Exécution structurée d'applications OpenMP à grain fin sur architectures multicoeurs

François Broquedis

RenPar'18 / SympA'2008 / CFSE'6
Fribourg, 11 au 13 février 2008

Exécution structurée d'applications OpenMP à grain fin sur architectures multicœurs

François Broquedis

Projet INRIA RUNTIME, LaBRI, Université Bordeaux 1,
351, cours de la Libération, 33405 TALENCE - France

Résumé
Les architectures multiprocesseurs contemporaines, qui se font naturellement l'écho de l'évolution actuelle des microprocesseurs vers des puces massivement multicœur, exhibent un parallélisme de plus en plus hiérarchique. Pour s'approcher des performances théoriques de ces machines, il faut désormais extraire un parallélisme de plus en plus fin des applications, mais surtout communiquer sa structure — et si possible des directives d'ordonnancement — au support d'exécution sous-jacent [18].
Dans cet article, nous expliquons pourquoi OpenMP est un excellent vecteur pour extraire des applications du parallélisme massif, structuré et annoté et nous montrons comment, au moyen d'une extension du compilateur GNU OpenMP s'appuyant sur un ordonnanceur de threads *NUMA-aware*, il est possible d'exécuter efficacement des applications dynamiques et irrégulières en préservant l'affinité des threads et des données.

Mots-clés : OpenMP, multicore, NUMA, thread, runtime

---

1. Introduction

L'avènement de l'ère multicœur et par incidence des machines fortement hiérachisées a rendu plus évident encore l'apport des approches multi-niveaux et plus généralement de celui du parallélisme imbriqué. Pour exploiter toute la puissance agrégée de ces nombreux cœurs, le programmeur se doit dorénavant d'exhiber un parallélisme à un grain très fin tout en portant une attention particulière à la répartition des traitements et des données auxquelles ceux-ci accèdent, afin de maximiser l'utilisation du cache et de minimiser les accès mémoire distants. Ainsi, il n'est plus possible d'obtenir des performances optimales en exploitant un parallélisme naïf reposant sur la génération de tâches « à plat », et de récents travaux [19] ont d'ailleurs montré l'importance de *structurer* le parallélisme afin d'assurer la répartition rationnelle des traitements sur la machine.
Dans cet article, nous montrons pourquoi OpenMP, par ses directives parallèles structurantes et sa simplicité d'utilisation, constitue un excellent outil dans ce registre à condition de lui adjoindre des politiques d'ordonnancement adaptées. En effet, en adoptant une approche haut niveau le programmeur peut avoir la sensation de perdre le contrôle de l'exécution de l'application, notamment lorsque les performances obtenues lui semblent décevantes. Par exemple, alors que de nombreux articles [16, 4, 7, 9, 2] démontrent la nécessité d'employer une approche parallèle multi-niveau pour passer à l'échelle, sa prise en charge par les environnements OpenMP a toujours mauvaise presse [3] : « *The overhead associated with the creation of parallel regions, the varying levels of support in different implementations [...] and the impossibility of controlling load balacing, make this approach impractical* ».
Nous proposons ici une politique d'ordonnancement conciliant la structure parallèle du programme OpenMP et celle de la machine sous-jacente qui assure l'exécution efficace d'applications irrégulières à

grain très fin sur des architectures hiérarchiques. Nous validons notre approche en comparant les performances de différentes implémentations d'une application de reconstruction de surface implicite baptisée MPU [14] au caractère très irrégulier pouvant générer des dizaines de milliers de threads par seconde.
Ce travail s'appuie sur la plateforme OpenMP FORESTGOMP [17] et la bibliothèque de threads BUBBLESCHED [18, 19] qui facilite le dialogue entre l'application et l'ordonnanceur afin de structurer l'exécution des programmes parallèles.

2. De la difficulté d'ordonnancer les applications OpenMP irrégulières

Ces dernières années la communauté OpenMP a produit de nombreux articles démontrant que le parallélisme imbriqué favorisait la scalabilité de nombreuses applications et proposant des techniques facilitant son expression ou son exploitation. Aussi la plupart des compilateurs OpenMP récents supportent l'imbrication de sections parallèles en s'appuyant sur des bibliothèques de threads de niveau utilisateur efficaces (NANOS Nthlib [10], Omni/ST [16], OMPi [11]) ou sur un pool de threads prédéfini pour l'exécution des tâches parallèles imbriquées, afin d'éviter les créations/destructions coûteuses (Sun Studio [1], Intel Compiler [21, 20], OdinMP [13]).
Par exemple, Omni/ST s'appuie sur une bibliothèque de threads qui utilise un nombre fixé de threads pour exécuter un nombre arbitraire de *filaments*, de façon similaire au langage Cilk [8]. Les performances obtenues sur architecture multiprocesseur symétrique sont souvent très bonnes, puisque beaucoup de tâches peuvent être exécutées de façon séquentielle sans surcoût quand tous les processeurs sont occupés.
Pour tenir compte du caractère fortement hiérarchique des nouvelles architectures, le compilateur OMPi utilise des threads de niveau utilisateur non préemptibles et ordonnancés de la façon suivante : les threads générés par le premier niveau sont distribués cycliquement et ajoutés en queue de liste ; les threads issus des niveaux imbriqués sont ajoutés en tête de la liste correspondant au processeur qui les a créés. Pour assurer la localité des données, un processeur inactif pioche un thread à exécuter en tête de sa propre liste, et lorsque elle est vide, en queue des listes des autres processeurs.
Cependant, ni Omni/ST ni OMPi ne permettent d'attacher aux tâches générées des informations de haut niveau comme l'affinité mémoire. Leurs supports exécutifs sont donc incapables de déterminer quels threads travaillent ensemble, le vol de tâche revêt donc un caractère aléatoire séparant des ensembles cohérents de threads, pénalisant ainsi leur exécution sur une architecture hiérarchique.
Par ailleurs, plusieurs extensions du langage OpenMP ont été proposées pour contrôler le placement des threads sur la machine. Un mécanisme présenté dans [10] permet par exemple de «*fixer*» les threads sur les processeurs, ce qui permet d'optimiser un programme pour une machine donnée. Le compilateur NANOS [7, 4] permet lui d'affecter de façon *statique* des régions parallèles à des groupes de threads.
Enfin, les compilateurs KAI/Intel [15] et NANOS Mercurium [5] supportent le parallélisme de tâche, et une proposition d'introduction de tâches parallèles dans OpenMP 3.0 a été présentée [3]. On peut cependant noter dans les perspectives de cet article que les auteurs relèvent la nécessité de développer des politiques complexes d'ordonnancement de tâches prenant en compte des caractéristiques de l'application, ou encore le besoin d'étendre le standard OpenMP afin de donner plus de contrôle au programmeur sur l'ordonnancement de son application. C'est justement dans cette optique que se situent nos travaux.

3. Une plateforme OpenMP pour le développement et l'optimisation de politiques d'ordonnancement NUMA-aware

Comme nous venons de le remarquer, les supports exécutifs actuels manquent d'information pour ordonnancer efficacement des groupes de threads sur les architectures hiérarchiques. Le programmeur connait assez bien son programme pour indiquer quelles sections de code parallèle contiennent des données partagées accédées fréquemment. Nous défendons l'idée que ces informations, couplées à une modélisation

adaptée des architectures hiérarchiques, devraient être transmises aux supports d'exécution afin d'ordonnancer efficacement des groupes de threads qui travaillent intensivement ensemble. Par exemple, il arrive souvent que les threads issus d'une même section parallèle partagent des variables. C'est une information qui pourrait être transmise au support d'exécution, pour qu'il prenne des décisions adéquates concernant le placement de ces threads par rapport à la localisation en mémoire de ces variables.
La plateforme FORESTGOMP a été développée dans ce but. Elle propose une extension du *runtime* GNU OpenMP GOMP qui permet de s'appuyer sur la bibliothèque MARCEL BUBBLESCHED. Cette bibliothèque permet de regrouper des threads dans des structures de bulles, auxquelles de nombreuses informations peuvent être attachées. Elle constitue de plus une véritable plateforme de création d'ordonnanceurs spécifiques pour la répartition efficace de ces bulles en tenant compte de ces informations.

3.1. Des bulles pour guider l'ordonnancement

La plateforme BUBBLESCHED [19] repose sur la bibliothèque de threads MARCEL. Elle permet l'écriture d'ordonnanceurs spécifiques, efficaces sur architectures hiérarchiques, grâce à deux mécanismes principaux : le regroupement de threads en relation et la modélisation des architectures hiérarchiques.
Son API fournit ainsi des fonctions permettant de regrouper des threads dans une structure récursive appelée *bulle*. Cette structuration permet au programmeur d'exprimer un certain nombre de relations entre threads, comme le partage de données, ou encore l'exécution fréquente d'opérations collectives, traduisant plus généralement des besoins spécifiques en ordonnancement.
Les architectures hiérarchiques sont modélisées dans BUBBLESCHED par une hiérarchie de listes de threads, grâce à une analyse topologique dynamique et portable. Chaque entité matérielle dispose ainsi de sa propre liste, des processeurs logiques SMT jusqu'à la machine toute entière. La liste sur laquelle un thread est placé exprime ainsi son domaine d'ordonnancement : si le thread est placé sur une liste représentant une puce physique, il sera ordonnancé par les processeurs de cette puce seulement ; s'il est placé sur la liste de la machine entière, il pourra être ordonnancé par n'importe quel processeur de la machine. L'ordonnanceur de threads MARCEL sous-jacent exécute les entités depuis les listes les plus spécifiques, pour ensuite remonter vers les plus générales au besoin.
Ainsi, écrire un ordonnanceur à bulles se résume à écrire un certain nombre de fonctions clés appelées à certains moments de l'exécution d'un programme. Une de ces fonctions sera par exemple appelée afin d'effectuer une distribution initiale des threads et bulles sur les différentes listes, une autre permettra de rééquilibrer la charge lorsqu'un processeur devient inactif.

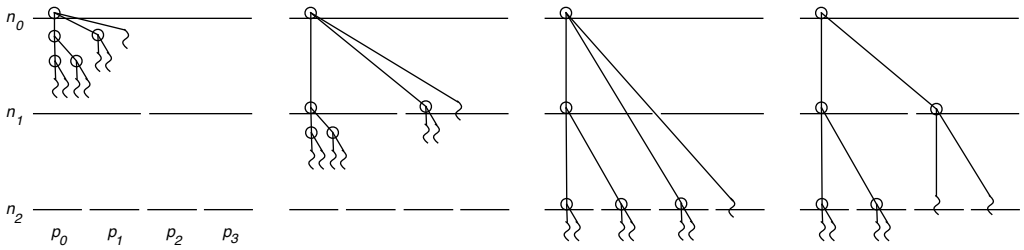
3.2. Génération et ordonnancement de bulles dans les programmes OpenMP

La plateforme FORESTGOMP est une extension du support d'exécution GNU OpenMP. Nous avons modifié la bibliothèque libgomp afin de remplacer les appels à la bibliothèque PTHREAD par leurs équivalents MARCEL. Nous avons de plus ajouté la génération de bulles MARCEL, selon ce schéma :
À l'entrée de chaque section parallèle OpenMP, le thread principal créé une bulle dans laquelle il se place lui-même. Il créé ensuite ses « coéquipiers » dans cette même bulle. À la fin de chaque section parallèle, le thread principal est extrait de sa bulle pour rejoindre son équipe d'origine, et cette bulle est ensuite détruite. Ainsi, les algorithmes de type « Diviser pour Régner » engendreront naturellement de nombreuses bulles de threads de manière récursive, qu'il s'agira d'ordonnancer en respectant au maximum les relations de fraternité entre threads.
Une fois les bulles générées à l'exécution, le point clé réside bien sûr dans leur ordonnancement rationnel sur l'architecture sous-jacente. Nous avons donc conçu une stratégie d'ordonnancement centrée sur le respect des relations d'affinité entre threads issus des mêmes sections parallèles.
L'ordonnanceur à bulles *Affinity* a été spécifiquement conçu pour cette classe d'algorithmes. Nous considérons ainsi que chaque bulle générée contient des threads et des sous-bulles qui sont en relation, la plupart du temps via le partage de données. *Affinity* a été développé avec la conviction que la meilleure

répartition de cet ensemble d'entités est obtenue en ordonnançant les membres d'une même bulle sur le même processeur, quitte à engendrer des déséquilibres de charge provoquant une redistribution locale de temps à autre.
Affinity dispose de deux algorithmes principaux, l'un pour effectuer une distribution des groupes d'entités sur une architecture hiérarchique, l'autre pour rééquilibrer cette répartition dynamiquement.

3.2.1. Une distribution gloutonne préservant l'affinité

Bien qu'on préfèrerait toujours ordonnancer les éléments d'une même bulle sur le même processeur, il est parfois nécessaire d'extraire le contenu d'une bulle pour augmenter le nombre d'entités ordonnançables, afin notamment d'occuper un nombre plus important de processeurs. On qualifie ce procédé d'*éclatement* de bulle. Les entités d'une bulle éclatée sur le niveau modélisant la machine tout entière, noté *n0* sur 1(a), peuvent être exécutées par n'importe quels processeurs, qu'ils soient sur le même banc NUMA ou non, alors qu'on aurait pu cantonner leur éxecution sur *p0* et *p1* en n'éclatant cette bulle qu'à partir du niveau *n1*. C'est pourquoi l'ordonnanceur *Affinity* retarde au maximum l'éclatement des bulles, afin de maximiser la localité entre entités extraites.
L'algorithme de distribution de l'ordonnanceur *Affinity* est un algorithme glouton récursif, portant sur les listes de threads de la topologie. Initialement, il est appelé sur la liste la plus générale.
À chaque appel, on récupère l'ensemble des entités positionnées sur la liste passée en paramètre. La figure 1(a) représente l'état initial de l'algorithme, où une bulle contenant récursivement 7 threads est déposée sur la liste la plus générale *n0*. *Affinity* commence par regarder si on dispose d'assez d'entités pour alimenter tous les processeurs de la machine. On voit ici qu'on ne dispose que d'une seule bulle pour alimenter 4 processeurs, l'algorithme extrait donc le contenu de cette bulle (un thread et deux sous-bulles) pour les déposer sur le niveau *n0*. L'ensemble d'entités sur cette liste ayant changé, on rappelle *Affinity* sur la liste *n0*. Cette fois-ci, on dispose de trois entités pour quatre processeurs. Avant d'éclater une bulle, on analyse le contenu des entités en présence pour savoir si certaines d'entres elles ne pourraient pas, à elles-seules, alimenter un sous-ensemble de processeurs. Par exemple la première bulle contient deux sous-bulles, et peut donc occuper les processeurs *p0* et *p1*. On la dépose alors sur la première liste du niveau *n1*, et on distribue le reste des entités gloutonnement, comme présenté en figure 1(b). Ce procédé permet de retarder l'éclatement de cette bulle d'un niveau, et donc de maximiser la localité de ses sous-entités.



(a) État initial

(b) Étape intermédiaire de la distribution gloutonne

(c) Résultat de la distribution gloutonne

(d) Rééquilibrage après terminaison du thread du processeur p3

FIG. 1 – Répartition des threads et des bulles par l'ordonnanceur *Affinity*

La figure 1(c) montre la répartition obtenue après l'appel à l'algorithme d'*Affinity*. On remarque que

seule une bulle a été éclatée[1] (au niveau *n1*). Cette approche favorise donc bien les relations d'affinité, parfois au détriment de l'équilibrage de charge. C'est d'ailleurs pour cela que l'ordonnanceur *Affinity* ne pourrait être efficace sans un algorithme de vol de travail pour redistribuer les threads et bulles lorsqu'un processeur devient inactif.

3.2.2. Un vol de travail adapté aux architectures hiérarchiques

La nature irrégulière de certains programmes OpenMP rend difficile l'estimation de la charge affectée à chaque thread créé. Lorsqu'on dépose un thread sur une liste de la topologie, on ne peut pas savoir si celui-ci terminera son exécution une dizaine de cycles plus tard ou s'il engendrera la création de plusieurs milliers de threads ! Ce manque d'informations oblige l'ordonnanceur *Affinity* à considérer de la même façon toute entité, quelle que soit leur durée de vie réelle.

Il n'est de ce fait pas rare que la terminaison de threads au cours de l'exécution du programme déséquilibre la répartition gloutonne initiale. C'est pourquoi l'ordonnanceur *Affinity* dispose d'un algorithme de vol de travail spécifique, dont le rôle consiste à rééquilibrer dynamiquement la charge en tenant compte des affinités et de l'architecture, de façon portable. Notre approche diffère de celle de Cilk, dans laquelle un processeur inactif vole du travail dans la pile de tâches d'un processeur tiré au sort, ce qui fonctionne efficacement pour des algorithmes qualifiés de *cache oblivious*, dans lesquels les données ont été découpées en blocs suffisamment petits pour tenir dans la mémoire cache du processeur. *Affinity* est amené à ordonnancer une classe d'applications plus vaste, comprenant des programmes qui allouent de la mémoire. C'est pourquoi l'objectif affiché du vol de travail d'*Affinity* est de rééquilibrer la charge en tenant compte de l'architecture sous-jacente.

Un processeur inactif appelle lui-même la primitive de vol de travail d'*Affinity*. L'algorithme part alors à la recherche d'entités à voler, le plus localement possible, augmentant son domaine de recherche de proche en proche au besoin. Si plusieurs entités sont disponibles, *Affinity* choisit de voler celle contenant récursivement le plus de threads. Si seule une bulle a été trouvée au voisinage du processeur inactif, son contenu est parcouru pour trouver un sous-arbre complet à voler, comme illustré par la figure 1(d).

4. Évaluation

Grâce à *Affinity* et aux multiples fonctionnalités de MARCEL et BUBBLESCHED, il est désormais possible de paralléliser, en ajoutant quelques annotations OpenMP, de nombreux algorithmes du type *diviser pour régner*, et obtenir une bonne accélération sur architecture hiérarchique.

Nous prenons l'exemple dans cet article de l'ordonnancement par *Affinity* de l'algorithme *Multi-level Partition of Unity*.

4.1. MPU, une application de reconstruction de surface implicite

MPU [14] est une application de reconstruction de surface implicite, qui prend en paramètre un nuage de points échantillonnant une surface géométrique, afin d'en calculer une représentation mathématique. En pratique, des scanners 3D sont capables de générer un nuage de points qui modélise la surface d'un objet. La reconstruction complète d'une telle surface à partir du nuage de points associé est utilisée par de nombreuses applications de rendu d'images 3D ou de simulations physiques par exemple.

L'algorithme tente d'approximer les points de la surface par l'utilisation de quadriques. Il considère au départ la totalité du nuage de points, contenue dans un unique cube, et génère une quadrique associée à ces points. Si certains des points sont jugés comme étant trop éloignés de cette surface, le cube est subdivisé en huit cubes de même volume, à l'intérieur desquels la fonction de reconstruction est appelée récursivement. Ce procédé s'applique ainsi jusqu'à ce que l'écart entre la surface générée et les

[1] Toute entité MARCEL est créée dans une bulle *mère*, représentée sur la liste *n0*, qui est éclatée lorsque le nombre de processeurs de la machine excède 1.

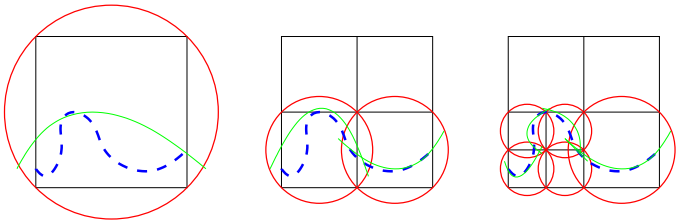


FIG. 2 – Approximation de surface utilisant une subdivision récursive engendrant une hiérarchie arborescente de zones. Chaque cube est divisé jusqu'à ce que la surface générée soit assez proche des points.

points du nuage soit suffisament petit (un paramètre estimant l'écart maximum est passé en paramètre du programme) comme le montre la figure 2.

Une telle organisation est facilement parallélisable grâce au langage OpenMP. En effet, chaque cube contient des points à approximer de façon indépendante, obtenant des partitions de surface pouvant être reconstruites par des threads différents. La façon la plus naturelle de paralléliser ce programme est donc de créer une équipe de threads OpenMP lors de chaque subdivision du volume. Aussi, l'ajout au programme séquentiel d'une seule annotation OpenMP a suffit pour le paralléliser.

4.2. Résultats

Nous avons validé notre approche en lançant l'application MPU sur un nuage de 437644 points. L'exécution correspondante provoque la création de 101185 threads.

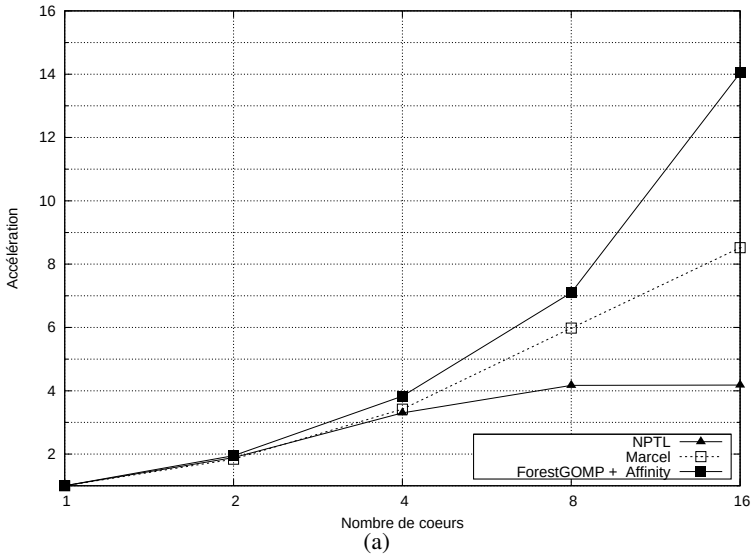
La machine sur laquelle les tests ont été effectués dispose de 8 processeurs bi-cœurs AMD Opteron, pour un total de 16 cœurs, et de 64 Go de mémoire vive. Le facteur NUMA mesuré entre les différents processeurs de la machine varie entre 1,06 (pour les processeurs voisins) à 1,4 (pour les processeurs les plus éloignés). Nous avons à la fois testé la bibliothèque de threads POSIX NPTL disponible nativement dans Linux 2.6, et la bibliothèque de threads MARCEL, en partitionnant l'ensemble des cœurs utilisables, de façon à exécuter nos tests sur respectivement 2, 4, 8 et 16 cœurs.

Dans tous nos tests, nous avons autorisé le *runtime* GOMP à générer des threads supplémentaires à l'exécution d'une section parallèle imbriquée. Les meilleures accélérations sont obtenues en créant 4 threads par section parallèle. En permettant les approches à parallélisme imbriqué, on autorise la création d'un grand nombre de threads. Les primitives de création et de gestion de ces derniers sont bien plus coûteuses pour un ordonnanceur opérant en espace noyaux comme celui de la bibliothèque NPTL. L'ordonnanceur MARCEL opère lui en espace utilisateur, ce qui explique qu'il passe mieux à l'échelle.

En revanche, aucun des supports d'exécution de ces deux bibliothèques n'ont suffisamment d'informations sur les relations entre threads pour les ordonnancer correctement (sans les éloigner) sur une machine hiérarchique. On peut voir sur la figure 3(a) que le respect des affinités en ordonnançant les groupes de threads le plus localement possible permet d'obtenir une meilleure accélération.

Afin d'évaluer l'efficacité de la politique d'ordonnancement *Affinity*/FORESTGOMP, nous avons remplacé l'algorithme de vol de travail existant par une approche de vol aléatoire, dans laquelle le processeur victime est tiré au sort. Nous avons comparé notre approche aux différentes implémentations à parallélisme de tâches de MPU proposées par François Diakhaté [6]. Dans la version File Globale, les tâches sont placées dans une file partagée par tous les processeurs de la machine. La version Opti correspond à une implémentation « à la Cilk » de l'algorithme MPU dans laquelle les tâches sont placées dans des files de Dijkstra utilisables la plupart du temps sans verrou.

Le tableau 3(b) montre que même si on obtient une accélération supérieure en optimisant « à la main » l'algorithme MPU, on approche de ces performances sans toucher au code, en respectant simplement les relations d'affinité.



(a)

NPTL	MARCEL	File globale	Vol aléatoire	*Affinity*	Opti
4,15	8,52	8,66	8,2 – 11,82	14,04	15,05

(b)

FIG. 3 – Accélérations de plusieurs implémentations de MPU

5. Conclusion

Transmettre des informations sur les relations d'affinité au support d'exécution est une technique essentielle à l'exploitation efficace des machines multiprocesseur contemporaines. Les langages de programmation parallèle comme OpenMP sont naturellement adaptés à cette transmission puisqu'ils se basent sur les indications du programmeur pour la parallélisation du programme.

Dans cet article nous avons montré comment définir et implémenter dans FORESTGOMP une politique d'ordonnancement bien adaptée à l'exécution de programmes irréguliers au parallélisme à grain très fin pouvant générer des dizaines de milliers de threads par seconde : les informations d'affinité, extraites de façon automatique des programmes OpenMP, donnent lieu à la création de groupes de threads appelés *bulles* qui sont ensuite réparties dynamiquement sur les architectures hiérarchiques en respectant au mieux les affinités. Les tests effectués sur MPU, une application de reconstruction de surface implicite au parallélisme fortement irrégulier, appuient la validité de cette approche en termes de facilité de développement pour le programmeur et de portabilité des performances.

Outre la définition de stratégies d'ordonnancement ad hoc, en particulier pour l'ordonnancement des tâches du futur standard OpenMP 3.0, ce travail ouvre de nombreuses perspectives. Par exemple, dans le cadre de l'ANR NUMASIS, des travaux préliminaires [12] ont montré que l'utilisation conjointe de BUBBLESCHED et d'une bibliothèque de gestion mémoire orientée NUMA permet de diriger dynamiquement placement mémoire et ordonnancement de thread. L'intégration de telles fonctionnalités permettront d'améliorer *Affinity* afin de décider de la migration de certaines pages mémoires lors d'un vol de travail. Ensuite, afin de privilégier l'intégrité des bulles partageant effectivement des données, il serait judicieux de mettre en œuvre des techniques d'analyse statique à la compilation et statistique à l'exécution. À terme, nous comptons proposer une extension au langage OpenMP afin de permettre au programmeur de sélectionner/définir la stratégie d'ordonnancement idoine à chaque section et ensemble de tâches parallèles du programme considéré.

Bibliographie

1. « OpenMP API User's Guide – Sun Studio 11 ». `http://docs.sun.com/source/819-3694`.
2. Dieter an MEY, Samuel SARHOLZ et Christian TERBOVEN. « Nested Parallelization with OpenMP ». *Parallel Computing*, 35(5) :459–476, octobre 2007.
3. AYGUADE, COPTY, DURANL, HOEFLINGER, LIN, MASSAIOLI, SU, UNNIKRISHNAN et ZHANG. « A proposal for task parallelism in OpenMP ». Dans *IWOMP*, Beijing, China, 2007.
4. AYGUADE, GONZALEZ, MARTORELL et JOST. « Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications ». Dans *IPDPS*, 2004.
5. BALART, DURAN, GONZÀLEZ, MARTORELL, AYGUADE et LABARTA. « Nanos Mercurium : A Research Compiler for OpenMP ». Dans *EWOMP*, octobre 2004.
6. DIAKHATÉ. « Reconstruction parallèle de surfaces par partition de l'unité hiérarchique ». Mémoire de dea, Université Bordeaux 1, juin 2007.
7. DURAN, GONZÀLES et CORBALÁN. « Automatic Thread Distribution for Nested Parallelism in OpenMP ». Dans *19th ACM ICS*, pages 121–130, Cambridge, MA, USA, juin 2005.
8. FRIGO, LEISERSON et RANDALL. « The Implementation of the Cilk-5 Multithreaded Language ». Dans *ACM SIGPLAN PLDI*, Montreal, Canada, juin 1998.
9. Andreas GERNDT, Samuel SARHOLZ, Marc WOLTER, Dieter an MEY, Christian BISCHOF et Torsten KUHLEN. « Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets ». Dans *Super Computing*, novembre 2006.
10. GONZALEZ, OLIVER, MARTORELL, AYGUADE, LABARTA et NAVARRO. « OpenMP Extensions for Thread Groups and Their Run-Time Support ». Dans *Languages and Compilers for Parallel Computing*. Springer Verlag, 2001.
11. HADJIDOUKAS et DIMAKOPOULOS. « Nested Parallelism in the OMPi OpenMP/C compiler ». Dans *EuroPar*, Rennes,France, juillet 2007. ACM.
12. JEULAND. « Ordonnancement de threads dirigé par la mémoire sur architecture NUMA ». Mémoire de master recherche, juin 2007.
13. Sven KARLSSON. « An Introduction to Balder - An OpenMP Run-time Library for Clusters of SMPs ». Dans *IWOMP*, juin 2005.
14. OHTAKE, BELYAEV, ALEXA, TURK et SEIDEL. « Multi-level partition of unity implicits ». *ACM Trans. Graph.*, 22(3) :463–470, 2003.
15. SU, TIAN, GIRKAR, HAAB, SHAH et PETERSEN. « Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures ». Dans *EWOMP*, octobre 2004.
16. TANAKA, TAURA, SATO et YONEZAWA. « Performance Evaluation of OpenMP Applications with Nested Parallelism ». Dans *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.
17. THIBAULT, BROQUEDIS, GOGLIN, NAMYST et WACRENIER. « An Efficient OpenMP Runtime System for Hierarchical Architectures ». Dans *IWOMP*, pages 148–159, Beijing,China, 6 2007.
18. THIBAULT, NAMYST et WACRENIER. « Building Portable Thread Schedulers for Hierarchical Multiprocessors : the BubbleSched Framework ». Dans *EuroPar*, Rennes,France, 8 2007. ACM.
19. Samuel THIBAULT. « *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications* ». PhD thesis, Univ. de Bordeaux 1, décembre 2007.
20. TIAN, GIRKAR, BIK et SAITO. « Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs ». *Comput. J.*, 48(5) :588–601, 2005.
21. TIAN, GIRKAR, SHAH, ARMSTRONG, SU et PETERSEN. « Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures ». Dans *8th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 47–55, avril 2003.