Ordonnancement de threads OpenMP et placement de données coordonnés sur architectures hiérarchiques

François Broquedis

RenPar'19 / SympA'13 / CFSE'7
Toulouse, France, du 9 au 11 septembre 2009

Ordonnancement de threads OpenMP et placement de données coordonnés sur architectures hiérarchiques

François Broquedis

Université de Bordeaux,
LaBRI – 351 cours de la Libération
F-33405 TALENCE – FRANCE
francois.broquedis@labri.fr

Résumé
Exploiter le potentiel des machines multiprocesseurs hiérarchiques nécessite une répartition précise des threads et des données sur l'architecture non-uniforme sous-jacente afin d'éviter des pénalités d'accès mémoire. Les langages à base de directives comme OpenMP fournissent au programmeur une façon simple de structurer le parallélisme de leurs applications et de transmettre cette information au support d'exécution. Notre support exécutif, basé sur une ordonnanceur de threads multi-niveaux combiné à un gestionnaire mémoire spécialement conçu pour les architectures NUMA, convertit cette information en indications à l'ordonnanceur pour respecter les affinités entre threads et données. Il offre une distribution dynamique de la charge de travail guidée par la structure de l'application et la topologie de la machine cible, dans le but d'atteindre la portabilité des performances. Les premières expériences montrent qu'une approche mixte, faisant intervenir conjointement déplacement de threads et migration de données se comporte mieux que les politiques de distribution de données basées sur *next-touch*, laissant entrevoir la possibilité de nouvelles optimisations.

Mots-clés : OpenMP, mémoire, NUMA, ordonnancement hiérarchique de threads, multi-cœur

1. Introduction

Les briques de base des architectures multiprocesseurs évoluent actuellement vers des puces massivement multicœur. Alors que le paysage du calcul hautes performances reste majoritairement peuplé de grappes de calcul, le degré de parallélisme à l'intérieur d'un nœud augmente. Le nombre d'unités de calcul croît aujourd'hui plus vite que n'augmente la bande passante mémoire. Afin de minimiser la contention, les constructeurs organisent unités de calcul et bancs mémoire de façon hiérarchique, donnant naissance à des architectures à accès mémoire non-uniforme. Ces architectures hiérarchiques sont répandues dans le domaine du calcul hautes performances et le deviennent pour le grand public grâce à la diffusion des technologies AMD HYPERTRANSPORT et INTEL QUICKPATH.
Il y a encore quelques années, une distribution efficace des traitements sur les différentes unités de calcul suffisait à exploiter efficacement les machines parallèles. Dans ce contexte homogène, les supports exécutifs des environnements de programmation à grain fin du langage Cilk et de la bibliothèque TBB s'avèrent être particulièrement performants mais sont cependant pénalisés dans le contexte des machines hiérarchiques, notamment lorsque les applications accèdent fréquemment à la mémoire. En effet, sur les architectures NUMA, une répartition judicieuse des données est nécessaire pour limiter au maximum le nombre d'accès mémoire distants et une potentielle congestion sur les liens intra-nœuds.
Des modèles de programmation parallèles tels OpenMP, HPF ou UPC permettent d'exprimer les relations d'affinité mémoire et il peut être intéressant d'exploiter ces informations : compilateurs et supports exécutifs doivent travailler en étroite coopération pour établir une répartition pertinente de traitements et de données, capable d'évoluer dynamiquement en fonction des besoins de l'application ou de l'état de la machine. Nos précédents travaux [9, 1] ont montré les gains de performances obtenus par une coopération pérenne entre un compilateur OpenMP et son support exécutif, en particulier sur les

Emplacement des données	Local	Local + Voisins
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. 1: Bande passante aggrégée sur une machine OPTERON à 4 nœuds comportant 4 cœurs chacun, dépendant de la quantité de travail (4 ou 16 threads) et de l'emplacement physique des données.

architectures NUMA multicœur. Nous avions pour cela développé FORESTGOMP [9] qui étend le support exécutif GOMP GNU OpenMP en s'appuyant sur la plateforme d'ordonnancement flexible BUBBLESCHED [10], permettant l'ordonnancement efficace des applications exhibant un parallélisme à grain fin. Il manquait cependant à ce support exécutif la capacité d'initier des mouvements de données pour améliorer les performances des applications aux accès mémoire intensifs.

Nous présentons dans cet article une extension du support exécutif FORESTGOMP qui connecte l'ordonnanceur à un gestionnaire mémoire adapté aux architectures NUMA. Ce nouveau support exécutif est capable de diriger l'ordonnancement des équipes OpenMP en tenant compte des informations qui leur sont attachées, relativement aux données auxquelles elles accèdent. Il lui est aussi possible de migrer des données de façon immédiate, ou différée grâce à la politique *next-touch*, dans des situations qui l'imposent. Nous traitons dans la suite de plusieurs de ces situations en mettant en avant, pour chacune d'entre elles, les paramètres à prendre en compte sur les architectures multicœur hiérarchiques.

2. Contexte et motivations

Dans cette section, nous introduisons brièvement les caractéristiques des architectures mémoire contemporaines qui compliquent la tâche du programmeur d'applications parallèles. Nous détaillons ensuite les approches logicielles existantes qui tentent de résoudre ces problèmes.

2.1. Architectures mémoire contemporaines

Les performances mémoire évoluant moins vite que le nombre de cœurs par machine, l'accès concurrent au bus mémoire provoque un goulet d'étranglement qui dégrade sensiblement les performances. Ce problème amène les constructeurs de machines parallèles à repenser le modèle de mémoire centralisée au profit d'architectures hiérarchiques à mémoire distribuée, où les caches et les nœuds mémoire sont proches de certains cœurs et éloignés des autres. Un cœur accède par exemple plus rapidement à un banc mémoire local qu'aux autres bancs de la machine. Le rapport entre les temps d'accès mémoire distants et locaux est appelé *facteur NUMA*. Il varie généralement entre 1.2 et 3 selon les architectures et influe grandement sur les performances des applications, qui s'exécuteront plus rapidement en accédant à de la mémoire locale. Le volume d'accès mémoire distants peut aussi engendrer l'apparition de contention sur les bus mémoire. De plus, la présence désormais courante de caches partagés entre plusieurs cœurs accroît la nécessité d'un ordonnancement respectant l'affinité entre traitements et données. Cependant, on programme ces machines NUMA comme une architecture multiprocesseur classique, bénéficiant d'une bande passante mémoire supérieure et d'une contention moindre, ce qui explique en partie leur position dominante sur le marché des serveurs de calcul.

Pour illustrer ce problème, nous avons effectué des expériences sur une machine OPTERON à 4 nœuds comportant 4 cœurs chacun. La deuxième ligne du tableau 1 montre qu'une application synthétique créant quelques threads sur une machine initialement non chargée obtient de meilleures performances si l'on distribue les données accédées sur les différents nœuds de la machine, maximisant ainsi la bande passante, et si l'on groupe les threads sur le même nœud, pour tirer parti des caches partagés. La troisième ligne du tableau 1 montre en revanche qu'une telle politique sur machine chargée n'obtient plus les meilleures performances, du fait de l'apparition de contention sur les liens mémoire. Dans ce cas, il est préférable de garder threads et données accédées sur le même nœud NUMA. Ces expériences suggèrent donc que tirer parti des architectures NUMA ne se résume pas à répartir les threads et les données en fonction de leurs affinités, puisqu'il faut aussi tenir compte de l'état de la machine.

2.2. Support logiciel pour la gestion mémoire et travaux apparentés

La plupart des systèmes d'exploitation modernes optent pour une politique d'allocation paresseuse : lorsque l'application alloue de la mémoire virtuelle, les pages physiques correspondantes ne sont allouées qu'au moment de leur premier accès. Cette technique a naturellement conduit à la politique d'allocation *first-touch* mise en œuvre dans la plupart des systèmes d'exploitations : chaque page mémoire est allouée dans le contexte du thread qui y accède en premier. Cependant, une page peut ne pas être allouée "au bon endroit" si le premier thread à y accéder n'est pas celui qui plus tard travaillera sur les données qu'elle contient. C'est pourquoi les programmeurs d'applications parallèles ont coutume de faire accéder de façon explicite chaque thread à leurs données pendant la phase d'initialisation, espérant ainsi que le système d'exploitation place les données à proximité des threads concernés. Cette politique a fait ses preuves sur les applications régulières. Cependant, elle n'est pas adaptée aux applications dont le motif d'accès à la mémoire change en cours d'exécution, comme celles s'appuyant sur des algorithmes à maillage adaptatif. Les affinités entre threads et données sont donc amenées à évoluer, rendant obsolète la répartition courante. Une solution consiste à déplacer explicitement les pages mémoire entre les nœuds pour garantir la localité des données tout au long de l'exécution. La migration mémoire reste cependant un mécanisme coûteux, quand il est disponible, et fortement intrusif.

La politique *next-touch* peut être perçue comme une généralisation de la politique *first-touch* : elle permet aux applications de demander au système d'exploitation la migration d'un ensemble de pages mémoire près du prochain thread qui y accèdera. Des études [4, 7, 8] montrent que cette politique améliore les performances d'applications irrégulières. Cependant, outre que cette politique soit rarement implémentée efficacement, elle ne tient pas compte de l'architecture sous-jacente et, ne coopérant pas avec l'ordonnanceur, n'est pas adaptée aux situations où plusieurs threads accèdent aux mêmes données.

Outre les études portant sur la politique *next-touch*, plusieurs projets de recherche ont été menés sur l'amélioration de la distribution des données pour les programmes OpenMP sur les architectures NUMA. Certains d'entre eux [2] reposent sur l'enrichissement de la norme OpenMP par l'ajout de directives qui s'inspirent du langage HPF. De telles directives sont utiles pour organiser les données de façon à maximiser la localité, et, dans notre contexte de recherche, peuvent constituer un moyen peu intrusif de transmettre des informations sur l'affinité mémoire à notre support exécutif. Nikolopoulos *et al.* [6] ont proposé un mécanisme de migration automatique des pages mémoire reposant sur une instrumentation du code utilisateur. Un échantillonnage des premiers tours de boucle des applications OpenMP détermine les affinités entre threads et données. Cette approche s'avère être plus performante encore lorsque le gestionnaire de migration et le système d'exploitation sont capables de communiquer. Elle ne convient cependant que dans les cas où le motif d'accès à la mémoire ne change pas en cours d'exécution.

3. Une approche dynamique pour le placement de threads et de données

L'amélioration des performances des applications parallèles sur les architectures NUMA hiérarchiques passe, de notre point de vue, par un ordonnancement multi-niveaux qui exploite les informations d'affinité mémoire *tout au long de l'exécution*. Cette section présente les éléments permettant d'y parvenir.

3.1. Objectifs

Notre principal objectif consiste à répartir threads et données de façon dynamique, en tenant compte des informations d'affinités fournies par le programmeur, le compilateur ou des compteurs matériels. Dans le cas d'applications irrégulières, les décisions d'ordonnancement et de placement ne peuvent être prises qu'en cours d'exécution et nécessitent une connaissance précise de l'architecture sous-jacente (topologie mémoire, caches partagés, *etc.*) et l'état de la machine. De notre point de vue, c'est au support exécutif de faire correspondre la structure parallèle de l'application à l'architecture de la machine.

Notre approche repose sur l'utilisation séparée de politiques d'ordonnancement spécifiques à différents niveaux de la topologie de la machine. Par exemple, un vol de travail de bas niveau peut équilibrer la charge entre cœurs voisins pendant qu'un ordonnancement de plus haut niveau migre des groupes de threads d'un nœud NUMA à un autre, en déplaçant au besoin leurs données, sans remettre en cause l'ordonnancement interne à ce groupe établi auparavant. Pour cela, le support exécutif doit "se souvenir" des relations entre threads et données *tout au long de l'exécution*. De telles informations devraient pouvoir être quantifiées et mises à jour dynamiquement, à la création d'un thread ou à l'allocation d'une donnée

par exemple, que ce soit par le programmeur d'applications, par une analyse statique du compilateur, ou encore par le support exécutif via l'instrumentation du code utilisateur. Le problème se résume alors à quand réagir et comment. Nous avons identifié plusieurs évènements susceptibles de remettre en cause une distribution. C'est le cas par exemple quand l'application alloue ou libère une ressource, quand un processeur devient inactif, quand des compteurs de performance matériels détectent une anomalie, comme de multiples accès à des nœuds distants, ou encore quand le programmeur d'application ajoute explicitement dans son code des informations d'affinité. Afin d'évaluer ce modèle, nous avons développé une extension du langage OpenMP basée sur l'instrumentation des applications.

3.2. FORESTGOMP, un support exécutif OpenMP performant sur les architectures hiérarchiques

FORESTGOMP est une extension du support exécutif GNU OpenMP s'appuyant sur la bibliothèque de threads utilisateur MARCEL/BUBBLESCHED. FORESTGOMP bénéficie du mécanisme performant de migration des threads MARCEL permettant le contrôle de l'ordonnancement des threads OpenMP. Le coût d'une migration de thread MARCEL est en effet de l'ordre de $0,06\mu s$ par thread et la latence n'excède jamais les $2,5\mu s$ contre $8\mu s$ et $4,8\mu s$ pour Linux. Ce support exécutif génère de plus des groupes de threads automatiquement, appelés *bulles*, à chaque section parallèle OpenMP dans le but de préserver *de façon pérenne* les relations d'affinités entre threads d'une même équipe. Une bulle correspond donc à exactement une équipe OpenMP. FORESTGOMP s'appuie aussi sur MAMI, une bibliothèque d'allocation pour les architectures NUMA qui implémente entre autres les politiques *first-touch* et *next-touch* et qui permet aussi une migration explicite d'une zone mémoire d'un nœud NUMA vers un autre, offrant au support exécutif un contrôle précis des mouvements de données.

La bibliothèque MARCEL modélise avec précision les architectures hiérarchiques, en détectant par exemple les cœurs, les caches partagés, les processeurs et les nœuds NUMA. BUBBLESCHED et MAMI peuvent s'appuyer sur cette topologie pour prendre des décisions quant au placement des bulles et des données. Les façons d'ordonnancer ces bulles sont définies par des ordonnanceurs spécifiques à BUBBLESCHED qui fournit une interface pour en concevoir de nouveaux. Par exemple, l'ordonnanceur à bulles *Cache* [1], dont le principal but est d'améliorer l'utilisation des caches partagés en ordonnançant ensemble les threads d'une même bulle, a été développé en utilisant cette interface. En revanche, cet ordonnanceur ne permet pas de prendre en compte l'affinité mémoire. Nous avons donc défini un nouvel ordonnanceur pour répondre à cet objectif.

3.3. Une politique d'ordonnancement guidée par les affinités mémoire

Nous adoptons une approche conservative pour équilibrer la charge de travail dans le but de minimiser les transferts mémoire et de maximiser l'affinité cache entre threads. Il s'agit d'attacher à tout thread la taille et l'emplacement des données qu'il utilise. De cette façon, l'ordonnanceur à bulles est capable de diriger la répartition des threads et d'automatiser la migration des données. Nous avons développé l'ordonnanceur à bulles *Memory* qui s'appuie sur la bibliothèque MAMI pour répartir, en trois phases distinctes, les équipes de threads en fonction de leurs affinités mémoire. La première phase consiste à attirer les threads vers le nœud hébergeant le plus de leurs données. La deuxième phase corrige la première répartition pour occuper l'ensemble des processeurs de la machine, en déplaçant au besoin les threads ayant le moins d'affinité avec le nœud NUMA sur lequel ils ont été précédemment placé de façon à réduire le volume des données à migrer. La troisième phase migre les données distantes à proximité de leurs threads. Une fois le travail de l'ordonnanceur *Memory* terminé, l'ordonnanceur *Cache* opère à l'intérieur de chacun des nœuds pour effectuer une distribution tenant compte de l'affinité cache [1].

La plateforme FORESTGOMP a aussi été étendue pour offrir au programmeur d'applications la possibilité de communiquer directement les affinités mémoire au support exécutif. Elle dispose ainsi d'un nouvel ensemble de fonctions utilisables depuis l'extérieur permettant de spécifier l'affinité mémoire de l'équipe OpenMP *avant sa création*, autorisant ainsi au support exécutif quelques optimisations ou à l'intérieur d'une région parallèle. FORESTGOMP ne déplace les threads qui si une nouvelle affinité mémoire vient contredire la distribution en place.

4. Evaluation des performances

Cette section présente notre plateforme expérimentale et détaille les améliorations de performances offertes par FORESTGOMP sur un ensemble d'applications à la complexité croissante.

Opération	LIBGOMP Pire - Meilleur	FORESTGOMP Pire - Meilleur
Copy	6 747 - 8 577	7 851 - 7 859
Scale	6 662 - 8 566	7 821 - 7 828
Add	7 132 - 8 821	8 335 - 8 340
Triad	7 183 - 8 832	8 357 - 8 361

(a) Bande passante agrégée obtenue par le benchmark STREAM sur une machine à 16 cœurs (1 thread par cœur) en Mo/s.

Opération	LIBGOMP Pire - Meilleur	FORESTGOMP Pire - Meilleur
Copy	6 900 - 8 032	8 302 - 8 631
Scale	6 961 - 7 930	8 201 - 8 585
Add	7 231 - 8 181	8 344 - 8 881
Triad	7 275 - 8 123	8 504 - 9 217

(b) Bande passante agrégée obtenue par le benchmark Nested-STREAM en Mo/s.

Opération	LIBGOMP	FORESTGOMP
Triad Phase 1	8 144 Mo/s	9 108 Mo/s
Triad Phase 2	3 560 Mo/s	6 008 Mo/s

(c) Débits moyens obtenus par le benchmark Twisted-STREAM en Mo/s.

TAB. 2: Performances obtenues par STREAM, Nested-STREAM et Twisted-STREAM.

4.1. Plateforme expérimentale

La plateforme expérimentale est composée de quatre processeurs AMD OPTERON 8347HE comprenant chacun quatre cœurs cadencés à 1,9 GHz. Chaque processeur contient 2Mo de cache de niveau 3 partagé entre les cœurs qu'il contient, ainsi que 8 Go de mémoire vive. Les accès mémoire distants sur cette machine sont d'autant plus lents que la distance entre les nœuds augmente, à savoir 83 ns, 98 ns ($\times$ 1,18) et 117 ns ($\times$ 1,41) selon qu'on accède en lecture au nœud local, à un nœud distant d'un ou de deux liens. La latence de base et le facteur NUMA sont plus grands pour les accès en écriture (142 ns, 177 ns ($\times$ 1,25) et 208 ns ($\times$ 1,46)), qui génèrent plus de trafic. Dans le cas d'un accès en écriture, le matériel peut mettre à jour le nœud mémoire distant de façon différée (politique *Write-Back*).

4.2. STREAM

STREAM [5] est un benchmark synthétique développé en C, parallélisé en OpenMP, qui mesure la bande passante mémoire soutenue de la machine et le taux de calcul correspondant en effectuant des opérations simples sur des vecteurs d'entiers. Ces derniers sont suffisament grands (20 millions d'entiers double précision) pour ne pas tenir dans le cache et sont alloués selon la politique *first-touch* puis initialisés en parallèle pour garantir à chaque thread la localité de ses données.

Le tableau 2a montre les résultats obtenus par les supports exécutifs FORESTGOMP et LIBGOMP de GCC 4.2 sur le benchmark STREAM non modifié. Les performances obtenues par la bibliothèque LIBGOMP montrent une variation pouvant aller jusqu'à 20%, qui peut être expliquée par le fait que les threads de l'application peuvent échanger leurs positions au moment d'un changement de contexte, invalidant ainsi la distribution mémoire initiale. Le support exécutif FORESTGOMP obtient au contraire des performances stables. En effet, en l'absence d'informations d'affinité mémoire, l'ordonnanceur *Cache* s'occupe seul de la répartition des threads et leur attribue à chacun un cœur dédié. Dans ce cadre, la politique d'allocation *first-touch* est valide tout au long de l'exécution et obtient d'excellentes performances.

4.3. Nested-STREAM

Afin d'étudier l'impact du placement des threads et des données sur les performances d'une application parallèle, nous avons modifié STREAM en y insérant des régions parallèles imbriquées : l'application crée désormais autant d'équipes OpenMP que de nœuds NUMA de la machine cible. Chaque équipe travaille sur son propre jeu de vecteurs STREAM initialisés en parallèle, à la façon de la version originale de STREAM. Pour s'adapter à notre plateforme expérimentale, l'application crée ici quatre équipes de quatre threads. Le tableau 2b montre les résultats obtenus par LIBGOMP et FORESTGOMP.

La bibliothèque LIBGOMP maintient un pool de threads pour les régions parallèles non imbriquées. De nouveaux threads sont créés chaque fois que l'application rencontre une région parallèle imbriquée et détruits une fois leur travail terminé. Rien n'oblige le support exécutif à placer ces threads près du thread

maître de leur équipe, ce qui explique pourquoi les résultats obtenus par LIBGOMP varient de 23%.
Pour FORESTGOMP, l'ordonnanceur à bulles sous-jacent distribue les threads au moment où l'application atteint la région parallèle. Chaque thread est placé sur un nœud NUMA différent, de façon permanente. De plus, FORESTGOMP assure que les threads des régions imbriquées sont créés à l'endroit où s'exécute le thread maître de l'équipe naissante. Comme les vecteurs accédés par ces threads ont été initialisés par le thread maître, traitements et données se retrouvent sur le même nœud NUMA, ce qui explique les bonnes performances obtenues et la stabilité des résultats.

4.4. Twisted-STREAM

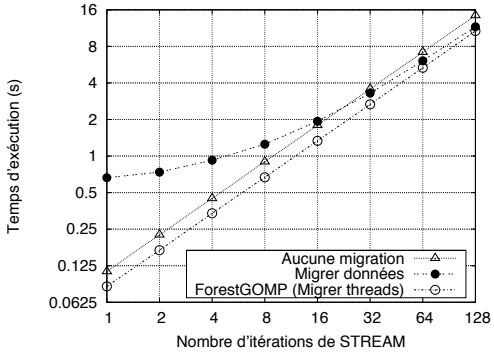
Pour compliquer le motif d'accès mémoire de STREAM, nous avons développé le benchmark Twisted-STREAM, qui contient deux phases distinctes. La première phase se comporte exactement comme Nested-STREAM, au détail près que seule l'opération *Triad* est exécutée, étant la seule à faire intervenir trois vecteurs de 160Mo chacun. Lors de la deuxième phase, chaque équipe travaille sur un jeu de vecteurs différent de celui sur lequel elle travaillait durant la première phase. La politique *first-touch* n'obtient de bons résultats que sur la première phase, comme le montre le tableau 2c. Une solution pour combler cette perte de performances fait intervenir la politique *next-touch* pour migrer les pages mémoire entre les deux phases de l'application. Nous montrons dans la suite qu'elle n'est pas toujours la meilleure réponse au problème de localité mémoire.
Nous avons expérimenté deux placements mémoire différents pour la seconde phase de Twisted-STREAM. Dans le premier, tous les vecteurs sont positionnés sur un nœud distant, alors que dans le second, seulement deux d'entre eux le sont. Nous utilisons l'API FORESTGOMP pour exprimer quelles données sont utilisées dans la seconde phase de l'application.

4.4.1. Données distantes

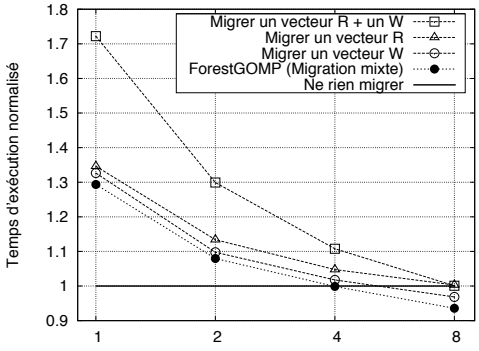
Le support exécutif sous-jacent dispose de deux options principales pour résoudre le problème des accès distants : il peut soit migrer les vecteurs vers le nœud NUMA sur lesquels sont ordonnancés les threads qui y accèdent, soit déplacer les threads vers les données. La figure 1a montre les résultats obtenus par chacune des options. Déplacer les threads est assurément la meilleure solution dans ce cas. La migration de 16 threads est bien plus rapide que la migration de trois vecteurs de 160Mo, et garantit que chaque équipe accède exclusivement à de la mémoire locale.

4.4.2. Données réparties localement et à distance

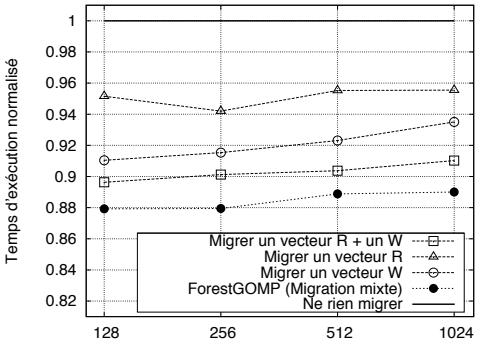
Dans ce cas, seulement deux des trois vecteurs STREAM sont situés à distance, l'un d'entre eux étant accédé en lecture, l'autre en écriture. Nous avons étudié l'impact du facteur NUMA en ne migrant qu'un seul des deux vecteurs distants (figure 1b). Comme mentionné en section 4.1, migrer le vecteur accédé en écriture offre un gain de performances supérieur, une écriture générant plus de trafic qu'une lecture.
Afin d'obtenir une meilleure répartition de threads et de données, le support exécutif sous-jacent peut bien entendu migrer les deux vecteurs distants vers le nœud qui héberge le troisième. En revanche, ne déplacer que les threads ne permet pas dans cette situation la suppression de tous les accès distants. C'est pourquoi nous proposons une approche mixte dans laquelle le support exécutif FORESTGOMP déplace les threads et le vecteur local vers le nœud contenant le plus de données, c'est-à-dire celui hébergeant les deux vecteurs distants dans ce cas. Nous obtenons de cette façon une répartition assurant la suppression totale des accès distants en ayant migré le moins de données possible. La figure 1b montre que le surcoût de cette approche est inférieur à celui engendré par la migration *next-touch* qui provoque la migration du double de données, tout en obtenant les meilleures performances quand la charge de travail par thread augmente, comme indiqué sur la figure 1c.
Nous avons aussi lancé les trois versions de STREAM à l'aide du compilateur Intel 11.0. Les performances obtenues sont meilleures que FORESTGOMP sur la version originale de STREAM (10 500 Mo/s) grâce notamment à des optimisations à la compilation. Les performances chutent néanmoins significativement sur les autres versions, avec des débits moyens de 7 764 Mo/s sur Nested-STREAM et 5 488 Mo/s sur la seconde phase de Twisted-STREAM, alors que FORESTGOMP obtient les meilleures performances.



(a) 3 vecteurs distants (100%)



(b) 2 vecteurs distants (66%), premières itérations



(c) 2 vecteurs distants (66%), itérations suivantes

FIG. 1: Temps d'exécution de différentes politiques de placement de threads et données sur différentes versions du benchmark Twisted-STREAM.

5. Discussion

Ces expériences montrent l'importance de la prise en compte de l'état de la machine, et notamment de la contention mémoire, que l'on peut minimiser en déplaçant prioritairement les données accédées en écriture, qui génèrent plus de trafic. Même si FORESTGOMP obtient les meilleurs résultats sur chacun des programmes synthétiques présentés dans cet article de façon totalement automatique, il reste encore des situations à étudier. Une donnée peut par exemple être attachée à plusieurs threads. Dans ce cas, il est possible d'effectuer une distribution des pages mémoire en tourniquet sur le sous-ensemble de nœuds qui hébergent les threads en question. L'analyse de la fréquence d'accès des données attachées pourrait aussi aider à prendre des décisions dans des situations conflictuelles où threads et données ne peuvent tous être placés localement. L'ordonnanceur doit aussi toujours être conscient de la charge mémoire du nœud vers lequel il souhaite migrer de la mémoire, sans quoi ses décisions peuvent être ignorées par le système d'exploitation.

6. Conclusion et perspectives

L'exploitation efficace des machines multiprocesseurs hiérarchiques nécessite une répartition judicieuse des threads et des données sur l'architecture non-uniforme sous-jacente. Les langages à base de directives fournissent une solution portable pour exprimer la structure parallèle des applications. Grâce

à cette information, l'ordonnanceur peut prendre des décisions appropriées quant à la migration des zones mémoires ou au déplacement des threads sur l'architecture. En effet, l'affinité entre threads et données est à prendre en compte pour améliorer la latence et surtout éviter les congestions mémoire sur les architectures NUMA.

Nous proposons un ordonnanceur de threads multi-niveaux combiné avec un gestionnaire mémoire pour machines NUMA. Il offre une répartition dynamique et cohérente de la charge de travail consciente des besoins de l'application et des contraintes du matériel, comme la capacité mémoire des nœuds, aidant ainsi à atteindre la portabilité des performances. Les expériences présentées dans cet articles montrent l'intérêt des approches mixtes, dans lesquelles le support exécutif migre threads et données conjointement. De plus, les approches traditionnelles inspirées de la politique *next-touch* sont quant à elles parfois insuffisantes, puisqu'elles ignorent par exemple la charge mémoire des nœuds. Migrer les threads s'avère être une solution plus performante pour éviter la saturation d'un nœud.

Nous avons étendu la plateforme FOREST GOMP pour améliorer les critères de décisions d'ordonnancement et automatiser la phase de migration des données. Elle obtient aujourd'hui les meilleures performances sur chacune des applications présentées dans cet article, et ce de façon totalement automatique. Les statistiques récoltées par les compteurs matériels pourraient sans aucun doute aider à rendre notre approche encore plus dynamique. Il nous faut aussi expérimenter plus avant cet ordonnancement à la fois sur des applications synthétiques et industrielles. Ces résultats suggèrent aussi le besoin d'étendre OpenMP pour transmettre les affinités mémoire au support d'exécution. Cette évolution pourrait aussi élargir le spectre d'OpenMP aux approches de programmation hybride [3].

Bibliographie

1. François BROQUEDIS, François DIAKHATÉ, Samuel THIBAULT, Olivier AUMAGE, Raymond NAMYST et Pierre-André WACRENIER. « Scheduling Dynamic OpenMP Applications over Multicore Architectures ». Dans *International Workshop on OpenMP (IWOMP)*, West Lafayette, IN, mai 2008.
2. Barbara M. CHAPMAN, Fréderic BREGIER, Amit PATIL et Achal PRABHAKAR. « Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems ». Dans *Concurrency : Practice and Experience*, volume 14, pages 713–739. John Wiley & Sons, 2002.
3. Alex DURAN, Josep M. PEREZ, Eduard AYGUADE, Rosa BADIA et Jesus LABARTA. « Extending the OpenMP Tasking Model to Allow Dependant Tasks ». Dans *International Workshop on OpenMP (IWOMP)*, West Lafayette, IN, mai 2008.
4. Henrik LÖF et Sverker HOLMGREN. « affinity-on-next-touch : increasing the performance of an industrial PDE solver on a cc-NUMA system ». Dans *19th ACM International Conference on Supercomputing*, pages 387–392, Cambridge, MA, USA, juin 2005.
5. John D. MCCALPIN. « Memory Bandwidth and Machine Balance in Current High Performance Computers ». *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, 1995.
6. Dimitrios S. NIKOLOPOULOS, Theodore S. PAPATHEODOROU, Constantine D. POLYCHRONOPOULOS, Jesús LABARTA et Eduard AYGUADÉ. « User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors ». Dans *International Conference on Parallel Processing*, pages 95–103. IEEE Computer Society Press, septembre 2000.
7. Markus NORDÉN, Henrik LÖF, Jarmo RANTAKOKKO et Sverker HOLMGREN. « Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers ». Dans *Second International Workshop on OpenMP (IWOMP)*, Reims, France, 2006.
8. Christian TERBOVEN, Dieter an MEY, Dirk SCHMIDL, Henry JIN et Thomas REICHSTEIN. « Data and Thread Affinity in OpenMP Programs ». Dans *MAW '08 : Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384, New York, NY, USA, 2008. ACM.
9. Samuel THIBAULT, François BROQUEDIS, Brice GOGLIN, Raymond NAMYST et Pierre-André WACRENIER. « An Efficient OpenMP Runtime System for Hierarchical Architectures ». Dans *International Workshop on OpenMP (IWOMP)*, pages 148–159, Beijing, China, juin 2007.
10. Samuel THIBAULT, Raymond NAMYST et Pierre-André WACRENIER. « Building Portable Thread Schedulers for Hierarchical Multiprocessors : the BubbleSched Framework ». Dans *European Conference on Parallel Computing (EuroPar)*, Rennes, France, août 2007. ACM.