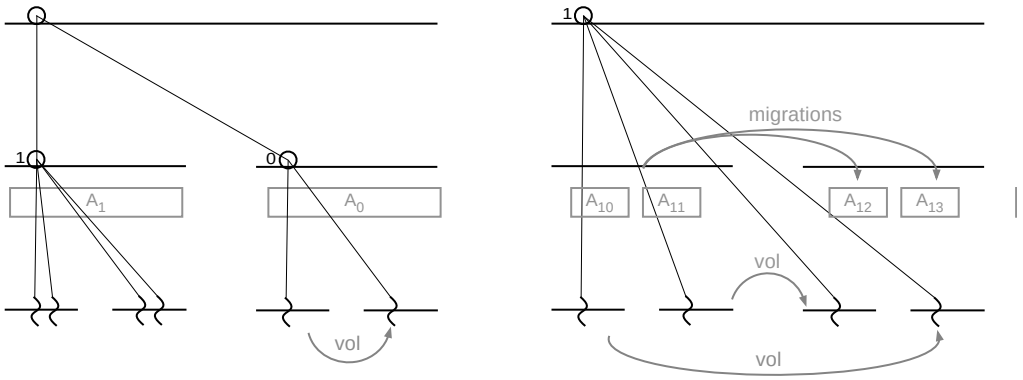


(d) Répartition effectuée par l'ordonnanceur **Memory**.



(e) L'ordonnanceur **Cache** peut ensuite être appelé à l'intérieur de chacun des nœuds.

FIG. 3.6 – Ordonnancement de threads et de données calculé par la combinaison des ordonnanceurs Memory et Cache sur une application OpenMP générant 2 équipes de 4 threads, exécutée sur une machine à deux nœuds à deux cœurs.



(a) Equilibrage de charge par CACHE après l'inactivité du cœur 3.

(b) Equilibrage de charge par MEMORY après l'inactivité des cœurs 2 et 3.

FIG. 3.7 – Equilibrage de charge par vol de travail orchestré par une combinaison des ordonnanceurs MEMORY et CACHE sur une application OpenMP générant 2 équipes de 4 threads, exécutée sur une machine à deux nœuds à deux cœurs.

programmeurs d'applications et de bibliothèques du domaine du calcul hautes performances à exprimer toujours plus de parallélisme. Bien qu'il soit parfois possible de raffiner le parallélisme existant d'une application parallèle, le programmeur est condamné à faire appel à de nouvelles sources de parallélisme, comme des bibliothèques parallèles, aux besoins en ordonnancement spécifiques. Cependant, la composition de telles bibliothèques implique l'exécution d'un parallélisme polymorphe difficile à ordonnancer. Les développeurs des bibliothèques de calcul scientifique *Intel MKL* [mkl] et *GotoBLAS* [GVDG08] en viennent même à fortement déconseiller l'utilisation de versions parallèles de leurs bibliothèques dans un code utilisateur déjà parallélisé. Ce chapitre, plus prospectif, présente les problèmes liés à la composition de codes parallèles et les évolutions des bibliothèques parallèles nécessaires à leur résolution, certaines d'entres elles étant déjà implantées dans ForestGOMP.

3.3.1 Composition de parallélisme

La composition de parallélisme passe par l'exécution concurrente de plusieurs codes parallèles aux besoins différents. Du point de vue d'OpenMP, notre approche s'appuie sur la composition d'ordonnanceurs à bulles pour adapter la répartition des traitements parallèles à la topologie sous-jacente et aux besoins applicatifs. Certaines politiques d'ordonnancement de ForestGOMP sont adaptées à des niveaux bien particuliers de la topologie. Par exemple, l'ordonnanceur *Memory* répartit threads et données de concert sur les différents nœuds d'une architecture NUMA, alors que l'ordonnanceur *Cache* distribue les threads en relation sur des cœuds partageant différents niveaux de cache, donc à l'intérieur d'un nœud NUMA. La composition de ces deux ordonnanceurs nous offre une politique d'ordonnancement qui s'adapte naturellement aux architectures NUMA, comme présenté sur la figure 3.8 où *Memory* distribue les équipes de threads et leurs données sur les différents nœuds NUMA et *Cache* ordonnance ces threads à l'intérieur

de chacun des nœuds.

D'un point de vue plus général, il existe des situations dans lesquelles la composition de code parallèles n'est pas aussi uniforme. Plusieurs régions de code parallèle, telles des briques de calcul, peuvent avoir des besoins différents nécessitant de confier leur répartition à des ordonnanceurs adaptés. Certains de ces besoins peuvent être contradictoires, comme par exemple grouper les threads d'une part pour maximiser l'utilisation du cache, et les répartir sur différents nœuds d'autre part pour obtenir la meilleure bande passante possible. De plus, la grande majorité des bibliothèques parallèles partent du principe que la totalité des ressources de calcul de la machine leur est dédiée, ce qui rend difficile leur intégration au sein d'applications elles-même parallèles. Une première solution consiste à sérialiser l'exécution de briques parallèles à la manière du *gang scheduling*. Elles sont alors ordonnancées tour à tour sur la machine. Cette solution ne permet pas en revanche l'exécution concurrente de briques de calcul parallèles et s'avère peu efficace si certaines d'entre elles peinent à exploiter la totalité de la machine. Une autre solution, que nous exposons dans la section suivante, consiste à confiner les différentes briques parallèles sur des sous-parties de la topologie, afin qu'elles puissent s'exécuter de manière concurrente en limitant les perturbations extérieures.

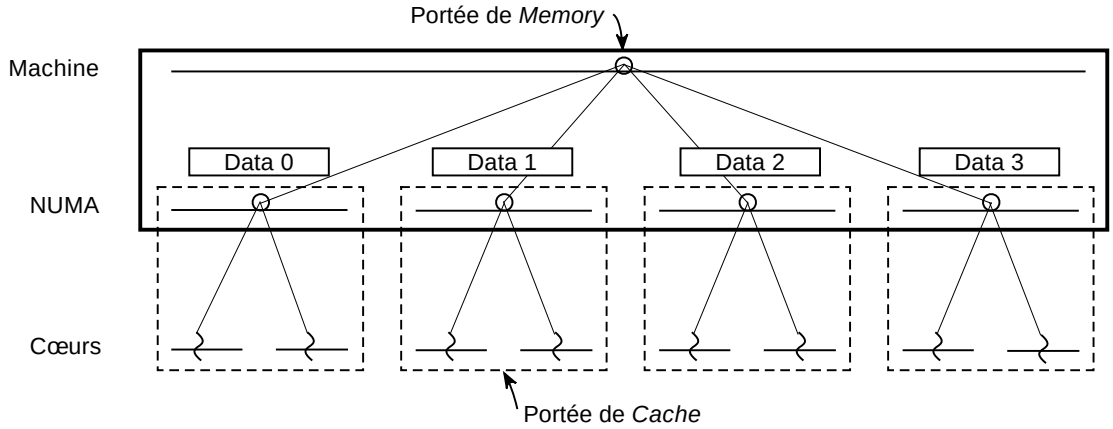


FIG. 3.8 – Composition d'un ordonnanceur *Memory*, en gras sur la figure, avec quatre ordonnanceurs *Cache*, représentés en pointillés, sur une machine à quatre nœuds NUMA de deux cœurs.

3.3.2 Confinement de parallélisme

Le confinement d'une région parallèle OpenMP est obtenu de façon directe par Forest-GOMP en lui associant une instance d'ordonnanceur qui définit une partition de la topologie sur laquelle les threads de cette région seront distribués. La figure 3.8 présente par exemple quatre instances d'ordonnanceur *Cache*. Ce partitionnement est généralisé quand il s'agit d'exécuter des applications différentes de manière concurrente, comme

présenté dans la section suivante.

3.3.2.1 Partitionnement de l'ensemble des ressources de calcul

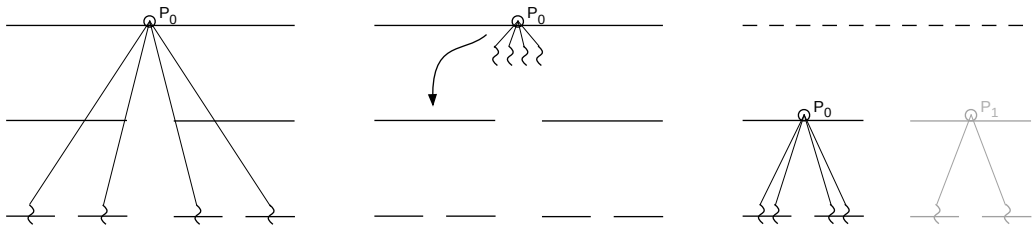
Plus généralement, l'exécution confinée de briques de calcul parallèles passe par un partitionnement de la machine qui se doit, pour être efficace:

1. d'être établi en connaissance des besoins en ressources de calcul des différents codes à exécuter en concurrence
2. d'être suffisamment dynamique pour pouvoir réaffecter des ressources de calcul à une partition, dans le cas par exemple où une région parallèle se termine et libère ses ressources.

Le cas particulier où plusieurs applications OpenMP sont exécutées de manière concurrente se traite naturellement avec *ForestGOMP*. En effet, la capacité de confinement de ce support exécutif lui permet de réserver une partie de la machine pour l'exécution d'une application OpenMP particulière. On a donc autant d'instances de *ForestGOMP* que de processus exécutés, le seul élément les différenciant étant la vision qu'ils ont de la topologie. La tâche de partitionnement de la machine est laissée à un *superviseur*, développé pour l'occasion au sein de *ForestGOMP*, qui arbitre les accès aux unités de calcul. Il est capable d'attribuer des cœurs, sous forme de *processeurs virtuels*, aux différents processus. Chaque instance de *ForestGOMP* considère les cœurs qui leur sont attribués comme une machine à part entière et construisent leur propre représentation de cette topologie. Le superviseur maintient une liste des processus en cours d'exécution et des ressources qu'ils occupent. Il est de plus capable d'enlever ou d'ajouter des cœurs à un processus en cours d'exécution.

La figure 3.9 illustre ce comportement: un processus noté *P0* est en train de s'exécuter sur une machine à quatre cœurs. L'intégralité des ressources de calcul de la machine est à sa disposition, comme le montre la figure 3.9(a). Un processus *P1* s'enregistre auprès du superviseur pour être exécuté. En conséquence, le superviseur alerte le processus *P0* que deux cœurs vont lui être enlevés. Ce changement se traduit simplement par une modification de la racine de la topologie vue par le processus *P0*. Ainsi, les entités de *P0* sont rassemblées sur la liste d'ordonnancement racine de sa topologie courante, puis déplacées vers la liste racine de sa nouvelle topologie, comme indiqué sur la figure 3.9(b). Une fois le parallélisme du processus *P0* confiné, le processus *P1* peut s'exécuter dans l'espace qui lui a été réservé par le superviseur, comme représenté sur la figure 3.9(c).

Le problème de composition se pose aussi en cas d'imbrication de codes parallèles de natures différentes. Prenons l'exemple d'une application OpenMP, ordonnancée par ForestGOMP, exécutant des routines de la bibliothèque d'algère linéaire *GotoBLAS* de manière concurrente. La documentation de cette bibliothèque indique qu'il est impératif d'utiliser la version séquentielle des appels BLAS dans une telle situation. En effet, l'implémentation parallèle de GotoBLAS s'appuie sur un pool de threads, appelé *serveur BLAS*, distribués sur les différents cœurs de la machine à l'initialisation de la bibliothèque. Ces threads sont réveillés pour traiter une fraction du travail associé à un appel BLAS puis rendormis quand ils ont terminé. L'appel à plusieurs routines BLAS depuis une région parallèle OpenMP pose ainsi deux problèmes majeurs:



(a) Initialement, le processus *P0* s'exécute sur toute la machine.

(b) Le superviseur signale au processus *P0* que deux cœurs vont lui être enlevés.

(c) Le processus *P1* débute son exécution une fois le parallélisme du processus *P0* confiné sur deux cœurs.

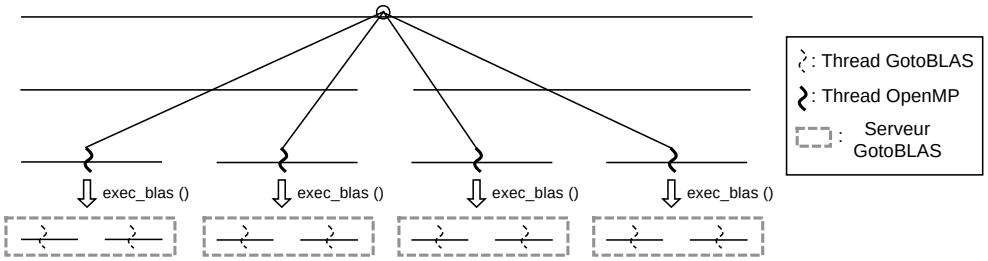
FIG. 3.9 – Cohabitation de deux processus OpenMP via le support exécutif *ForestGOMP* sur une architecture à quatre cœurs.

1. La conception même de la version parallèle de GotoBLAS empêche l'exécution de plusieurs opérations BLAS de manière concurrente: les opérations sont exécutées les unes à la suite des autres par tous les threads du pool.
2. Les threads ForestGOMP et GotoBLAS se battent pour obtenir l'accès aux ressources de calcul, se perturbant les uns les autres.

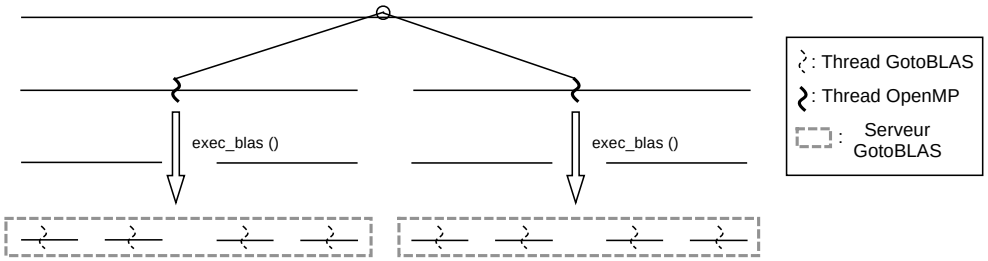
Afin de résoudre ces problèmes, nous avons tout d'abord modifié la bibliothèque GotoBLAS de façon à rendre possible le recrutement de sous-ensembles distincts de threads du serveur BLAS pour exécuter des opérations BLAS différentes. On confine ainsi l'exécution d'une routine GotoBLAS à un ensemble contigu de threads, et donc de cœurs. Nous faisons ensuite cohabiter les threads ForestGOMP et GotoBLAS selon le mode de fonctionnement des ordonnanceurs coopératifs: un thread ForestGOMP qui exécute une routine BLAS recrute un ensemble de threads GotoBLAS et leur passe la main, qu'il ne récupèra qu'une fois l'opération BLAS terminée. Le partitionnement des ressources de calcul est ici implicitement effectué par ForestGOMP, comme illustré par la figure 3.10. Le découpage de la machine en serveurs GotoBLAS est défini par le nombre de threads OpenMP, comme illustré sur les figures 3.10(a) et 3.10(b). Pour des raisons de performances, le placement d'un thread ForestGOMP sur la topologie définit l'ensemble des threads GotoBLAS qu'il pourra recruter pour exécuter une opération BLAS.

3.3.2.2 Gestion de la surcharge en environnement confiné

Les solutions que nous venons de voir traitent des briques de calcul au parallélisme unidimensionnel. L'imbrication de parallélisme à l'intérieur d'une brique pose de nouveaux problèmes au support exécutif. En particulier, ce parallélisme apparait alors que des threads sont déjà positionnés et en cours d'exécution sur les différents cœurs de la machine. Le parallélisme imbriqué constitue aussi un moyen de surcharger les processeurs, une approche qui, lorsqu'elle est mal supportée par le support exécutif, peut dégrader les performances de l'application. Ce problème n'est pas nouveau pour ForestGOMP: le programmeur OpenMP dispose en effet de constructions lui permettant d'affiner le parallélisme de son application. Il lui est ainsi possible de paralléliser un traitement déjà affecté à un thread OpenMP, donnant naissance à une région parallèle



(a) Appel à une opération BLAS depuis une région parallèle à 4 threads. La taille d'un serveur BLAS est automatiquement fixée à 2.

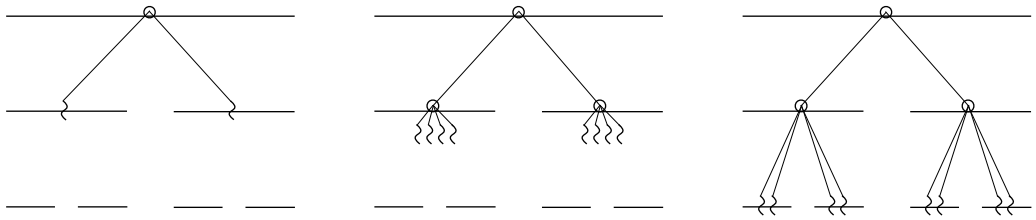


(b) Appel à une opération BLAS depuis une région parallèle à 2 threads. La taille d'un serveur BLAS est automatiquement fixée à 4.

FIG. 3.10 – Exécution de routines parallèles GotoBLAS à l'intérieur d'une région parallèle OpenMP ordonnancée par ForestGOMP sur une machine à 8 cœurs.

dite *imbriquée*, ou *nested* en anglais. Il semble d'autant plus naturel d'imbriquer des régions parallèles que les architectures multicœurs imbriquent leurs composants à n'en plus finir: des processeurs logiques dans des cœurs, des cœurs dans des processeurs, des processeurs dans des nœuds NUMA. La gestion efficace du parallélisme imbriqué requiert un dynamisme important du support exécutif pour évaluer s'il est nécessaire de remettre en question les décisions d'ordonnancement à chaque nouvelle région parallèle.

De notre point de vue, la gestion efficace du parallélisme imbriqué passe par l'exécution confinée des threads nouvellement créés sur un ensemble de cœurs partageant des ressources. L'orchestration d'un tel parallélisme ne peut donc se faire qu'en harmonie avec la topologie de la machine cible. Le confinement de régions parallèles imbriquées est obtenu de façon directe par ForestGOMP, qui garantit que les threads créés sont placés sur la même liste d'ordonnancement que leur créateur. La figure 3.11 illustre la gestion du parallélisme imbriqué par ForestGOMP, sur une application synthétique à deux niveaux de parallélisme exécutée sur une machine à deux processeurs bi-cœurs. La figure 3.11(a) présente la répartition calculée par l'ordonnanceur CACHE de la région parallèle externe de l'application. On remarque que cette région parallèle a généré deux threads qui sont placés sur chacun des processeurs de la machine. L'entrée dans une région parallèle interne provoque la création de nouveaux threads. Ces threads sont créés sur la liste d'ordonnancement de leur créateur, comme illustré par la figure 3.11(b). Ainsi, l'équipe 0 ne peut être exécutée que par l'un des deux cœurs du processeur 0. On confine ainsi de façon naturelle les régions parallèles imbriquées. L'algorithme de répartition de l'ordonnanceur courant est ensuite appelé à partir de chacune des listes hébergeant les équipes nouvellement créées, pour finalement obtenir le placement représenté par la figure 3.11(c).



(a) Placement des threads après ordonnancement de la région parallèle externe.

(b) Création locale des threads issus des régions parallèles imbriquées.

(c) Ordonnancement local des équipes nouvellement créées.

FIG. 3.11 – Création et ordonnancement par ForestGOMP des équipes de threads issues de régions parallèles imbriquées.

Outre l'exécution confinée de briques parallèles différentes se pose le problème de quelle granularité affecter à chaque région ou application parallèle. En effet, l'une des raisons de la popularité d'OpenMP est sa portabilité: le support exécutif détermine le nombre d'unités de calcul sur la machine pour créer automatiquement le nombre de threads correspondant. Une application au parallélisme récursif ne peut cependant pas se contenter de générer autant de threads que de cœurs à chaque région parallèle. En effet, la création d'un trop grand nombre de threads dégradera à coup sûr les perfor-

mances de cette application. Il est donc nécessaire, pour que le parallélisme imbriqué soit exploité de façon efficace, de définir le nombre de threads affecté à chaque région parallèle. La section suivante présente les outils qui permettent au programmeur, en s'appuyant sur *ForestGOMP*, de définir une granularité adaptée au comportement de l'application.

3.3.3 Dimensionnement de régions parallèles OpenMP

Estimer les besoins en ressource de calcul d'une application, ou même d'une région parallèle, peut s'avérer très difficile. Le comportement par défaut d'OpenMP est de créer autant de threads par régions parallèles que de cœurs de l'architecture. Par exemple, une application OpenMP compilée sur une ordinateur portable à deux cœurs peut ainsi être exécutée sur une machine de calcul à plus grande échelle et occuper toutes ses unités de calcul. En revanche, rien n'indique que les performances de cette application seront proportionnelles au nombre de cœurs de la machine sur laquelle elle est exécutée. En pratique, il est très difficile d'atteindre les performances crête d'une machine de calcul à grande échelle. Certaines régions parallèles ne profitent pas de la totalité des cœurs qu'on leur attribue, parfois pour des raisons algorithmiques, mais aussi parce qu'elles peinent à s'adapter aux spécifités des architectures hiérarchiques. La définition du nombre de threads par région parallèle OpenMP reste à la charge du programmeur, et peut devenir très complexe en fonction du niveau d'imbrication du parallélisme et de l'architecture cible.

Nous avons exploré la possibilité d'estimer, au niveau du support exécutif, la *scalabilité* des régions parallèles OpenMP pour faire remonter cette information au programmeur. Nos expériences préliminaires s'appuient sur un échantillonnage dynamique des performances d'une boucle OpenMP. Le support exécutif calcule le nombre d'itérations traitées par seconde en fonction du nombre de threads associés à la boucle parallèle échantillonnée. L'analyse de ces performances indique parfois qu'il n'est pas nécessaire d'occuper toute la machine pour exécuter une ou plusieurs régions parallèles. Les cœurs inoccupés sont ainsi laissés à disposition du superviseur, lequel peut décider de les utiliser pour exécuter une autre application parallèle ou de les affecter à une application en cours d'exécution. Ce procédé ne présente en revanche qu'une première approximation des besoins en puissance de calcul d'une région parallèle. Le relevé de performances effectué peut en effet être mis en défaut quand plusieurs applications sont exécutées en même temps. L'état de la machine peut alors changer, en particulier l'occupation des bus mémoire, nécessitant la reconstruction des différents profils de performances calculés jusqu'alors.

3.4 Synthèse

L'exploitation efficace des architectures hiérarchiques passe, de notre point de vue, par une coopération étroite entre le programmeur d'applications parallèles et le support exécutif. Les travaux de cette thèse s'appuient sur cette vision pour proposer une solution portable et performante pour exécuter des programmes OpenMP sur les architec-

tures multicœurs. La contribution de cette thèse s'articule en trois axes majeurs:

1. Conserver de façon pérenne et transmettre la structure des applications OpenMP au support exécutif ForestGOMP: les régions parallèles OpenMP génèrent automatiquement des bulles BubbleSched.
2. Ordonnancer ce parallélisme structuré à l'aide des ordonnanceurs *Cache* et *Memory*, développés respectivement pour maximiser l'utilisation des caches partagés d'un processeur multicœur et distribuer conjointement les threads et leurs données sur les architectures à accès mémoire non-uniformes.
3. Composer ce parallélisme avec d'autres bibliothèques et applications parallèles, en mettant en avant les caractéristiques de ForestGOMP qui facilitent la composition: confinement, repliement de parallélisme sur une partie de machine, gestion efficace du parallélisme imbriqué et de la surcharge.

Le chapitre suivant développe certains points techniques concernant l'implémentation du support exécutif ForestGOMP et des ordonnanceurs *Cache* et *Memory*.

Chapitre 4

Éléments d'implémentation

Sommaire

4.1 Au niveau de ForestGOMP 61
4.1.1 Une implémentation de barrière pour les architectures hiérarchiques 61
4.1.2 Un effort de compatibilité 65
4.2 Au niveau de BubbleSched 66
4.2.1 Des ordonnanceurs instanciables et composables 66
4.2.2 Un module d'aide au développement d'algorithmes gloutons . 68
4.3 Au niveau de GotoBLAS 70
4.4 Synthèse et discussion 72

Ce chapitre développe quelques points techniques sur ForestGOMP, BubbleSched et l'adaptation de la bibliothèque d'algèbre linéaire GotoBLAS.

4.1 Au niveau de ForestGOMP

Cette section traite de l'implémentation des barrières de ForestGOMP ainsi que des efforts qu'il a fallu fournir pour garantir la compatibilité binaire de notre support exécutif avec celui du compilateur GCC.

4.1.1 Une implémentation de barrière pour les architectures hiérarchiques

La barrière est l'outil de synchronisation privilégié d'OpenMP. Elle peut être appelée soit de façon explicite par le programmeur via la directive ad'hoc, soit de façon implicite à la terminaison des régions parallèles comme à la sortie de la plupart des structures de distribution de travail (`sections`, `single` et `for`). Son impact sur les performances est donc très important pour les micro-benchmarks et est souvent perceptible sur les benchmarks. Nous avons été amené à développer un nouvel algorithme de barrière afin d'assurer à ForestGOMP la capacité de supporter correctement la surchage pour un surcoût raisonnable. Pour cela nous avons travaillé principalement deux aspects: éviter

les pièges de l'attente active (livelocks) sans pour autant payer le surcoût important induit par l'endormissement des threads d'une part, et limiter la contention au niveau des structures de données de la barrière d'autre part.

Nous évitons les pièges de l'attente active en faisant coopérer l'ordonnanceur et les threads usagers d'une barrière en appliquant le principe suivant: tout thread arrêté par une barrière doit passer rapidement la main à l'ordonnanceur, et celui-ci doit régulièrement ordonnancer des threads qui n'attendent pas sur une barrière. Nous avons pour cela développé deux implémentations:

1. Une première implémentation assure un ordonnancement équitable (sans famine) des threads. Pour cela, nous plaçons les threads sur les listes d'ordonnancement de plus bas niveau et utilisons la primitive `yield()` pour appeler l'ordonnanceur qui passera la main, en tourniquet, aux threads prêts de la liste. La situation progresse fatalement car tous les threads de la machine ne peuvent simultanément être arrêtés par des barrières.
2. Dans une deuxième implémentation, plus générale mais passant moins bie à l'échelle, nous passons explicitement la main à un thread qui n'a pas franchi la barrière. Pour cela, nous mémorisons dans la structure de donnée associée à la barrière l'identité des threads participants et de ceux étant arrivés. Tout thread arrêté par la barrière sera alors en mesure de proposer à l'ordonnanceur une liste des threads à ordonnancer en priorité. La situation progresse car la structure arborescente des équipes OpenMP empêche la création de cycle: si l'ordonnanceur passe la main à un thread arrêté par une autre barrière alors cette barrière est celle d'une équipe imbriquée strictement plus profondément que la première.

Il s'agit ensuite de contrôler la contention pour éviter les dégradations de performances. Pour se synchroniser, les threads échangent des informations contenues dans des lignes de caches. Pour limiter la contention, il est donc important de maitriser le nombre de lignes qui entrent et sortent d'un cache. Idéalement, le nombre de lignes échangées par les caches devrait correspondre au nombre de caches mis en jeu. Ce n'est en revanche pas toujours le cas. Prenons l'exemple d'une barrière implémentée par un entier incrémenté de façon atomique par chaque thread signalant son arrivée. Le nombre de lignes échangées par les caches sera au pire de l'ordre du nombre de threads (un défaut de cache par écriture). Pour minimiser le nombre d'échanges de ligne de cache tout en favorisant le parallélisme, il est nécessaire d'utiliser une barrière hiérarchique dont les structures de données utiles à la synchronisation reposent sur plusieurs lignes de cache utilisées le plus localement possible. Cependant, outre le fait d'avoir un nombre varié de threads par cœur, une autre difficulté vient du fait qu'un thread peut parfois passer d'un cœur à un autre. Nous avons donc conçu des barrières où les threads utilisent un index pour signaler leur arrivée, comme par exemple le numéro du cœur où le thread a été enregistré. Cet index peut être modifié lorsqu'une différence jugée significative est observée, par exemple lorsqu'un thread est exécuté par autre cœur d'un processeur différent.

La première barrière que nous avons développé est indépendante de l'architecture de la machine. Elle est basée sur un arbre binaire de booléens, comme présenté en figure 4.1. Un thread signale sa présence dans une ligne de cache désignée par son index puis s'informe dans la même ligne de cache de l'arrivée de son compagnon. Le thread re-

monte ainsi l'arbre tant qu'il constate la présence de ses compagnons successifs. Notre implémentation permet de faire varier le nombre de synchronisations sur une même ligne de cache afin de pouvoir jouer sur la contention. L'originalité de cette approche est d'utiliser le faux partage comme outil de synchronisation, n'impliquant le plus souvent qu'une seule et unique opération atomique pour désigner le thread qui va assurer la redistributions des index.

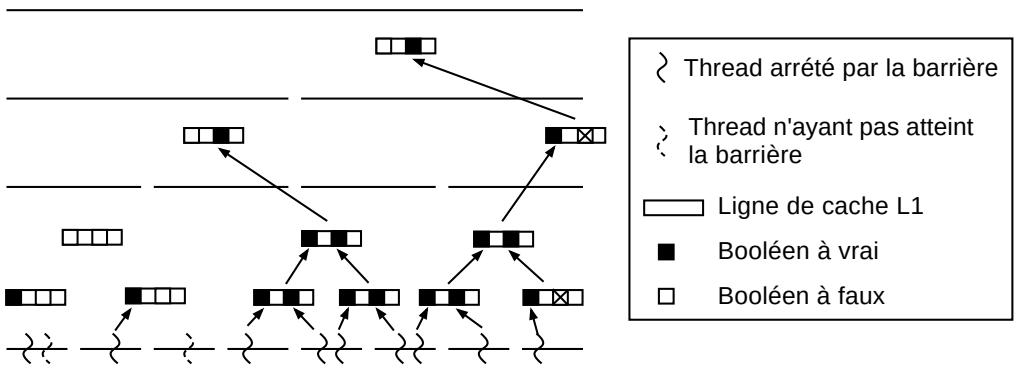


FIG. 4.1 – Barrière ForestGOMP à arbre binaire de booléens, sur une machine à 8 cœurs.

Dans une deuxième approche, nous avons créé une barrière reposant sur la structure matérielle, la première approche s'étant avérée inadaptée aux forts facteurs NUMA (supérieurs à 3). Pour adapter la barrière à l'architecture, nous avons utilisé une ligne de cache par cœur pour comptabiliser/synchroniser les threads que chacun exécute, puis une ligne de cache par processeur pour comptabiliser les cœurs participant à la barrière, puis par nœud NUMA et une dernière pour la machine tout entière, comme représenté sur la figure 4.2. Nos tests sur différentes architectures récentes ont montré qu'il n'était pas nécessaire de consacrer une ligne de cache par cœur mais plutôt par niveau de cache le plus élevé (niveau L3 en général).

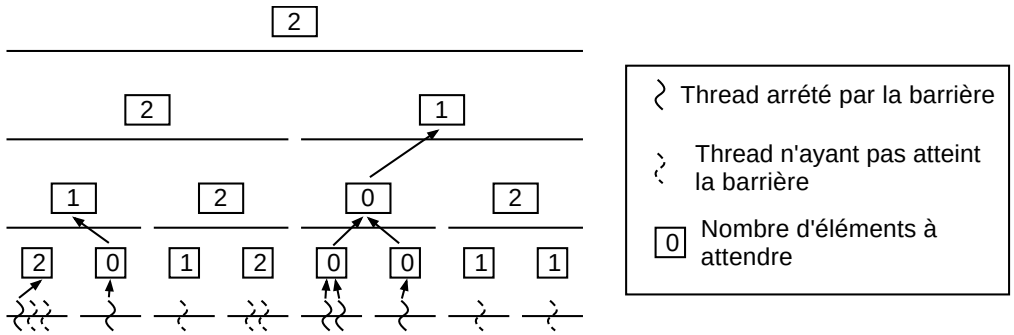


FIG. 4.2 – Barrière ForestGOMP à hiérarchie de compteurs, sur une machine à 8 cœurs.

Afin d'évaluer les performances de ces deux implémentations, nous exécutons le microbenchmark OpenMP *EPCC*, qui sera présenté en détails dans le chapitre suivant, sur une machine à 96 cœurs Intel Xeon X7460 cadencés à 2,66Ghz organisée en 4 nœuds

NUMA de 24 cœurs. A titre de comparaison, nous testons aussi la barrière du support exécutif d'Intel ICC et relevons ses performances en fonction du placement des threads sur la machine, spécifié à l'aide de la variable d'environnement `KMP_AFFINITY`, disponible uniquement sur architecture Intel. Nous évaluons pour cela deux placements différents : *compact*, pour un placement des threads sur un ensemble contigu de cœurs partant de 0 ; *scatter* pour une distribution éclatée des threads sur la machine. Les performances obtenues sont représentées sur la figure 4.3.

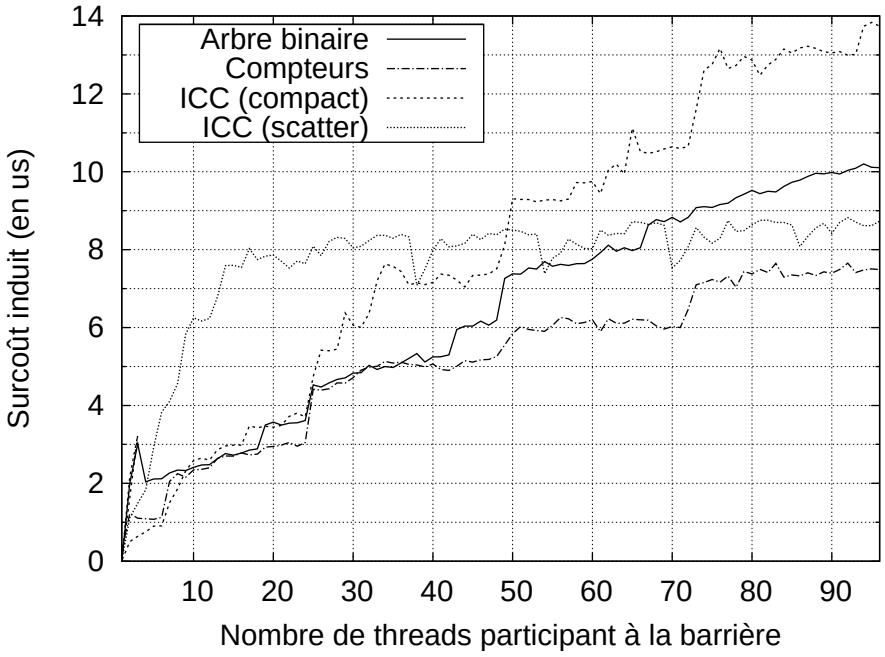


FIG. 4.3 – Surcoût induits par plusieurs algorithmes de barrières exécutés sur une machine à 96 cœurs.

Le placement *scatter* permet à la barrière d'ICC de limiter le surcoût de sa barrière à 9 μs pour 96 threads. Le surcoût pour un faible nombre de threads est cependant supérieur au placement *compact*, puisque les threads doivent ici communiquer entre différents nœuds NUMA. On observe plusieurs sauts sur la courbe du surcoût associé au placement *compact* qui retranscrivent le relief de la machine: entre 6 et 7, on sort d'un processeur hexacœur, entre 24 et 25 on sort d'un nœud NUMA, à partir de 49 on fait intervenir 3 nœuds NUMA, et 4 à partir de 73. Les performances de la barrière ForestGOMP basée sur un arbre binaire de booléens sont comparables à celles d'ICC, le placement des threads opéré par l'ordonnanceur Cache étant ici identique à la version *compact*. On constate enfin que la barrière ForestGOMP à hiérarchie de compteur, tenant compte de l'architecture, se comporte le mieux ici, avec un surcoût ne dépassant pas les 7,7 μs pour 96 threads.

Plus généralement, les performances des barrières sans surcharge que nous avons développées sont comparables à celles d'ICC et bien meilleures que celles de libGOMP3, implémentées à l'aide d'un compteur unique, sur les machines dépassant le nombre de 24 cœurs. Elles se comportent en revanche bien mieux que les barrières d'ICC en contexte surchargé. Le tableau 4.1 montre les surcoût induits par des barrières au nombre de threads excédant le nombre de cœurs de la machine. On teste ici 16, 32 puis 64 threads sur une machine à 4 processeurs quadri-cœurs Opteron. La barrière de ForestGOMP testée ici est celle basée sur la hiérarchie de compteurs. Les performances qu'elle obtient confirment que cet algorithme de barrière a été pensé pour limiter le surcoût lié à la surcharge des processeurs.

Support exécutif	16 threads	32 threads	64 threads
libGOMP	2,91	187,9 (×64)	371,1 (×128)
Intel ICC	4,93	17,98 (×3,64)	63,29 (×12,84)
ForestGOMP	3,31	3,87 (×1,17)	5,15 (×1,56)

TAB. 4.1 – Impact de la surcharge sur le surcoût de la barrière calculé par le benchmark EPCC en fonction du nombre de threads sur une machine à 16 cœurs Opteron.

4.1.2 Un effort de compatibilité

Un des objectifs de ForestGOMP est d'assurer la compatibilité binaire avec libGOMP, le support d'exécution de GNU OpenMP (GCC). Cette compatibilité permet de tester les algorithmes d'ordonnancement fournis par ForestGOMP et BubbleSched de manière transparente sur des applications OpenMP, sans qu'il soit nécessaire de les recompiler.

Certaines des structures de données exposées par libGOMP, telles que `omp_lock_t`, englobent elles-mêmes des structures de données fournies par le système d'exploitation, en particulier le noyau et la bibliothèque de threads POSIX (`libpthread`). ForestGOMP est donc dans l'obligation d'être également compatible au niveau binaire avec ces API.

Par exemple, sur GNU/Linux, le type `omp_lock_t` de libGOMP repose sur les *futexes*[Dre04], un mécanisme de synchronisation bas-niveau spécifique à ce système d'exploitation. Le type `omp_lock_t` se résume à un entier: libGOMP utilise l'appel système `futex` pour enregistrer ou réveiller les threads en attente de verrou.

Les futexes font l'hypothèse que chaque thread de l'application est fourni par le noyau, comme c'est le cas avec la libpthread GNU (NPTL). Or, ForestGOMP utilise des threads utilisateurs Marcel exécutés par des threads noyaux. Une application Marcel ou ForestGOMP est donc dans l'impossibilité d'utiliser les futexes.

ForestGOMP fournit donc un type `omp_lock_t` dont les contraintes d'alignement et de taille sont les mêmes que celles de ce type dans libGOMP, mais les opérations sur `omp_lock_t` sont implémentées en termes de verrous Marcel et non de futexes. Malheureusement, sur architecture 64 bits, la taille de `omp_lock_t` est de 32 bits alors que celle d'un pointeur est de 64 bits. ForestGOMP voit donc les `omp_lock_t`

comme des entiers dont la valeur est interprétée comme un indice dans un tableau de `marcel_lock_t`.

Par ailleurs, la bibliothèque standard C de GNU/Linux est intimement liée avec la bibliothèque de threads POSIX qu'elle fournit. Par exemple, les verrous internes à la Glibc, les *low-level locks*, sont désactivés tant que la libpthread ne l'a pas informée de la présence de plusieurs threads. Bien entendu, ce mécanisme est interne à la Glibc et à la NPTL. La couche de compatibilité binaire libpthread de Marcel s'arrange pour utiliser ce mécanisme. Les fonctions d'entrée/sortie de la Glibc utilisent les *low-level locks* qui, là encore, font l'hypothèse du modèle de threads de la NPTL, différent du modèle N:M de Marcel.

De même la Glibc dépend de libpthread, via des symboles « faibles », ainsi que de librt, avec laquelle chaque programme OpenMP compilé par GCC est automatiquement lié. De plus, une application compilée sous GNU/Linux est susceptible d'utiliser le mécanisme de stockage par thread, communément appelé *thread-local storage* ou *TLS*, mis en œuvre conjointement par GCC, la Glibc et la NPTL.

Toutes ces raisons font que ForestGOMP doit obligatoirement être utilisé conjointement avec la couche de compatibilité binaire libpthread de Marcel, mais aussi avec la bibliothèque PukABI. PukABI fournit des remplacements des fonctions d'entrée/sortie de la Glibc, protégés par des verrous Marcel. Ces deux bibliothèques doivent être préchargées, à l'aide de la variable d'environnement *LD_PRELOAD*, pour que leurs symboles prévalent sur ceux de la Glibc et de la NPTL. La commande `run-forest` se charge de préparer cet environnement d'exécution.

4.2 Au niveau de BubbleSched

Cette section développe certains points techniques sur l'instanciation et la composabilité des ordonnanceurs à bulles avant de mettre en lumière un module conçu pour aider le développement d'algorithmes d'ordonnancement gloutons.

4.2.1 Des ordonnanceurs instanciables et composables

À l'origine, l'ordonnanceur à bulle était choisi à la compilation de BubbleSched. Afin de lever cette contrainte, nous avons proposé une interface permettant de définir des instances d'ordonnanceurs. Pour des raisons de simplicité d'utilisation, la définition d'un ordonnanceur par défaut a été conservée, mais celui-ci peut être changé au lancement d'une application. Un ordonnanceur est désormais décrit par une structure, représentée en figure 4.4, contenant un ensemble de pointeurs de fonctions et une liste d'ordonnancement définissant sa portée.

Instancier un ordonnanceur à bulle revient donc à allouer l'espace nécessaire pour stocker cette structure, brancher les fonctions pour définir son comportement et enfin spécifier sa portée en affectant le champ `root_level`. Au démarrage de BubbleSched, une instance de chaque type d'ordonnanceur est créée, avec une portée étendue à la machine entière. Choisir son ordonnanceur à l'exécution est maintenant direct: il suffit

```
struct ma_bubble_sched_struct {
  ma_bubble_sched_init init;
  ma_bubble_sched_start start;
  ma_bubble_sched_exit exit;

  /* Répartition d'une bulle */
  ma_bubble_sched_submit submit;

  /* Vol de travail */
  ma_bubble_sched_vp_is_idle vp_is_idle;

  /* Portée de l'ordonnanceur */
  struct marcel_topo_level *root_level;
};
```

FIG. 4.4 – L'objet *ordonnanceur* de la bibliothèque BubbleSched.

de positionner la variable globale indiquant l'ordonnanceur courant à l'instance d'ordonnanceur choisie.

Afin de faciliter l'instanciation d'ordonnanceurs existants, nous avons de plus introduit la notion de *classe* d'ordonnanceurs, qui définit un état par défaut pour l'objet instancié. Par exemple, l'instantiation d'un objet de classe *ordonnanceur Cache* produit une structure dans laquelle sont déjà branchés les algorithmes de répartition et de vol de travail de Cache. Il reste à l'utilisateur à définir la portée de l'ordonnanceur instancié.

Nous avons enfin étendu l'interface de programmation de BubbleSched de manière à pouvoir faire référence à l'instance d'ordonnanceur concernée lorsqu'on demande la répartition d'une nouvelle bulle, comme l'illustre la figure 4.5.

```
int
marcel_bubble_sched_instantiate (marcel_bubble_sched_class_t *c,
                                 marcel_bubble_sched_t *s);

marcel_bubble_sched_t *
marcel_bubble_get_current_sched (void);

marcel_bubble_sched_t *
marcel_bubble_set_sched (marcel_bubble_sched_t *new_s);

int
marcel_bubble_submit_to_sched (marcel_bubble_sched_t *s,
                               marcel_bubble_t *b);
```

FIG. 4.5 – Interface de gestion des instances d'ordonnanceurs à bulles de BubbleSched.

4.2.2 Un module d'aide au développement d'algorithmes gloutons

Les algorithmes de répartition des ordonnanceurs Cache et Memory peuvent être qualifiés de gloutons: ils distribuent les entités d'une liste d'ordonnancement sur les listes sous-jacentes en prenant soin de toutes les occuper. Sur toutes les répartitions possibles qui valident cet objectif, seule une poignée d'entre elles satisfait les critères de l'ordonnanceur Cache. Au lieu de toutes les calculer puis d'éliminer celles qui ne respectent pas les affinités entre threads, l'algorithme de répartition utilise des heuristiques calculant une répartition satisfaisante pour un critère donné. Il est possible que cette répartition ne remplisse cependant pas certains objectifs, auquel cas l'ordonnanceur peut juger nécessaire de la modifier, voire d'en calculer une nouvelle.

Prenons l'exemple d'une application créant 8 threads sur une machine à deux nœuds NUMA contenant 4 cœurs chacun. Si les données attachées à ces 8 threads ont été allouées sur le nœud 0, l'algorithme de répartition de l'ordonnanceur Memory va calculer une première répartition, positionnant les 8 threads sur les cœurs correspondant. Cette répartition contente le critère de respect des affinités mémoire, au détriment de l'équilibrage de charge. C'est pourquoi, dans un deuxième temps, Memory nuance cette répartition pour occuper tous les processeurs de la machine.

On comprend ici que l'application de plusieurs heuristiques successives pour calculer une répartition amène un grand nombre de manipulations sur les entités exécutées sur la machine et sur les listes d'ordonnancement. En particulier, déplacer un thread d'une liste vers une autre oblige à verrouiller le contenu de la bulle dont il fait partie, ainsi que les listes source et destination de son déplacement. Ainsi, plus la politique d'ordonnancement est complexe, plus elle nécessite d'étapes intermédiaires qui impliquent de multiples déplacements de bulles et de threads avant d'atteindre la répartition souhaitée.

Afin de réduire les surcoûts induits par les manipulations d'entités sur les listes d'ordonnancement, et plus généralement pour aider l'expert à concevoir des algorithmes gloutons voire de modifier l'existant, nous avons développé le module `ma_distribution` permettant le calcul des répartitions intermédiaires sur des structures de données annexes qu'il pourra manipuler sans verrou. On associe à chaque liste d'ordonnancement une structure particulière présentée en figure 4.6, qui contient la liste des entités que l'on souhaite placer dessus, leur nombre ainsi que leur charge totale.

A chaque étape de l'algorithme de répartition, on construit un tableau de structures représentant les listes d'ordonnancement sur lesquelles on doit répartir les entités. Les heuristiques calculant des répartitions intermédiaires sont ainsi appliquées sur ce tableau, au lieu d'être directement appliquées à la topologie elle-même. Les manipulations d'entités se font à l'aide de l'interface de programmation présentée en figure 4.7. Une fois la répartition finale calculée, on l'applique en appelant la fonction `ma_apply_distribution` qui parcourt le tableau de structures et déplacement physiquement chaque entité qu'elles contiennent vers les listes d'ordonnancement qu'elles représentent.

Ce comportement permet de ne déplacer les entités qu'une fois par niveau de la topologie. Le module `ma_distribution` est aussi utilisé pour parcourir les listes d'ordon-

```
struct ma_distribution {
  /* Liste d'entités attirées
     par le niveau topologique _level_ */
  ma_entity_ptr_t *entities;

  /* Nombre d'éléments de la liste _entities_ */
  unsigned int nb_entities;

  /* Charge totale de la liste d'entités _entities_ */
  unsigned int total_load;

  /* Niveau topologique vers lequel
     les entités _entities_ sont attirées */
  struct marcel_topo_level *level;
};
```

FIG. 4.6 – Structure de données décrivant un niveau de la topologie et les entités qu'il attire.

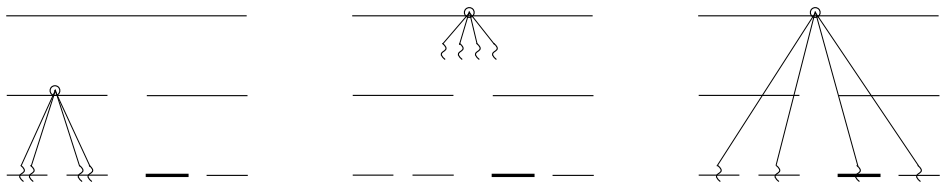
```
/* Functions d'ajout/suppression d'une entité */
void ma_distribution_add_head (marcel_entity_t *e,
                               ma_distribution_t *d);
void ma_distribution_add_tail (marcel_entity_t *e,
                               ma_distribution_t *d);
marcel_entity_t * ma_distribution_remove_head (ma_distribution_t *d);
marcel_entity_t * ma_distribution_remove_tail (ma_distribution_t *d);

/* Accession aux niveaux topologiques les plus/moins chargés */
int ma_distribution_least_loaded_index (const ma_distribution_t *d,
                                        unsigned int arity);
int ma_distribution_most_loaded_index (const ma_distribution_t *d,
                                       unsigned int arity);

/* Application de la distribution calculée */
void ma_apply_distribution (ma_distribution_t *d, unsigned int arity);
```

FIG. 4.7 – Interface de programmation du module `ma_distribution`.

nancement lors d'un vol de travail. Il participe à la stabilité de la répartition en ne déplaçant que les threads effectivement volés, comme indiqué sur la figure 4.8. Pour équilibrer la charge sur la machine, l'algorithme de vol de l'ordonnanceur Cache remonte l'équipe de thread exécutée par les cœurs 0 et 1 sur la figure 4.8(a), afin de pouvoir appliquer l'algorithme de répartition depuis la liste d'ordonnancement générale. Cette étape intermédiaire, représentée en figure 4.8(b), étant en fait calculée sur les structures annexes du module `ma_distribution`, on finit par ne déplacer effectivement que les deux threads volés, laissant s'exécuter sans interruption les threads en cours d'exécution sur les cœurs 0 et 1.



(a) Les cœurs 2 et 3 sont inactifs, et font appel à l'algorithme de vol de travail de l'ordonnanceur Cache.

(b) Cache remonte les entités sur la liste d'ordonnancement générale pour rappeler l'algorithme de répartition: cette phase est exécutée sur les structures annexes du module `ma_distribution`.

(c) La répartition finale est calculée, seuls deux threads sont physiquement déplacés par la fonction `ma_apply_distribution`.

FIG. 4.8 – Utilisation du module `ma_distribution` dans le cadre du vol de travail de l'ordonnanceur Cache.

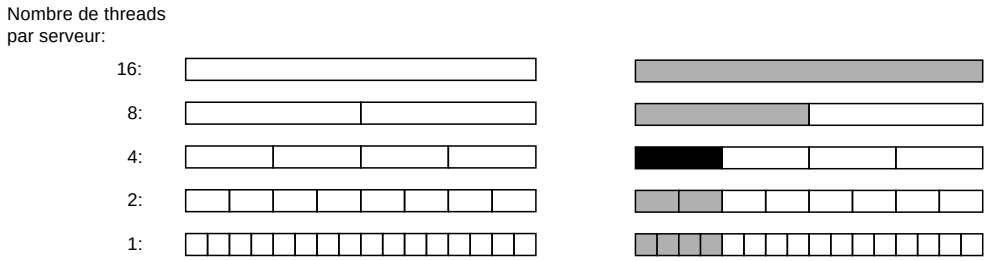
4.3 Au niveau de GotoBLAS

La composition de ForestGOMP et de la bibliothèque d'algèbre linéaire GotoBLAS s'inspire du mode de fonctionnement des ordonnanceurs coopératifs: lorsqu'un programme OpenMP souhaite exécuter une opération BLAS, un thread ForestGOMP passe la main à un pool de threads GotoBLAS qui exécute le calcul jusqu'à son terme avant de rendre la main au thread l'ayant invoqué. L'initialisation de la bibliothèque GotoBLAS engendre la création d'un `pthread` par cœur de la machine et alloue une zone de mémoire destinée à stocker les vecteurs et matrices sur lesquels travailler, initialisée selon la politique *first-touch* à l'aide de l'opération BLAS `touch_memory`. Pour des raisons de performances, nous avons conservé cette initialisation, et étendu la bibliothèque de manière à pouvoir créer des instances logiques de serveurs BLAS. Nous sommes ainsi capables de recruter un sous-groupe contigu de threads GotoBLAS pour exécuter une opération BLAS, indépendamment des autres threads. Le thread ForestGOMP stocke ainsi une référence vers l'instance qu'il vient de créer. La distribution du travail sur les threads d'une instance de serveur BLAS est explicitée par l'algorithme 1. L'espace d'itérations de la boucle en ligne 2 s'étendait initialement de 0 au nombre de cœurs de la machine. Nous l'avons modifié pour qu'il ne concerne que les threads appartenant à l'instance de serveur BLAS concernée.

Algorithme 1 Distribution de travail sur une instance de serveur BLAS.

1: $blas_server \leftarrow$ fgomp_get_blas_server()
2: **for** $i = blas_server \rightarrow first_thread$ to $blas_server \rightarrow last_thread$ **do**
3: assign_work_to_thread (i)
4: **end for**

GotoBLAS est maintenant capable d'exécuter des opérations différentes de manière concurrente en recrutant autant de serveurs BLAS que nécessaire. Les ressources recrutables sont cependant limitées à un thread par cœur de la machine, et deux opérations BLAS à 16 threads ne pourront s'exécuter en parallèle sur une machine à 16 cœurs. Pour résoudre ces conflits, nous avons dû développer un mécanisme qui arbitre la réservation de threads BLAS sur la machine. La machine cible est ainsi partitionnée en sous-groupes de puissances de 2 de cœurs. Prenons l'exemple d'une machine à 4 processeurs quadri-cœurs, nommée *Kwak*. Les 16 cœurs qui la composent permettent de recruter un serveur de taille 16, ou deux serveurs de taille 8, ou 4 serveurs de taille 4, et ainsi de suite. On remarque que le recrutement d'un serveur de taille 4 limite le nombre de serveurs de taille 8 à 1, et empêche le recrutement d'un serveur de taille 16. Pour simplifier cette gestion, nous maintenons l'état des réservations de serveurs BLAS sur la machine sous forme arborescente, comme l'illustre la figure 4.9.



(a) Serveurs BLAS disponibles sur la machine Kwak.

(b) Réservation d'un serveur BLAS de 4 threads (en noir). Les serveurs BLAS en gris sont marqués comme non-réservables.

FIG. 4.9 – Système de réservation de serveurs GotoBLAS sur la machine Kwak.

L'algorithme de réservation peut être résumé en quelques points :

1. Parcourir la ligne correspondant à la taille de serveur désirée afin de trouver une case libre ;
2. Si une case libre a été trouvée, la marquer, sinon s'endormir sur un verrou associé à la taille du serveur qu'on souhaite réserver ;
3. Parcourir et marquer les fils de la case réservée ;
4. Parcourir et marquer les ancêtres de la case réservée.

La libération d'un serveur BLAS provoque le parcours inverse pour démarquer les cases correspondantes. Avant de démarquer la case père, on vérifie que notre voisin n'est pas marqué. Libérer un serveur de taille N implique le réveil d'un thread en attente de réservation d'un serveur de même taille.

4.4 Synthèse et discussion

Les développements effectués au cours de cette thèse ont fait l'objet de plusieurs distributions logicielles disponibles à la fois sur le site internet de ForestGOMP [for] ainsi que sur la forge INRIA [gfo]. Bien que ce chapitre ne présente que quelques points techniques, il laisse entrevoir trois catégories dans lequelles peuvent être rangés les développements de cette thèse:

1. **La bibliothèque ForestGOMP**: les développements au sein de cette bibliothèque ont permis d'optimiser les performances brutes de notre support exécutif. C'est ici qu'on retrouve les différents algorithmes de barrière exposés précédemment, mais aussi la gestion de pools de threads pour optimiser la création d'équipes OpenMP, ou encore l'implémentation des fonctions permettant au programmeur de transmettre des informations au support exécutif.
2. **Les ordonnanceurs à bulles**: une grande partie du temps de développement a aussi été consacré au développement des ordonnanceurs *Cache* et *Memory*, cette fois au sein de la plateforme BubbleSched, mais aussi à l'intégration des bibliothèques de bas niveau, comme la bibliothèque de gestion mémoire MaMI, aux stratégies d'ordonnancement.
3. **Les preuves de concept**: on retrouve dans cette catégorie la modification de la bibliothèque parallèle GotoBLAS pour la rendre composable avec les applications OpenMP ordonnancées par ForestGOMP, mais aussi les développements d'applications synthétiques comme la série de benchmarks STREAM qui sera présentée dans le chapitre suivant.

Les développements au sein de ForestGOMP sont nécessaires puisqu'ils garantissent la compétitivité de notre support exécutif vis-à-vis de la concurrence en limitant les surcoûts liés à la gestion des bulles notamment, et plus généralement à l'utilisation d'OpenMP. En plus des optimisations apportées à la gestion des différents mot-clés du langage OpenMP, nous nous sommes attachés à améliorer la facilité d'installation et d'utilisation de notre support exécutif, de sorte qu'aujourd'hui un simple lanceur, baptisé *run-forest*, suffit à exécuter une application OpenMP compilée avec GCC sur le support exécutif ForestGOMP.

Des développements futurs permettront d'améliorer encore les performances de ForestGOMP. En particulier, l'algorithme de franchissement de barrière pourrait facilement être adapté à l'application d'une réduction OpenMP. Il reste aussi à optimiser le partage de travail invoqué par les directives d'ordonnancement `schedule`. Du côté de BubbleSched, il reste difficile pour un néophyte de développer une stratégie d'ordonnancement ad'hoc. Cette difficulté, étroitement liée à la manipulation incessante de listes et de verrous en langage C, pourrait être levée en s'appuyant sur un langage de haut niveau pour proposer une interface de programmation accessible au plus grand nombre.

Chapitre 5

Evaluation

Sommaire

5.1 Plateforme expérimentale 74
5.1.1 Hagrid 74
5.1.2 Kwak 74
5.2 Performances brutes de ForestGOMP 76
5.2.1 EPCC 76
5.2.2 Nested EPCC 77
5.2.3 La suite de benchmarks de la NASA 77
5.3 Ordonnancement d'applications au parallélisme irrégulier 79
5.3.1 BT-MZ, une application à deux niveaux de parallélisme aux comportements différents 79
5.3.2 MPU, une application au parallélisme imbriqué irrégulier dans le temps 80
5.4 Ordonnancement d'applications aux accès mémoire intensifs 83
5.4.1 STREAM 83
5.4.2 Nested-STREAM 84
5.4.3 Twisted-STREAM 85
5.4.3.1 100% de données distantes 85
5.4.3.2 66% de données distantes 86
5.4.4 Imbalanced-STREAM 87
5.4.5 LU 89
5.5 Composabilité, dimensionnement 91

Nous présentons dans ce chapitre les résultats d'expériences menées pour valider notre approche et en évaluer les performances. Nous nous intéressons dans un premier temps aux performances brutes du support exécutif ForestGOMP, avant d'évaluer l'impact des ordonnancements calculés par Cache et Memory sur les performances générales de plusieurs benchmarks et applications. Nous évaluons enfin la composition de notre environnement de programmation avec la bibliothèque d'algèbre linaire GotoBLAS à l'aide d'une expérience faisant intervenir plusieurs décompositions Cholesky.

5.1 Plateforme expérimentale

Cette section présente les spécificités des machines de calcul, baptisées *Hagrid* et *Kwak* utilisées pour évaluer les performances du support exécutif ForestGOMP.

5.1.1 Hagrid

La machine Hagrid contient 8 nœuds NUMA. Chacun d'entre eux est composé d'un processeur bicœur Opteron, présenté en figure 5.1, et de 8Go de mémoire. Les cœurs d'un processeur sont cadencés à 1,8GHz et ne partagent pas de mémoire cache: ils accèdent chacun à deux niveaux de cache qui leur sont propres, un cache L1 de 64ko et un cache L2 de 1024ko. Les nœuds NUMA sont connectés par un réseau de liens HyperTransport. Le facteur NUMA varie selon la position relative de l'emplacement mémoire accédé et du processeur effectuant l'accès. En pratique, nous avons observé trois valeurs typiques de facteur NUMA pour cette machine: environ $1,1$, $1,25$ et $1,4$. Nous en avons alors déduit la topologie du réseau interne présentée en figure 5.2, puisque les facteurs minimum correspondent à un seul saut dans le réseau. Cette topologie apparament étrange s'explique 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.

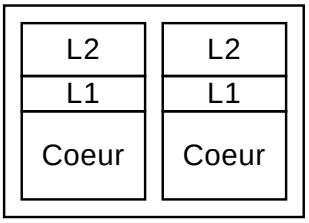


FIG. 5.1 – Processeur bicœur Opteron 865. La machine Hagrid en comporte 8.

5.1.2 Kwak

Bien que les machines Hagrid et Kwak disposent du même nombre de cœurs, leur architecture est différente. La machine Kwak dispose en effet de processeurs quadricœur Opteron de type Barcelona cadencés à 1,9GHz illustrés en figure 5.3. Chaque cœur accède à deux niveaux de cache privés L1 et L2 de taille 64ko et 512ko, ainsi qu'à un cache de niveau 3 de 2048ko, partagé entre les 4 cœurs d'un processeur. La machine compte 4 nœuds NUMA contenant chacun un processeur quadricœur et 8Go de mémoire. L'interconnexion de ces nœuds est représentée par la figure 5.4. Les facteurs NUMA mesurés sur cette machine sont présentés dans le tableau 5.1. On remarque qu'ils dépendent du type d'accès: la politique d'écriture sur cette machine nécessite de lire une ligne de cache avant de pouvoir écrire, générant ainsi plus de trafic.

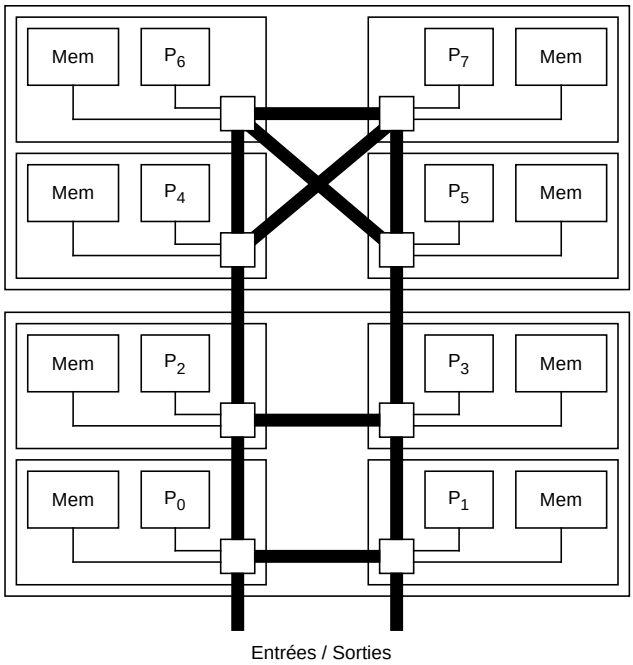


FIG. 5.2 – Architecture de la machine Hagrid.

Type d'accès	Local	Nœud voisin	Nœud opposé
Lecture	83 ns	98 ns ($\times$ 1.18)	117 ns ($\times$ 1.41)
Ecriture	142 ns	177 ns ($\times$ 1.25)	208 ns ($\times$ 1.46)

TAB. 5.1 – Latences d'accès mémoire (hors cache) en fonction de l'emplacement des données sur la machine Kwak.

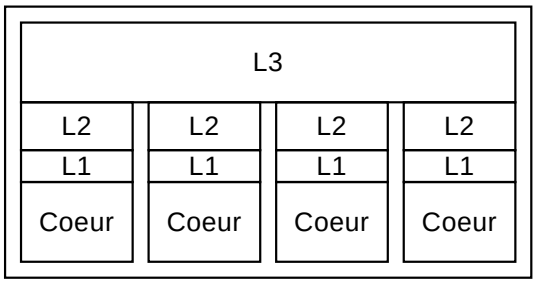


FIG. 5.3 – Processeur quadricœur Opteron 8347HE. La machine Kwak en comporte 4.

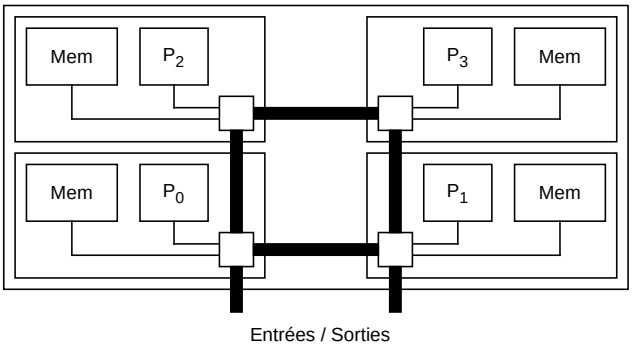


FIG. 5.4 – Architecture de la machine Kwak.

5.2 Performances brutes de ForestGOMP

Nous évaluons les performances brutes de notre support exécutif à l'aide du benchmark *EPCC* en version originale et à parallélisme imbriqué et de la suite de benchmarks *NAS NPB*.

5.2.1 EPCC

Le microbenchmark *EPCC* [Bul99] mesure les temps induits par l'invocation de mot-clé du langage OpenMP pour en déduire les surcoûts imputés au support exécutif. Ces mesures sont effectuées comme présenté en figure 5.2.1. Chaque construction OpenMP est exécutée un grand nombre de fois. Le benchmark affiche enfin les temps moyen, minimum et maximum pour chaque mot-clé testé. Notons T_1 le temps nécessaire à l'exécution séquentielle du code OpenMP mesuré, et T_P le temps d'exécution de ce même code cette fois-ci exécuté en parallèle par p processeurs. Le surcoût imputé au support exécutif, noté T_s, est calculé de la sorte: $T_s = T_p - T_1/p$.

```
  for (k = 0; k <= OUTERLOOPS; k++)
  {
    start = getclock ();
    for (j = 0; j < INNERLOOPS; j++)
#pragma omp parallel
      delay (delaylength);
    times[k] = (getclock () - start) * 1.0e6 / (double) INNERLOOPS;
  }
```

FIG. 5.5 – Mesure du surcoût induit par la création d'une région parallèle OpenMP par le benchmark EPCC.

Les performances obtenues par libGOMP3, ICC 11.0 et ForestGOMP sont présentées par le tableau 5.2. On observe qu'aucun de ces supports exécutifs ne se détache sin-

gulièrement des autres. Les performances de ForestGOMP et de libGOMP3 sont comparables. On peut cependant remarquer que les performances obtenues par ICC sur les barrières, les réductions et le mot-clé `single` sont légèrement en retrait par rapport aux autres supports exécutifs. En revanche, ICC dispose de verrous très efficaces lui permettant d'implémenter les constructions `lock` et `critical` de manière performante.

Mot-clé	libGOMP3	Intel ICC 11.0	ForestGOMP
atomic	0,30	0,39	0,29
barrier	2,91	5,12	3,31
critical	4,01	2,20	4,03
for	2,91	5,07	3,34
lock	3,95	2,21	3,88
parallel	7,15	8,15	7,02
parallel for	8,15	8,18	7,46
reduction	8,74	15,66	7,58
single	3,74	19,50	2,56

TAB. 5.2 – Surcoûts moyens, en microsecondes, de trois supports exécutifs OpenMP différents obtenus en exécutant le benchmark EPCC sur la machine Kwak.

5.2.2 Nested EPCC

Afin d'évaluer le surcoût associé à l'exécution de région parallèles imbriquées, Dimakopoulos et al.[DHP10] ont proposé une version imbriquée du microbenchmark EPCC. Ici, chaque test d'EPCC est lancé un grand nombre de fois à l'intérieur d'une région parallèle OpenMP. La constante `INNERLOOPS` prend une valeur suffisamment grande pour que le temps de création de la région parallèle externe puisse être négligé. Sur la machine Kwak, Nested EPCC génère 16 threads sous la forme de 4 régions imbriquées dans une région parallèle à 4 threads. Les résultats obtenus par les supports exécutifs libGOMP3, Intel ICC et ForestGOMP sont synthétisés sur le tableau 5.3. On peut constater que libGOMP3 peine à gérer efficacement le parallélisme imbriqué. Une raison à cela reste le surcoût associé à la création de threads noyaux, qui n'est cependant pas visible dans la version originale d'EPCC, puisque libGOMP3 optimise le cas particulier où plusieurs régions parallèles sont exécutées en séquence, en permettant la réutilisation de threads du premier niveau de parallélisme. Les supports exécutifs ForestGOMP et ICC reposent eux sur une bibliothèque de threads de niveau utilisateur et exhibent des surcoûts très inférieurs.

5.2.3 La suite de benchmarks de la NASA

Depuis une vingtaine d'années, la NASA distribue une suite de benchmarks [BBB+91] pour les architectures parallèles, représentatifs des types de problèmes que l'on est amené à traiter dans le domaine du calcul hautes performances. Chaque application dispose de différentes versions, ou *classes*, qui diffèrent par la taille du jeu de données

Mot-clé	libGOMP3	Intel ICC 11.0	ForestGOMP
atomic	0,52	0,87	0,49
barrier	75,51	26,98	27,56
critical	12,90	39,01	4,13
for	80,44	28,17	27,33
lock	4,69	4,41	4,06
parallel	3209,75	304,94	171,66
parallel for	3222,49	311,58	170,56
reduction	3220,41	452,20	171,58
single	59,91	81,64	11,47

TAB. 5.3 – Surcoûts moyens, en microsecondes, induits par l'invocation de mot-clés OpenMP en contexte de parallélisme imbriqué 4×4, en fonction du support exécutif utilisé sur la machine Kwak.

qu'ils ont à traiter. La version 3 des *NPB NAS Parallel Benchmarks* dispose d'une implémentation OpenMP de la plupart des applications de cette suite. Le tableau 5.4 montre les performances obtenues en exécutant la classe B de chaque application de la suite de benchmarks NPB 3.3 sur la machine Kwak, à l'aide des supports exécutifs libGOMP3, ICC 11.0 et ForestGOMP. Les accélérations présentées sont relatives au temps d'exécution de la version séquentielle de chaque application compilée avec ICC, qui bénéficie de meilleures optimisations que GCC.

Le support exécutif ForestGOMP se comporte mieux sur les applications BT et MG grace à une répartition de threads plus stable que ses concurrents. Le support exécutif d'Intel obtient quant à lui les meilleures performances sur EP et SP grâce à de meilleures optimisations de compilation. En effet, les temps d'exécution obtenus par les versions séquentielles de ces applications compilées avec GCC sont très supérieurs à ceux obtenus avec ICC: 157s contre 88s sur EP, 655s contre 493s sur SP. Les verrous du support exécutif d'Intel étant plus performants que ceux de libGOMP et ForestGOMP, ICC se comporte mieux sur l'application UA qui utilise plus de 300000 verrous. Les performances obtenues par les trois supports exécutifs sur CG, FT et LU, dont l'accélération excède 16 grâce aux effets de cache, sont comparables.

Support exécutif	BT	CG	EP	FT	LU	MG	SP	UA
libGOMP3	11,5	12,4	9,1	10,1	17	5,1	3,2	3
ForestGOMP	12,1	12,4	9,0	9,9	17	5,6	3,6	3,1
Intel ICC	11,2	12,3	14,9	10,1	17,9	4,7	5,3	6,6

TAB. 5.4 – Accélérations obtenues en exécutant la suite de benchmarks NPB 3.3 sur la machine Kwak.

5.3 Ordonnancement d'applications au parallélisme irrégulier

Cette section présente l'évaluation de notre support exécutif, en particulier de l'ordonnanceur Cache, sur deux applications au parallélisme irrégulier: l'application multi-zone *BT-MZ* de la suite de benchmarks NBP 3.3, et l'application de reconstruction de surface implicite *MPU*.

5.3.1 BT-MZ, une application à deux niveaux de parallélisme aux comportements différents

Certaines applications de la suite de benchmark NPB sont disponibles en version multi-zones: le domaine de calcul est découpé en zones qui sont traitées en parallèle. Elles font donc intervenir plusieurs niveaux de parallélisme, le plus souvent dans le but d'expérimenter les approches de programmation hybride, comme des processus MPI embarquant des régions parallèles OpenMP. Au cours de sa thèse, Samuel Thibault a adapté l'application BT-MZ, qui simule la dynamique des fluides dans un domaine 3D, afin qu'elle exhibe deux niveaux de parallélisme OpenMP: le parallélisme externe chargé de découper le domaine de calcul en zones, et le parallélisme interne invoqué à l'intérieur de chacune des zones. La taille des zones est irrégulière. Il existe par exemple un facteur 25 entre la plus grande et la plus petite zone. Ce paramètre rend le parallélisme externe irrégulier. En revanche, la charge de travail associée à chaque thread d'une même zone est identique, le parallélisme interne est donc lui régulier.

Nous avons évalué les performances de la classe C de BT-MZ sur la machine Kwak, en faisant varier le nombre de threads créés par les régions parallèles externes et internes. Les résultats obtenus, figurant dans le tableau 5.5, confirment l'efficacité du support du parallélisme imbriqué de ForestGOMP. Les supports exécutifs libGOMP3 et ICC obtiennent leurs meilleures performances, avoisinant une accélération de 14, en créant un thread par cœur, sans surcharge. De son côté, ForestGOMP atteint une accélération de $14,5$ en créant 32 équipes de 8 threads. Surcharger les processeurs de la machine offre à l'ordonnanceur Cache un choix plus large quand il s'agit de voler du travail pour rééquilibrer la charge.

La difficulté majeure pour parvenir à exécuter efficacement cette application provient de l'irrégularité du parallélisme externe. Le temps d'exécution des threads d'une région parallèle interne est proportionnel à la taille du domaine qu'ils traitent. En cas de déséquilibre majeur sur la machine, l'algorithme de vol de travail de Cache est amené à rompre des relations d'affinités entre threads pour occuper les cœurs inactifs. Afin de limiter ce phénomène, nous proposons au programmeur de transmettre la taille associée à chaque zone au support exécutif. ForestGOMP attache alors cette information à la bulle de l'équipe concernée, qualifiant ainsi la charge de travail des threads qui la composent. La dernière colonne du tableau 5.5 montre que tenir compte de cette information pour distribuer les équipes de façon à attribuer une quantité de travail comparable à chaque processeur permet d'atteindre une accélération de 15.

Externe × Interne	libGOMP3	Intel ICC	ForestGOMP (Cache)	
			Original	Charge de travail
4×4	9.4	13.8	14.1	14.1
16×1	**14.1**	**13.9**	14.1	14.1
16×2	11.8	9.2	14.1	14.2
16×4	11.6	6.1	14.1	14.9
16×8	11.5	4.0	14.4	**15.0**
32×1	12.6	10.3	13.5	13.8
32×2	11.6	5.9	14.2	14.2
32×4	11.2	3.4	14.3	14.8
32×8	10.9	2.8	**14.5**	14.7

TAB. 5.5 – Accélérations obtenues à partir de la classe C de BT-MZ sur la machine Kwak, en fonction du nombre de threads créés depuis les régions parallèles internes et externes, et du support exécutif utilisé.

5.3.2 MPU, une application au parallélisme imbriqué irrégulier dans le temps

Les technologies d'acquisition 3D permettent aujourd'hui de numériser un objet entier pour en obtenir une représentation sous forme de nuage de points. Numériser un objet avec une précision poussée génère ainsi des représentations comportant plusieurs milliards de points. La surface qu'ils modélisent peut être reconstruite afin d'obtenir un rendu satisfaisant de l'objet 3D plein. L'algorithme *MPU*, pour *Multi-level Partition of Unity*, a été développé dans ce but. Il s'appuie sur une approximation implicite du nuage de points à l'aide de fonctions mathématiques. Ce fonctionnement est présenté en figure 5.6: on cherche ici à approximer les points contenus dans chacun des cercles en calculant une quadrique $Q(x) = 0$. L'équation globale $f(x) = 0$ est obtenue en sommant ces quadriques. Cette équation, aussi complexe soit elle, facilite grandement l'application de déplacements et transformations sur l'objet 3D.

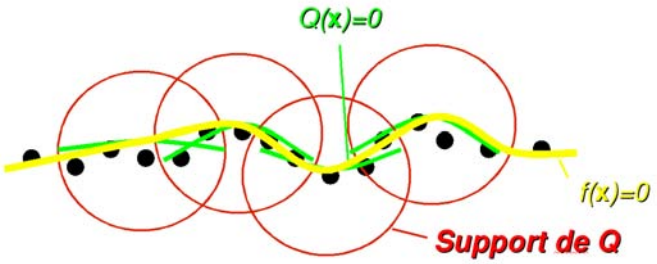


FIG. 5.6 – Approximation d'un ensemble de points à l'aide de l'algorithme MPU.

En pratique, MPU commence par construire un maillage cubique grossier englobant tous les points de la surface. Ce domaine 3D est ensuite découpé en 8 sous-cubes à l'intérieur desquels on applique l'algorithme d'approximation, qui évalue pour chaque

sous-domaine si la quadrique calculée s'approche suffisament de la surface de l'objet. Si tel est le cas, la quadrique est conservée, et le travail associé au sous-domaine concerné est terminé. Dans le cas contraire, le maillage est raffiné localement de manière à redécouper ce sous-domaine en 8. Ce fonctionnement est représenté par la figure 5.7.

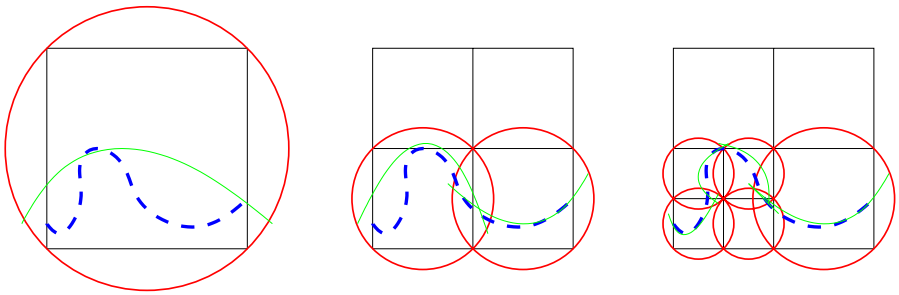


FIG. 5.7 – Représentation 2D de l'approximation d'une surface 3D par MPU par raffinement successif du maillage associé à l'objet numérisé. Le domaine est subdivisé chaque fois que l'approximation calculée n'est pas satisfaisante.

La parallélisation de cet algorithme à l'aide d'OpenMP est directe: il a suffi d'ajouter la ligne indiquée en gras sur la figure 5.3.2. Des threads OpenMP vont donc être créés récursivement au cours de chaque raffinement du domaine considéré en sous-cubes. On peut déjà remarquer que sans activer le support du parallélisme imbriqué, l'accélération maximale que l'on peut obtenir est limitée à 8, étant donné que l'application créé 8 threads au premier niveau de parallélisme. D'un autre côté, l'activation du support pour le parallélisme imbriqué peut provoquer la création d'un très grand nombre de threads. En pratique, l'arbre de récurrence atteint souvent une profondeur de 15 et une largeur de plusieurs centaines voire plusieurs milliers de threads.

```
void ImplicitOctCell::buildFunction () {
  /* Calcul de l'approximation sur le cube. */
  if (computeLA () < maxError)
    return;

  /* Approximation insuffisante,
   * raffiner en découpant en 8 sous-cubes */

  /* Rappeler récursivement l'approximation
   * dans chaque sous-cube créé. */
```

#pragma omp parallel for

```
    for (i = 0; i < 8; i++)
      subCell[i]->buildFunction ();
 }
```

FIG. 5.8 – Parallélisation de l'algorithme MPU à l'aide d'OpenMP.

Nous avons exécuté MPU sur la machine Hagrid afin de comparer les performances de ForestGOMP à celles des supports exécutifs libGOMP3 et Intel ICC. La reconstruction du nuage de 437644 points choisi pour cette expérience conduit à la création de 101185 threads dans le cas où le support exécutif accepte la génération de parallélisme imbriqué. Les résultats présentés dans le tableau 5.6 montrent une fois encore que la création massive de threads noyaux handicape les performances du libGOMP3. En particulier, en désactivant le support du parallélisme imbriqué pour ce support exécutif, on atteint une accélération légèrement plus élevée, de l'ordre de 5.

Les performances obtenues par ICC, qui bénéficie d'une création de thread moins coûteuse, sont bien meilleures. Ce support exécutif atteint ainsi une accélération de $7,9$. Haab et al.[STG+02] ont proposé une extension d'OpenMP, intégrée depuis au compilateur ICC, qui facilite la parallélisation des algorithmes de type *diviser pour régner*. Cette extension implémente le principe de *workqueuing*, rendant possible la création de tâches ordonnancées dans des files définies par le programmeur. L'équilibrage de charge s'effectue par vol de tâche d'une file vers une autre. L'adaptation du code de MPU à ce modèle permet d'atteindre une accélération de $9,7$.

Afin d'évaluer l'impact de l'ordonnancement sur les performances de ForestGOMP, nous avons dans un premier temps exécuté MPU à l'aide de l'ordonnanceur *Null*, qui place tous les threads sur la liste d'ordonnancement la plus générale de la topologie. Aucune stratégie d'ordonnancement particulière n'est donc appliquée ici, et tous les cœurs de la machine peuvent exécuter n'importe quel thread de l'application. L'équilibrage de charge n'est donc plus un problème dans ce cas, et cette solution permet d'atteindre une accélération de $8,6$. En revanche, l'exécution désordonnée des threads de l'application et la contention liée à l'accès concurrent de 16 cœurs sur une même liste d'ordonnancement limitent cette accélération. L'utilisation de l'ordonnanceur Cache est parfaitement adaptée ici. Sa capacité à ordonnancer les groupes de threads de façon compacte assure une exécution cohérente des threads issus d'un même domaine MPU, menant à une accélération de 14. Au moment de répartir la charge, l'ordonnanceur vole du travail de proche en proche sur la machine, ce qui explique en partie pourquoi les performances obtenues par Cache dépassent largement celles obtenues par la version Intel Workqueuing, qui vole du travail entre n'importe quels cœurs de la machine, sans se soucier de la topologie et des relations d'affinité entre les threads de l'application.

libGOMP3	Intel ICC 11.0		ForestGOMP	
	OpenMP	Workqueuing	Null	Cache
4,2	7,9	9,7	8,6	14

TAB. 5.6 – Accélérations obtenues en exécutant MPU à l'aide de différents supports exécutifs sur la machine Hagrid.

5.4 Ordonnancement d'applications aux accès mémoire intensifs

Cette section présente l'évaluation des performances de ForestGOMP quand il s'agit d'ordonnancer des applications dont les performances dépendent de la bande passante mémoire disponible. Nous étudions pour ce faire différentes déclinaisons du benchmark mémoire *STREAM* ainsi qu'une implémentation OpenMP de la factorisation *LU*.

5.4.1 STREAM

STREAM[McC07] est un benchmark synthétique, développé en langage C et parallélisé à l'aide d'OpenMP, qui détermine la bande passante mémoire soutenue d'une machine de calcul en effectuant des opérations arithmétiques simples sur des vecteurs de flottants. Ces vecteurs sont suffisamment grands, de l'ordre de 20 millions de flottants double précision, pour limiter les gains de performance liés à la présence du cache. Ils sont initialisés selon la politique d'allocation *first-touch* afin que les pages mémoire soient allouées au plus près du premier thread qui y accèdera. Le benchmark fait intervenir 4 opérations successives sur un ensemble de trois vecteurs notés A, B, C:
- **Copy**: $C = A$
- **Scale**: $B = \lambda C$, avec λ un scalaire
- **Add**: $C = A + B$
- **Triad**: $A = B + \lambda C$, avec λ un scalaire

Ces opérations sont exécutées un grand nombre de fois successivement, et le benchmark relève les bandes passantes minimum, maximum et moyennes pour chacune d'entre elles.

Le tableau 5.7 présente les performances de STREAM sur la machine Kwak, en fonction du support exécutif utilisé et des opérations exécutées. Le support exécutif libGOMP3 est celui qui se comporte le moins bien ici. Il est une fois de plus victime du comportement de l'ordonnanceur préemptif du système d'exploitation. Il arrive parfois que plusieurs threads soient préemptés en même temps et s'échangent leur position sur la machine, mettant en défaut la distribution initiale des données effectuée selon la politique d'allocation *first-touch*. Nous avons pu observer ce phénomène à l'aide de traces d'exécution indiquant la position des processus légers au cours de l'exécution du benchmark. L'utilisation de la variable d'environnement `GOMP_CPU_AFFINITY` pour fixer l'emplacement des threads OpenMP permet de résoudre ce problème. Les performances de libGOMP3 deviennent alors comparables à celles de ForestGOMP, au prix d'une manipulation non portable. ForestGOMP désactive la préemption par défaut, et obtient des performances plus stables : l'ordonnanceur Cache place un thread par cœur et le laisse s'exécuter sans interruption jusqu'à la fin de l'opération en cours. Le support exécutif du compilateur ICC obtient les meilleures performances ici, grâce en particulier à des optimisations supplémentaires à la compilation comme la vectorisation des opérations très simples effectuées dans STREAM.

Opération	libGOMP3		Intel ICC		ForestGOMP	
	Min-Max	Moyenne	Min-Max	Moyenne	Min-Max	Moyenne
Copy	8,5-12,1	10,6	15,2-15,5	15,3	14,3-14,4	14,4
Scale	8,4-11,9	10,6	15,1-15,4	15,3	14,3-14,4	14,4
Add	9,3-12,4	11,1	15,2-15,4	15,2	14,3-14,4	14,4
Triad	9,3-12,4	11,1	15,2-15,5	15,2	14,3-14,4	14,4

TAB. 5.7 – Bandes passantes, en Go/s, obtenues par le benchmark STREAM sur la machine Kwak.

5.4.2 Nested-STREAM

Nous avons développé le benchmark *Nested-STREAM* pour permettre l'exécution en parallèle de plusieurs instances de STREAM. Il exhibe ainsi deux niveaux de parallélisme OpenMP. Une première région parallèle, externe, détermine le nombre d'instances de STREAM qui s'exécuteront de manière concurrente. Chaque instance de STREAM s'exécute elle-aussi en parallèle, comme la version originale de ce benchmark présentée dans la section précédente, et travaille sur son propre jeu de vecteurs de flottants initialisés là encore selon la politique *first-touch*.

Afin d'adapter le parallélisme de l'application à l'achitecture de la machine Kwak, nous exécutons 4 instances de STREAM créant 4 threads chacune. Ainsi, chaque nœud NUMA exécute sa propre instance. Les résultats obtenus sont synthétisés dans le tableau 5.8. ForestGOMP obtient ici les meilleures performances grâce sa gestion efficace du parallélisme imbriqué. L'ordonnanceur Cache fait en sorte que chaque processeur quadricœur exécute sa propre instance de STREAM. La politique d'allocation *first-touch* garantit alors que l'ensemble des pages mémoire accédées par cette instance sont allouées sur le banc mémoire local. Les threads d'une même instance accèdent ainsi localement à leurs données, ce qui explique les bonnes performances obtenues par ForestGOMP, contrairement à libGOMP3 et ICC qui n'assurent pas l'exécution compacte des threads issus des régions parallèles imbriquées.

Opération	libGOMP3		Intel ICC		ForestGOMP	
	Min-Max	Moyenne	Min-Max	Moyenne	Min-Max	Moyenne
Copy	1,55-1,98	1,79	2,50-3,37	3,07	3,61-3,63	3,62
Scale	1,61-2,02	1,81	2,67-3,41	2,96	3,61-3,63	3,62
Add	1,67-2,10	1,94	2,44-3,52	2,92	3,61-3,63	3,62
Triad	1,51-2,17	1,89	2,51-3,39	3,02	3,61-3,63	3,62

TAB. 5.8 – Bandes passantes, en Go/s par nœud NUMA, calculée par le benchmark Nested-STREAM sur la machine Kwak.

5.4.3 Twisted-STREAM

Nous avons modifié l'application Nested-STREAM afin d'analyser l'impact des changements de motif d'accès à la mémoire sur les performances. Cette application, baptisée *Twisted-STREAM*, comporte deux phases distinctes. La première correspond en tout point à l'application Nested-STREAM: quatre instances de STREAM sont exécutées en parallèle sur la machine Kwak. Au cours de la seconde phase, nous changeons le motif d'accès à la mémoire: l'équipe i travaille désormais sur les vecteurs alloués par l'équipe $i + 1$. Le tableau 5.9 montre l'écart de performances entre les deux phases de l'application, obtenues sur l'opération Triad. Cet écart s'explique par le fait que le support exécutif ne détecte pas le changement de motif d'accès aux données de façon automatique. Les threads et leurs données restent en place les accès mémoire distants se multiplient. Les performances obtenues au cours de la phase 2 sont donc fortement dégradées par rapport à celles obtenues en phase 1.

Opération	libGOMP3		ICC		ForestGOMP (Cache seul)	
	Min-Max	Moyenne	Min-Max	Moyenne	Min-Max	Moyenne
Triad, phase 1	1,51-2,17	1,89	2,51-3,39	3,02	3,61-3,63	3,62
Triad, phase 2	1,03-1,27	1,19	1,89-2,32	2,01	1,81-1,93	1,90

TAB. 5.9 – Bandes passantes, en Go/s par nœud NUMA, calculée par le benchmark Twisted-STREAM sur la machine Kwak.

Avec l'aide de l'interface de programmation de ForestGOMP, le programmeur a la possibilité d'attacher une zone mémoire à une région parallèle. Cet attachement signale au support exécutif, plus particulièrement à l'ordonnanceur Memory dans ce cas, que ces données doivent être placées au plus près des threads auxquels elles sont attachées. Nous étudions la réaction du support exécutif ForestGOMP à ce changement de motif d'accès à l'aide de deux versions de l'application Twisted-STREAM qui diffèrent sur la quantité de données (en pratique, le nombre de vecteurs) accédées à distance au cours de la phase 2.

5.4.3.1 100% de données distantes

Dans une première version, les trois vecteurs de flottants A, B et C sont accédés à distance au cours de la phase 2. Cette situation peut être traitée en s'appuyant sur la politique d'allocation *next-touch* lorsqu'elle est disponible, qui migre alors les données au plus près du prochain thread qui y accède. L'ordonnanceur Memory préfère déplacer les threads vers les nouvelles données qui leur sont affectées. En effet, la migration mémoire reste bien plus coûteuse que le déplacement de 16 threads sur cette machine. De plus, le temps de calcul est extrêmement court: chaque opération STREAM s'exécute en un temps ne dépassant jamais quelques dizaines de microsecondes. Pour estimer à partir de quelle durée de calcul la migration mémoire devient intéressante, nous faisons varier le nombre d'exécutions des opérations STREAM. La figure 5.9 montre les performances obtenues, en temps d'exécution normalisés par rapport à exécuter Twisted-STREAM sans réagir en phase 2, par la politique *next-touch* et l'ordonnanceur Memory

en fonction du temps de calcul affecté à STREAM. Ces courbes confirment le surcoût important qui accompagne la migration de données. L'utilisation de *next-touch* devient intéressante au delà de 25 itérations STREAM. La solution offerte ici par Memory ne dépend pas du temps de calcul et permet de réduire de 20% le temps d'exécution de la phase 2 de l'application.

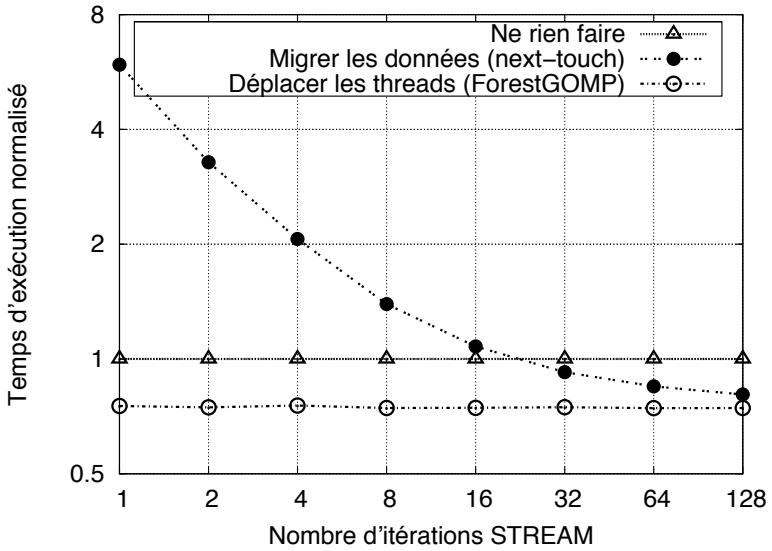


FIG. 5.9 – Temps d'exécution de plusieurs politiques de placement de threads et de données sur le benchmark Twisted-STREAM, dans le cas où les trois vecteurs sont accédés à distance au cours de la phase 2.

5.4.3.2 66% de données distantes

Dans un deuxième temps, nous avons modifié Twisted-STREAM de façon à complexifier les accès mémoire en phase 2. Ici, seuls deux des trois vecteurs utilisés par STREAM sont accédés à distance au cours de cette phase. Nous nous focalisons dans cette partie sur l'opération STREAM Triad, la seule faisant intervenir les trois vecteurs. Parmi les deux vecteurs distants, un est accédé en lecture, l'autre en écriture. Nous abordons plusieurs approches pour traiter ce changement de motif d'accès, toutes testées sur la machine Kwak:

- **Migrer un vecteur (R)**: la machine Kwak présentant des facteurs NUMA différents selon le type d'accès aux données, nous choisissons de ne migrer ici que le vecteur accédé en lecture.
- **Migrer un vecteur (W)**: même chose, concernant cette fois le vecteur accédé en écriture.

– **next-touch**: l'utilisation de la politique *next-touch* aura ici pour effet de migrer les deux vecteurs distants vers l'emplacement des threads.
– **ForestGOMP (Memory)**: l'ordonnanceur Memory choisit lui de migrer les threads et le 3e vecteur vers le nœud NUMA qui héberge les deux vecteurs distants.

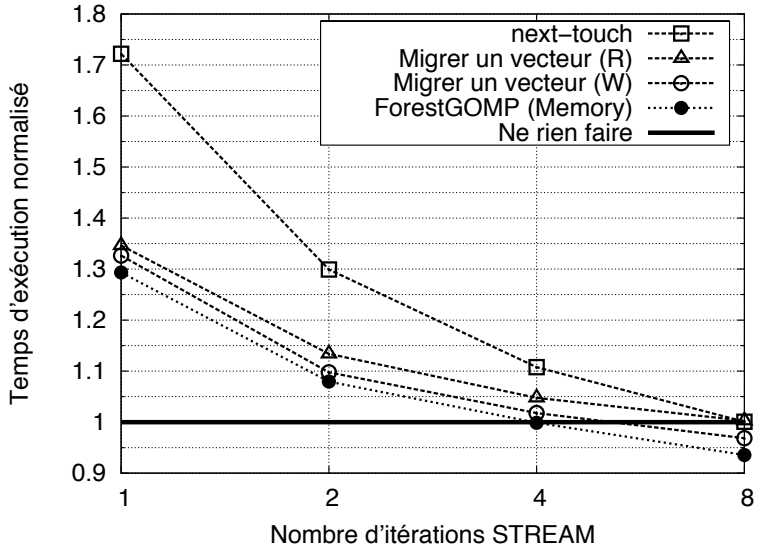
Les performances obtenues par chacune de ces approches sont présentées sur la figure 5.10, sous forme de temps d'exécution normalisés par rapport à ne rien faire en phase 2 de Twisted-STREAM. Là encore, nous faisons varier la quantité de calcul en exécutant plusieurs fois l'opération Triad successivement. La figure 5.10(a) présente les performances de chaque solution quand la quantité de calcul est faible. La politique *next-touch* souffre d'un surcoût supérieur aux autres approches, puisqu'elle fait intervenir la migration de deux vecteurs au lieu d'un. Le facteur NUMA associé aux écritures distantes étant supérieur à celui associé aux lectures sur cette machine, la migration du vecteur accédé en écriture obtient de meilleurs résultats. Le surcoût associé à l'ordonnancement de Memory est comparable à ceux des politiques ne migrant qu'un vecteur. Les performances obtenues par ForestGOMP sont cependant légèrement meilleures, puisque l'ordonnanceur Memory assure que tous les threads accèdent à des données locales après répartition, ce qui n'est pas le cas lorsqu'on décide de ne migrer qu'un vecteur. Memory continue de bien se comporter quand on augmente la durée du calcul, comme présenté en figure 5.10(b), où ForestGOMP permet de réduire le temps d'exécution de la phase 2 de 10% en moyenne.

5.4.4 Imbalanced-STREAM

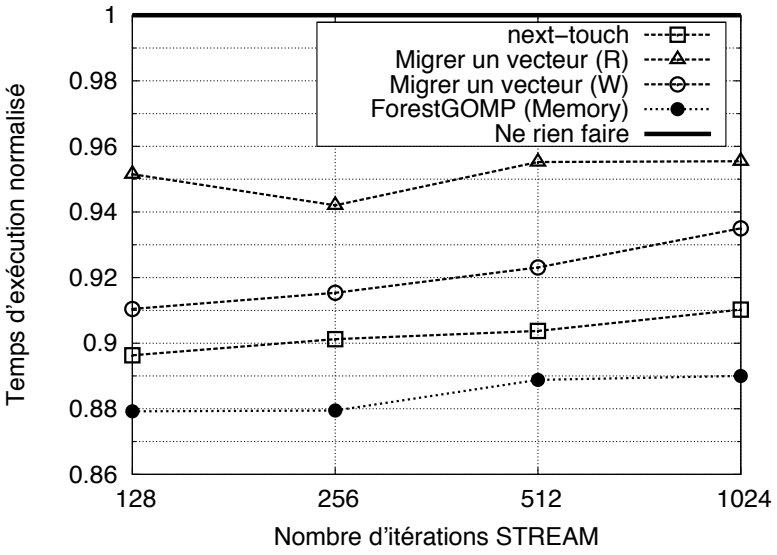
Les différentes versions du benchmark STREAM étudiées jusqu'alors exhibent un parallélisme régulier. Nous avons par la suite développé l'application *Imbalanced-STREAM* pour évaluer la dynamicité de Memory quant aux déséquilibres de charge. Imbalanced-STREAM étend Nested-STREAM pour permettre de définir une charge de travail associée à chaque instance de STREAM. En pratique, une charge de travail de 10 signifie que les threads exécutant l'instance de STREAM considérée auront à exécuter 10 fois la quantité de travail effectuée par la version originale de STREAM. Nous affectons ainsi une charge de 15 aux instances 0 et 1, une charge de 30 à l'instance 2 et une charge de 1 à l'instance 3. Nous doublons aussi le nombre de threads créés par une instance de STREAM. Chaque cœur de la machine exécute maintenant deux threads, ce qui rend possible l'équilibrage de charge si certains cœurs deviennent inactifs. Les temps d'exécution des équipes de threads traitant chacune des instances, dans le cas où ForestGOMP n'équilibre pas la charge sur la machine, sont présentés dans le tableau 5.10.

On remarque que l'équipe 3, à laquelle on attribue une charge de 1, termine son exécution bien plus tôt que les autres, laissant une partie des cœurs de la machine inactifs. À l'inverse, le temps total correspond au temps d'exécution de l'équipe 2, étant affectée de la charge de travail la plus conséquente.

Nous expérimentons tout d'abord le vol de travail de l'ordonnanceur Cache pour équilibrer la charge quand l'équipe 3 termine son exécution. L'algorithme a ici le choix de l'équipe depuis laquelle voler des threads. Les temps d'exécution obtenus en fonction de l'équipe choisie par l'algorithme de vol sont représentés dans le tableau 5.11. En comparant ces temps avec ceux du tableau 5.10, on remarque que l'utilisation de cet



(a) Performances de chaque solution pour une quantité de calcul faible.



(b) Performances de chaque solution pour une quantité de calcul supérieure.

FIG. 5.10 – Temps d'exécution de plusieurs politiques de placement de threads et de données sur le benchmark Twisted-STREAM, dans le cas où seuls deux des trois vecteurs sont accédés à distance au cours de la phase 2.

Temps d'exécution	Moyenne	Min	Max
Équipe 0	1,6595	1,6578	1,6613
Équipe 1	1,4879	1,4876	1,4881
Équipe 2	2,7150	2,7113	2,7170
Équipe 3	0,1336	0,1335	0,1338
Temps total	2,7151	2,7114	2,7171

TAB. 5.10 – Temps d'exécution, en secondes, de chaque équipe de threads du benchmark Imbalanced-STREAM exécuté sur la machine Kwak, dans le cas où le vol de travail de ForestGOMP est désactivé.

algorithme dégrade les performances. En effet, bien que l'équilibrage de charge soit meilleur ici, Cache déplace des threads d'un nœud NUMA vers un autre. Ces threads accèdent donc à leurs données à distance, et participent à l'apparition de contention sur le bus mémoire. En particulier, on observe que voler des threads depuis les équipes 0 ou 1 dégrade les performances de l'équipe 2, qui pourtant continue d'accéder localement à ses données.

Temps d'exécution	Vol depuis l'équipe 0	Vol depuis l'équipe 1	Vol depuis l'équipe 2
Équipe 0	1,9755	1,5228	1,5252
Équipe 1	1,5226	1,9755	1,5306
Équipe 2	2,9716	2,9692	3,4574
Équipe 3	0,1334	0,1333	0,1329
Temps total	**2,9696**	**2,9693**	3,4575

TAB. 5.11 – Temps d'exécution, en secondes, de chaque équipe de threads du benchmark Imbalanced-STREAM exécuté sur la machine Kwak, dans le cas où ForestGOMP vole des threads pour équilibrer la charge après la terminaison de l'équipe 3.

La composition des ordonnanceurs Cache et Memory permet d'obtenir de meilleurs résultats ici, en offrant en particulier la possibilité à l'algorithme de vol de travail de migrer les données accédées par les threads que l'on déplace. En pratique, Memory utilise la politique d'allocation *next-touch* pour marquer les données attachées aux threads volés comme étant à migrer lors du prochain accès. Les performances obtenues sont présentées dans le tableau 5.12. Voler depuis l'équipe 2 se trouve être la meilleure solution ici, étant donné que la charge de travail affectée à cette équipe dépasse celle des autres.

5.4.5 LU

Les applications que nous avons présentés jusqu'à présent disposent de motifs d'accès à la mémoire relativement réguliers. Il est de ce fait facile pour le programmeur d'application d'exprimer les changements relatifs aux affinités entre threads et données. Il existe des applications pour lesquelles cette information est plus difficile à fournir, en

Temps d'exécution	Vol depuis l'équipe 0	Vol depuis l'équipe 1	Vol depuis l'équipe 2
Équipe 0	1,4401	1,7564	1,7374
Équipe 1	1,7453	1,4285	1,7720
Équipe 2	2,7178	2,7155	2,2843
Équipe 3	0,1632	0,1627	0,1629
Temps total	2,7179	2,7156	**2,2844**

TAB. 5.12 – Temps d'exécution, en secondes, de chaque équipe de threads du benchmark Imbalanced-STREAM exécuté sur la machine Kwak, dans le cas où ForestGOMP vole des threads **et migre les données associées** pour équilibrer la charge après la terminaison de l'équipe 3.

raison de leur complexité d'une part, mais aussi des limites d'expressivité de l'interface de programmation de ForestGOMP permettant d'attacher des données aux régions parallèles.

L'algorithme *LU* de factorisation de matrices illustre bien ce problème. Son implémentation parallèle découpe la matrice en blocs manipulés par une bibliothèque d'algèbre linéaire. À chaque étape, un nouveau bloc pivot est traité sur la diagonale de la matrice. Les valeurs des blocs lignes et colonnes correspondants sont alors mises à jour, suivies de la sous-matrice non traitée. La même résolution est ensuite appelée récursivement à partir de cette sous-matrice. Ainsi, à chaque étape de l'algorithme 2 présenté plus bas, on travaille sur une matrice plus petite, sous-partie de la matrice initiale. Le découpage en blocs de la matrice est représenté sur la figure 5.11. Les fonctions de ForestGOMP permettant d'attacher des données aux régions parallèles ne s'appliquent qu'aux zones de mémoire contiguës. Attacher une donnée à plusieurs dimensions, comme une sous-matrice ici, peut vite s'avérer fastidieux.

Nous lançons dans un premier temps l'application LU en prenant soin de distribuer en tourniquet les pages mémoire associées à la matrice traitée sur les différents nœuds NUMA de la machine, de façon à maximiser la bande passante. Nous mesurons dans ce cas un temps de $388,9$ s, correspondant à des performances de $60,3$ GFlop/s, pour factoriser une matrice large de 32768 flottants double précision, chaque dimension de la matrice étant découpée en 64 blocs.

Dans un deuxième temps, nous cherchons à améliorer la localité entre les threads et les blocs de la matrice qu'ils traitent. Pour ce faire, nous nous reposons sur une approche paresseuse qui migre les données de la matrice selon la politique *next-touch* de façon automatique sur consultation de compteurs matériels. ForestGOMP relève ainsi le pourcentage d'accès distants de façon régulière. Lorsque ce taux dépasse une valeur seuil, définie par le programmeur, les pages mémoire associées à la matrice sont marquées comme étant à migrer la prochaine fois qu'un thread y accède. La valeur seuil nécessite d'être définie par l'expérimentation: définir une valeur trop basse aura pour effet d'effectuer des migrations mémoire très souvent, dégradant les performances de l'application, tandis qu'adopter une valeur trop haute handicape la réactivité du support exécutif. Ici, déclencher une migration *next-touch* quand le taux d'accès mémoire

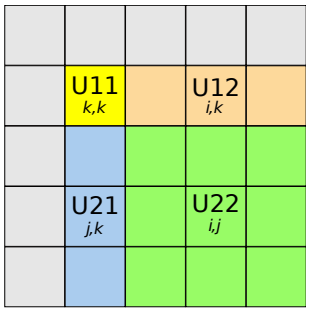


FIG. 5.11 – Factorisation LU par blocs

Algorithme 2 Factorisation LU

1: **for** k = 1 to n **do**
2: Factorize $U11_{(k,k)}$
3: **for** i = (k+1) to n **do**
4: Solve $U11_{(k,k)}X = U12_{(i,k)}$
5: $U12_{(i,k)} \leftarrow X$
6: **end for**
7: **for** j = (k+1) to n **do**
8: Solve $XU11_{(k,k)} = U21_{(k,j)}$
9: $U21_{(k,j)} \leftarrow X$
10: **end for**
11: **for** i = (k+1) to n **do**
12: **for** j = (k+1) to n **do**
13: $U22_{(i,j)} -= U12_{(i,k)}U21_{(k,j)}$
14: **end for**
15: **end for**
16: **end for**

distant dépasse les 25% permet de diminuer le temps de la factorisation de 30%, jusqu'à $298,2$ s, soit $80,78$ GFlop/s.

5.5 Composabilité, dimensionnement

Nous évaluons la composition d'OpenMP, exécuté par ForestGOMP, et de la version `pthread` de la bibliothèque d'algèbre linéaire GotoBLAS à l'aide de l'application Cholesky. Cette application décompose une matrice A en un produit d'une matrice triangulaire L et de sa transposée, et fait intervenir plusieurs opérations BLAS différentes.

Le but de l'expérience que nous avons mené consiste à exécuter 16 opérations Cholesky indépendantes sur la machine Kwak en cherchant à minimiser le temps total d'exécution. Une première façon de faire consiste à exécuter les 16 opérations Cholesky en séquence sur la machine. Chaque instance de Cholesky disposent ainsi des 16 cœurs de Kwak à sa disposition. Les performances d'une telle approche, sur lesquelles nous nous baserons pour évaluer la composition OpenMP+GotoBLAS, avoisinent les 80 GFlop/s.

Afin d'exécuter plusieurs opérations en parallèle, nous ajoutons une région parallèle OpenMP qui déterminera le nombre d'opérations Cholesky à exécuter de manière concurrente. Le nombre de threads affecté à une opération Cholesky est ensuite automatiquement adapté aux ressources disponibles. Par exemple, 4 opérations en parallèle disposeront de 4 threads chacune sur la machine Kwak. La région parallèle OpenMP est exécutée autant de fois que nécessaire pour atteindre le nombre de 16 opérations Cholesky effectuées. Nous faisons ainsi varier le nombre de threads affecté au parallélisme OpenMP, externe, et celui affecté au parallélisme `pthread`, interne. Les performances des différentes combinaisons sont représentées dans le tableau 5.13.

Opérations concurrentes × Threads par opération	Performances (GFlops)
1×16	79,87
2×8	82,14
4×4	93,97
8×2	88,81
16×1	87,12

TAB. 5.13 – Performances obtenues par 16 opérations Cholesky exécutées par une application OpenMP sur la machine Kwak, en fonction du nombre d'opérations exécutées en parallèle et du nombre de threads affecté à chaque opération.

Chapitre 6

Conclusion et perspectives

Ces dernières années ont vu se développer les architectures à base de processeurs multicœurs sur les marchés scientifique et grand public. Les constructeurs de processeurs, premiers partisans de cette révolution, proposent aujourd'hui des configurations à plusieurs puces multicœurs accédant à des bancs mémoire répartis selon le modèle des architectures à accès mémoire non-uniformes. Les cœurs d'une même puce accèdent à différents niveaux de cache, dont certains sont partagés entre plusieurs cœurs. Ces caractéristiques de conception rendent la topologie des architectures multicœurs fortement hiérarchique.

La plupart des langages de programmation parallèle utilisés pour les programmer ont été développés pour tirer parti des architectures multiprocesseurs symétriques. Ils exposent ainsi un parallélisme unidimensionnel qui peine à s'adapter au relief des architectures multicœurs. Leur exploitation efficace passe en effet par le respect des affinités entre traitements d'une part pour tirer parti du partage de cache entre plusieurs cœurs et des affinités mémoire d'autre part pour limiter le taux d'accès mémoire distants et la contention sur les différents bus. De plus, l'augmentation constante du nombre de cœurs par puce oblige le programmeur d'applications parallèles à exprimer de plus en plus de parallélisme.

Le langage OpenMP reste parmi les langages de programmation parallèles les plus répandus dans la communauté scientifique. L'arrivée des processeurs multicœurs sur le marché grand public amène des programmeurs non-spécialistes à s'intéresser au parallélisme, et la portabilité et la simplicité d'utilisation offertes par OpenMP participent à l'expansion de sa popularité. Ce langage semble de plus le mieux placé pour tirer parti des architectures multicœurs : sa capacité à imbriquer des régions parallèles lui permet d'exprimer un parallélisme structuré qui s'adapte naturellement aux topologies hiérarchiques. Pourtant, les performances obtenues par OpenMP sur les architectures hiérarchiques sont souvent décevantes. La structure du parallélisme n'est en effet pas transmise au support exécutif.

Contributions de la thèse

Nous avons proposé ForestGOMP, un support exécutif pour OpenMP qui capture la structure du parallélisme en regroupant les threads issus d'une même région parallèle à l'intérieur d'une même bulle de la bibliothèque BubbleSched. Ces bulles, visibles par l'ordonnanceur de ForestGOMP, traduisent des relations d'affinité entre les threads qu'elles contiennent que l'ordonnanceur peut consulter *tout au long de l'exécution de l'application*.

Deux ordonnanceurs ont été développés pour répartir les bulles issues de régions parallèles OpenMP sur les architectures multicœurs hiérarchiques. Le premier, *Cache*, ordonnance les bulles de façon compacte sur la machine afin de maximiser l'utilisation des différents niveaux de cache d'un processeur multicœur, et s'avère particulièrement performant pour orchestrer l'exécution d'algorithmes de type *diviser pour régner*, comme on a pu le voir avec l'application MPU. Le second, *Memory*, effectue un placement conjoint des threads et des données de l'application, dans le but de respecter les affinités mémoire exprimées au niveau applicatif à l'aide d'une interface de programmation. En particulier, Memory cherche à limiter le taux d'accès mémoire distants sur les architectures à accès mémoire non-uniformes, de manière à minimiser la contention sur les différents bus mémoire de la machine.

Ces ordonnanceurs sont instantiables sur une sous-partie de la topologie, ce qui facilite leur composition. Nous avons ainsi montré le bon comportement d'une composition des ordonnanceurs Cache et Memory sur la déclinaison de benchmarks STREAM, Memory étant en charge de la distribution des bulles et de leurs données entre les nœuds NUMA de la machine, Cache opérant lui à l'intérieur d'un nœud.

Nous avons aussi montré que le confinement des threads ForestGOMP sur une partie de machine facilitait la composition du parallélisme issu de ForestGOMP avec d'autres bibliothèques parallèles, comme la bibliothèque d'algèbre linéaire GotoBLAS. Alors que l'exécution concurrente de plusieurs applications OpenMP est obtenue grâce à un superviseur arbitrant l'accès aux ressources, la composition d'OpenMP et de GotoBLAS a nécessité l'adaptation de cette bibliothèque au contexte multiprogrammé : tout comme ForestGOMP, nous l'avons rendu instanciable et confinable. Les performances obtenues sur la décomposition Cholesky confirment le rôle prépondérant de la composition de parallélisme dans l'exploitation efficace des architectures multicœurs de demain.

Perspectives

Ces travaux ouvrent de nombreuses perspectives.

Vers une meilleure intégration avec le compilateur

Nous envisageons dans un premier temps d'améliorer l'interaction entre notre support exécutif et le compilateur. Dans l'état, la plateforme ForestGOMP, bien qu'implémentée comme une extension de GNU OpenMP, constitue un support d'exécution générique

qui pourrait être utilisé comme cible par n'importe quel compilateur. Toutefois, nous pouvons enrichir l'API de notre support exécutif afin par exemple de transmettre des informations spécifiques au compilateur, pour guider la répartition des données ou lui indiquer la quantité de données globales partagées, ou encore de générer des traces d'exécution exploitables par les outils de visualisation, de profiling et de débogage.

Nous avons également des idées concernant certaines extensions au standard OpenMP pour mieux appréhender les nouvelles architectures multicœurs. Notre expérience en matière d'ordonnancement de régions parallèles imbriquées nous porte en effet à penser que les extensions actuellement proposées pour résoudre les problèmes de placement de threads/tâches sur les cœurs ne vont pas dans la bonne direction. De notre point de vue, il donnent au programmeur des outils beaucoup trop explicites, voire bas niveau, pour dimensionner le parallélisme d'une part, avec la définition du nombre de threads par région parallèle, et projeter le parallélisme sur la topologie sous-jacente d'autre part, avec la proposition pour une prochaine version du standard de mots-clés permettant de placer explicitement un thread sur un cœur en faisant référence à son numéro. Nous pensons qu'il est possible de définir des clauses d'ordonnancement dans des directives OpenMP qui donnent davantage de souplesse au support exécutif pour effectuer les projections, tout en permettant au programmeur d'exprimer des besoins particuliers. On peut par exemple imaginer de proposer au programmeur des mots clés pour qualifier le comportement, ou plus précisément les besoins, d'une région parallèle, qui feront intervenir des politiques d'ordonnancement différentes au sein du support exécutif, offrant ainsi une projection de parallélisme adaptée à la topologie de façon portable. L'inconvénient de cette approche reste la difficulté pour le programmeur de prendre connaissance et de comprendre ensuite les décisions prises par le support exécutif. Ce point rejoint la perspective précédente sur les liens à affectuer entre le programme source et les événements collectés au niveau du support exécutif ou même du matériel.

Suivre les évolutions matérielles

Jusqu'à présent, notre implémentation a plutôt bien montré son adéquation aux architectures multicœurs contemporaines. Mais, dans un avenir assez proche, ces architectures vont encore subir de profonds changements. En particulier, les puces multicœurs contenant des processeurs organisés en grilles, avec une topologie rappelant les *Transputers*, semblent une piste sérieuse pour le fabricant Intel. Par ailleurs, IBM et Intel ont récemment fait un pas en direction d'architectures renonçant à la cohérence de cache matérielle : les processeurs qui les composent disposent d'une mémoire locale non cohérente avec les autres (IBM, Intel), ni même cohérente avec la mémoire globale (IBM). Ces évolutions ont des conséquences importantes pour l'architecture de ForestGOMP. L'organisation en grille nécessite probablement de revoir la perception de la topologie des machines comme une grille en deux dimensions, et non comme un arbre 1D. Les algorithmes de projection d'arbres 1D vers des grilles 2D, déjà beaucoup explorés dans la littérature, nécessitent d'être revus pour imaginer des versions dynamiques capables de s'approcher de l'optimal sans vision globale de l'état du système. L'absence de cohérence de cache pose, d'une certaine façon, moins de problème en ce qui concerne

l'évaluation de coût d'explosion des bulles, la pénalités étant, en quelque sorte, très liées au nombre de synchronisations explicites de la mémoire. Toutefois, l'ordonnanceur ForestGOMP lui-même nécessite d'être écrit de façon complètement distribuée, un effort s'imposant de lui-même pour les futures architectures many-core.

Demain, des composants OpenMP off-the-shelf?

Dans un futur proche, il s'agira véritablement pour le programmeur de pouvoir composer des codes parallèles sans se soucier de la façon dont ils sont implémentés. Ainsi, le problème dépasse le cadre du langage OpenMP : il faudra pouvoir composer des codes écrits en OpenMP, en Intel TBB ou même directement en pthread. Il y a donc des efforts à fournir pour converger vers un support d'exécution unifié, capable de supporter aisément la plupart des paradigmes utilisés par ces outils, comme les threads noyau, les threads utilisateur, les tâches, les continuations et même les processus MPI multithreadés, en virtualisant véritablement les ressources. La communauté scientifique a longtemps ignoré ces problèmes, et peu de travaux proposent des solutions. *Lithe* [PHA10] est un effort remarquable de support d'exécution permettant de confiner l'exécution de codes parallèles hétérogènes, mais n'offre toutefois pas de solution pour aider à dimensionner dynamiquement le parallélisme exhibé par chacun de ces codes. Une thèse devrait débuter dans l'équipe *Runtime* en septembre prochain sur ce thème.

Annexe A

Interface de programmation de ForestGOMP

Sommaire

A.1 Variables globales et structures de données 97
A.2 Primitives de gestion de données et d'affinités mémoire 98
A.3 Primitives pour définir des charges de travail 98
A.4 Primitives de débogage 99

Cette annexe liste les fonctionnalités de ForestGOMP mises à disposition du programmeur sous forme d'une interface de programmation.

A.1 Variables globales et structures de données

```
/* The ForestGOMP memory manager, needed by MaMI. */
extern mami_manager_t *fgomp_memory_manager;

/* Structure to describe a memory area attached to a gomp thread. */
struct fgomp_memory_area
{
  /* Beginning of the memory area. */
  void *buffer;

  /* The size of the memory area */
  size_t chunk_size;
};

/* Variables to specify the behavior of the BubbleSched scheduler
 * upon memory affinity updates:
 *    - WAIT to wait until all the threads of the team have reached
 *      the update call to perform a global thread distribution,
 *    - NOWAIT to make every thread perform its own distribution */
```

```
typedef enum fgomp_shake_mode
{
  FGOMP_SHAKE_WAIT,
  FGOMP_SHAKE_NOWAIT,
  FGOMP_DONT_SHAKE
} fgomp_shake_mode_t;
```

A.2 Primitives de gestion de données et d'affinités mémoire

```
/* Allocates size_t bytes inside the fgomp memory manager. */
extern void *fgomp_malloc (size_t);

/* Allocates size_t bytes on a specific node. */
extern void *fgomp_malloc_on_node (size_t, unsigned int);

/* Mark the pages holding the buffer passed in argument
 * to be migrated the next time a thread access them. */
extern int fgomp_migrate_on_next_touch (void *, size_t);

/* Mark the pages holding the buffer passed in argument
 * to be attached to the next thread to access them. */
extern int fgomp_attach_on_next_touch (void *, size_t);

/* Specify that the upcoming parallel section will create
 * threads working on "buffer". */
extern int fgomp_set_next_team_affinity (void *, size_t,
                                         fgomp_shake_mode_t);

/* Declare a buffer to be accessed by the current thread. */
extern int fgomp_set_current_thread_affinity (void *,
                                              size_t,
                                              fgomp_shake_mode_t);

/* Free a memory area allocated using
 * fgomp_malloc/malloc_on_node. */
extern void fgomp_free (void *);
```

A.3 Primitives pour définir des charges de travail

```
/* Set the current ForestGOMP thread load. This helps the scheduler
 * estimating how long the execution of the current thread job will
```

```
 * last. */
extern void fgomp_set_current_thread_load (unsigned int);

/* Set the number of threads the caller will create the next time it
 * goes parallel. */
extern void fgomp_set_num_threads (int);

/* Get the number of threads set by fgomp_set_num_threads (). If the
 * caller never called fgomp_set_num_threads (), return
 * omp_get_num_threads (). */
extern int fgomp_get_num_threads (void);
```

A.4 Primitives de débogage

```
/* Return the location (the core id) of the calling thread. */
extern int fgomp_get_location (void);

/* Print the runqueue the caller is scheduled on. */
extern void fgomp_say_hi (const char *msg);
```

Annexe B

Instrumentation du benchmark Nested-STREAM

Sommaire

B.1 Allocation et libération de zones mémoire 101
B.2 Attachement des zones mémoire aux équipes OpenMP 102

Cette annexe illustre l'instrumentation du benchmark synthétique Nested-STREAM à l'aide des fonctions d'attachement mémoire fournies par ForestGOMP.

B.1 Allocation et libération de zones mémoire

```
double **a, **b, **c;

struct fgomp_memory_area my_a[NB_NUMA_NODES];
struct fgomp_memory_area my_b[NB_NUMA_NODES];
struct fgomp_memory_area my_c[NB_NUMA_NODES];

a = fgomp_malloc (NB_NUMA_NODES * sizeof (double *));
b = fgomp_malloc (NB_NUMA_NODES * sizeof (double *));
c = fgomp_malloc (NB_NUMA_NODES * sizeof (double *));

/* Initial data distribution */
for (i = 0; i < NB_NUMA_NODES; i++) {

  /* Allocate the STREAM arrays */
  a[i] = fgomp_malloc_on_node (TAB_SIZE, i);
  b[i] = fgomp_malloc_on_node (TAB_SIZE, i);
  c[i] = fgomp_malloc_on_node (TAB_SIZE, i);

  /* Link the memory areas to the STREAM arrays */
  my_a[i].buffer = a[i];
```

```
    my_b[i].buffer = b[i];
    my_c[i].buffer = c[i];

    /* Set the memory areas length */
    my_a[i].buffer_len = N;
    my_b[i].buffer_len = N;
    my_c[i].buffer_len = N;

    /* Set the length of each chunk, a chunk being
     * the amount of memory accessed by a single thread */
    my_a[i].chunk_size = N / NB_THREADS_PER_TEAMS;
    my_b[i].chunk_size = N / NB_THREADS_PER_TEAMS;
    my_c[i].chunk_size = N / NB_THREADS_PER_TEAMS;
  }

  ...

  /* Free the memory areas */
  for (i = 0; i < NB_NUMA_NODES; i++) {
    fgomp_free (a[i]);
    fgomp_free (b[i]);
    fgomp_free (c[i]);
  }

  fgomp_free (a);
  fgomp_free (b);
  fgomp_free (c);
```

B.2 Attachement des zones mémoire aux équipes OpenMP

```
  /* STREAM's Copy operation */
#pragma omp parallel for num_threads (NB_NUMA_NODES)
  for (i = 0; i < NB_NUMA_NODES; i++) {
    fgomp_set_next_team_affinity (FGOMP_SHAKE_WAIT,
                                  &my_c[i],
                                  &my_a[i],
                                  NULL);
#pragma omp parallel for num_threads (NB_THREADS_PER_TEAMS)
    for (j = 0; j < N; j++)
      c[i][j] = a[i][j];
  }

  /* STREAM's Scale operation */
#pragma omp parallel for num_threads (NB_NUMA_NODES)
  for (i = 0; i < NB_NUMA_NODES; i++) {
```

```
    fgomp_set_next_team_affinity (FGOMP_SHAKE_WAIT,
                                  &my_b[i],
                                  &my_c[i],
                                  NULL);
#pragma omp parallel for num_threads (NB_THREADS_PER_TEAMS)
    for (j = 0; j < N; j++)
      b[i][j] = scalar * c[i][j];
  }




  /* STREAM's Add operation */
#pragma omp parallel for num_threads (NB_NUMA_NODES)
  for (i = 0; i < NB_NUMA_NODES; i++) {
    fgomp_set_next_team_affinity (FGOMP_SHAKE_WAIT,
                                  &my_a[i],
                                  &my_b[i],
                                  &my_c[i],
                                  NULL);
#pragma omp parallel for num_threads (NB_THREADS_PER_TEAMS)
    for (j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
  }

  /* STREAM's Triad operation */
#pragma omp parallel for num_threads (NB_NUMA_NODES)
  for (i = 0; i < NB_NUMA_NODES; i++) {
    fgomp_set_next_team_affinity (FGOMP_SHAKE_WAIT,
                                  &my_a[i],
                                  &my_b[i],
                                  &my_c[i],
                                  NULL);
#pragma omp parallel for num_threads (NB_THREADS_PER_TEAMS)
  for (j = 0; j < N; j++)
    a[i][j] = b[i][j] + scalar * c[i][j];
  }
```

Bibliographie

[ACC+90] Alverson (R.), Callahan (D.), Cummings (D.), Koblenz (B.), Porterfield (A.) et smith (B.), « The tera computer system », 1990.

[ACD+09] Ayguadé (E.), Copty (N.), Duran (A.), Hoeflinger (J.), Lin (Y.), Massaioli (F.), Teruel (X.), Unnikrishnan (P.) et Zhang (G.), « The design of openmp tasks », *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, nº 3, 2009, p. 404–418.

[BBB+91] Bailey (D. H.), Barszcz (E.), Barton (J. T.), Browning (D. S.), Carter (R. L.), Dagum (D.), Fatoohi (R. A.), Frederickson (P. O.), Lasinski (T. A.), Schreiber (R. S.), Simon (H. D.), Venkatakrishnan (V.) et Weeratunga (S. K.), « The NAS Parallel Benchmarks », *The International Journal of Supercomputer Applications*, vol. 5, nº 3, Fall 1991, p. 63–73.

[BCFP+09] Bastos Castro (M.), Fernandes (L. G.), Pousa (C.), Méhaut (J.-F.) et Aguiar (M.), « NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines », dans *Proceedings of the 10th International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*, Roma, Italy, may 2009.

[BEKK00] Borkenhagen (J.), Eickemeyer (R.), Kalla (R.) et Kunkel (S.), « A multithreaded powerpc processor for commercial servers », dans *IBM J. Res. Dev. vol. 44*, p. 885–898, 2000.

[Ber96] Berg (D. J.), « Java threads - a white paper », 1996.

[BKP93] Bodin, Kervella et Priol, « Fortran-s: A fortran interface for shared virtual memory architectures », *SC Conference*, vol. 0, 1993, p. 274–283.

[BOM+10] Broquedis (F.), Clet Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.), « hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications », dans IEEE, éditeur, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italie, 2010.

[Bul99] Bull (J. M.), « Measuring synchronisation and scheduling overheads in openmp », 1999.

[CDC+99] Carlson (W.), Draper (J.), Culler (D.), Yelick (K.), Brooks (E.) et Warren (K.), « Introduction to UPC and Language Specification ». Rapport technique nº CCS-TR-99-157, George Mason University, mai 1999.

[CHJ+06] Chapman (B. M.), Huang (L.), Jin (H.), Jost (G.) et Supinski (B. R. D.), « Toward enhancing openmp's work-sharing directives », dans *in the Euro-Par06 Conference*, p. 645–654, 2006.

[DHP10] Dimakopoulos (V.), Hadjidoukas (P.) et Philos (G.), « A microbenchmark study of openmp overheads under nested parallelism », dans Eigenmann (R.) et de Supinski (B.), éditeurs, *OpenMP in a New Era of Parallelism*, vol. 5004 (coll. *Lecture Notes in Computer Science*), p. 1–12. Springer Berlin / Heidelberg, 2010.

[DM05] Drepper (U.) et Molnar (I.), « The native posix thread library for linux ». Rapport technique, 2005.

[Dre04] Drepper (U.), « Futexes Are Tricky », avril 2004. `http://people.redhat.com/drepper/futex.pdf`.

[EEL+97] Eggers (S. J.), Emer (J. S.), Levy (H. M.), Lo (J. L.), Stamm (R. L.) et Tullsen (D. M.), « Simultaneous multithreading: A platform for next-generation processors », *IEEE Micro*, vol. 17, 1997, p. 12–19.

[FLPR99] Frigo (M.), Leiserson (C. E.), Prokop (H.) et Ramachandran (S.), « Cache-oblivious algorithms », dans *In 40th Annual Symposium on Foundations of Computer Science*, p. 285–297, 1999.

[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.

[for] « Site internet de forestgomp. ». `http://http://runtime.bordeaux.inria.fr/forestgomp/`.

[FR] Fluet (M.) et Rainey (M.), « A scheduling framework for general-purpose parallel languages », dans *In Proc. of the Int. Conf. on Funct. Program*, p. 241–252. ACM.

[gfo] « Forestgomp sur la forge inria. ». `https://gforge.inria.fr/projects/forestgomp/`.

[GVDG08] Goto (K.) et Van De Geijn (R.), « High-performance implementation of the level-3 blas », *ACM Trans. Math. Softw.*, vol. 35, n° 1, 2008, p. 1–14.

[HD07] Hadjidoukas (P. E.) et Dimakopoulos (V. V.), « Nested parallelism in the ompi openmp/c compiler », 2007.

[Hoe06] Hoeflinger (J.), « Extending OpenMP to clusters », *White Paper, Intel Corporation*, 2006.

[HP03] Hennessy (J.) et Patterson (D.), *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[HSU+] Hinton (G.), Sager (D.), Upton (M.), Boggs (D.), Carmean (D.), Kyker (A.) et Roussel (P.), « The microarchitecture of the pentium 4 processor », *Intel Technology Journal*, n° 1, p. 2001.

[IBM] IBM, « Next generation posix threading. ». `http://oss.software.ibm.com/developerworks/oss/pthreads`.

[KMAC03] Keltcher (C. N.), McGrath (K. J.), Ahmed (A.) et Conway (P.), « The amd opteron processor for multiprocessor servers », *IEEE Micro*, n° 23, 2003, p. 66–76.

[McC07] McCalpin (J. D.), « Stream: Sustainable memory bandwidth in high performance computers ». Rapport technique, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[MCS91] Mellor-Crummey (J.) et Scott (M.), « Algorithms for scalable synchronization on shared-memory multiprocessors », *ACM Transactions on Computer Systems (TOCS)*, vol. 9, nº 1, 1991, p. 65.

[mkl] « The Intel Math Kernel Library ». `http://software.intel.com/en-us/intel-mkl/`.

[MLV+04] 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 Kosch (H.), Böszörményi (L.) et Hellwagner (H.), éditeurs, *Euro-Par 2003 Parallel Processing*, vol. 2790 (coll. *Lecture Notes in Computer Science*), p. 1291–1294. Springer Berlin / Heidelberg, 2004.

[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.

[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.

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

[PCMC10] POUSA (C. R.), CASTRO (M.), Méhaut (J.-F.) et Carissimi (A.), « Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas », dans *9th International Meeting High Performance Computing for Computational Science (VECPAR)*, 2010.

[PHA10] Pan (H.), Hindman (B.) et Asanović (K.), « Composing parallel software efficiently with lithe », *SIGPLAN Not.*, vol. 45, nº 6, 2010, p. 376–387.

[PW96] Peleg (A.) et Weiser (U.), « Mmx technology extension to the intel architecture », *IEEE Micro*, nº 16, 1996, p. 42–50.

[SAtH+06] Saha (B.), reza Adl-tabatabai (A.), Hudson (R. L.), Minh (C. C.) et Hertzberg (B.), « Mcrt-stm: a high performance software transactional memory system for a multi-core runtime », dans *In Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, p. 187–197. ACM Press, 2006.

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

[SGDM94] Sunderam (V.), Geist (G.), Dongarra (J.) et Manchek (R.), « The PVM concurrent computing system: Evolution, experiences, and trends », *Parallel computing*, vol. 20, nº 4, 1994, p. 531–545.

[Smi81] Smith (B. J.), « Architecture and applications of the hep multiprocessor computer system », dans *SPIE*, p. 241–248, 1981.

[SO98] Snir (M.) et Otto (S.), *MPI-The Complete Reference: The MPI Core*. MIT Press Cambridge, MA, USA, 1998.

[SRM00] Schlansker (M. S.), Rau (B. R.) et Multitemplate, « Epic: An architecture for instruction-level parallel processors ». Rapport technique, 2000.

[STG+02] Su (E.), Tian (X.), Girkar (M.), Haad (G.), Shah (S.) et Petersen (P.), « Compiler support of the workqueuing execution model for intel smp architectures », 2002.

[STMB10] Schmidl (D.), Terboven (C.), an Mey (D.) et Bücker (M.), « Binding nested openmp programs on hierarchical memory architectures », vol. 6132, 2010, p. 29–42.

[TBB] « Version Open Source d'Intel TBB ». `http://www.threadingbuildingblocks.org/`.

[Thi07] Thibault (S.), *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2007. 128 pages.

[Ver] Version (S.), « Maspar programming language (ansi c compatible mpl) user guide ».

[WCO+08] Williams (S.), Carter (J.), Oliker (L.), Shalf (J.) et Yelick (K.), « Lattice boltzmann simulation optimization on leading multicore platforms », dans *in Interational Conference on Parallel and Distributed Computing Systems (IPDPS*, 2008.

Liste des publications

Revues internationales avec comité de lecture

[BFG+10] Broquedis (F.), Furmento (N.), Goglin (B.), Wacrenier (P.-A.) et Namyst (R.), « ForestGOMP: an efficient OpenMP environment for NUMA architectures », *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Müller and Eduard Ayguadé*, vol. 38, nº 5, 2010, p. 418–439.

Conférences internationales avec publication des actes et comité de lecture

[BAG+10] Broquedis (F.), Aumage (O.), Goglin (B.), Thibault (S.), Wacrenier (P.-A.) et Namyst (R.), « Structuring the execution of OpenMP applications for multicore architectures », dans *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, avril 2010. IEEE Computer Society Press.

[BCOM+10] Broquedis (F.), Clet-Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.), « hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications », dans *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, p. 180–186, Pisa, Italia, février 2010. IEEE Computer Society Press.

[BDT+08] Broquedis (F.), Diakhaté (F.), Thibault (S.), Aumage (O.), Namyst (R.) et Wacrenier (P.-A.), « Scheduling Dynamic OpenMP Applications over Multicore Architectures », dans *OpenMP in a New Era of Parallelism, 4th International Workshop on OpenMP, IWOMP 2008*, vol. 5004 (coll. *Lecture Notes in Computer Science*), p. 170–180, West Lafayette, IN, mai 2008. Springer.

[BFG+09] Broquedis (F.), Furmento (N.), Goglin (B.), Namyst (R.) et Wacrenier (P.-A.), « Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective », dans Matthias S. Muller (B. C. Bronis R. de Supinski), éditeur, *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, IWOMP 2009*, vol. 5568 (coll. *Lecture Notes in Computer Science*), p. 79–92, Dresden, Germany, juin 2009. Springer.

[TBG+08] Thibault (S.), Broquedis (F.), Goglin (B.), Namyst (R.) et Wacrenier (P.-A.), « An Efficient OpenMP Runtime System for Hierarchical Architectures »,

dans Chapman (B. M.), Zheng (W.), Gao (G. R.), Sato (M.), Ayguadé (E.) et Wang (D.), éditeurs, *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings*, vol. 4935 (coll. *Lecture Notes in Computer Science*), p. 161–172. Springer, 2008.

Conférences nationales avec publication des actes et comité de lecture

[Bro08] Broquedis (F.), « Exécution structurée d'applications OpenMP á grain fin sur architectures multicoeurs », dans *18èmes Rencontres Francophones du Parallélisme*, Fribourg / Suisse, février 2008. École d'ingénieurs et d'architectes de Fribourg.

[Bro09] Broquedis (F.), « Ordonnancement de threads OpenMP et placement de données coordonnés sur architectures hiérarchiques », dans *19èmes Rencontres Francophones du Parallélisme*, Toulouse / France, septembre 2009.

Rapports de recherche

[Bro07] Broquedis (F.), « De l'exécution structurée de programmes openmp sur architectures hiérarchiques ». Mémoire de dea, Université Bordeaux 1, juin 2007.

-=-=-=-