RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Extensions d'OpenMP pour les Architectures à Hiérarchie Mémoire Multiple

Frédéric Brégier, Barbara Chapman, Amit Patil, Achal Prabhakar

University of Houston,
Computer Science Department,
Philip. G. Hoffman Hall, 4800 Calhoun Street,
Houston Texas, 77204-34775 - USA
bregier@cs.uh.edu, chapman@cs.uh.edu

Résumé
Dans cet article, nous présentons un ensemble d'extensions à OpenMP qui tendent à améliorer les performances sur des architectures du type NUMA, architectures à mémoire distribuée et grappes de noeuds à mémoire partagée (architectures à hiérarchie mémoire multiple). Ces extensions utilisent essentiellement le concept de privatisation en s'inspirant de HPF. Nous présentons des expérimentations montrant l'intérêt de cette approche sur deux types d'architectures, une machine cc-NUMA avec un SGI Origin 2000, et une architecture à mémoire distribuée avec un IBM SP2 en utilisant une approche de mémoire partagée répartie logicielle avec la bibliothèque TreadMarks. Ces expérimentations sont représentatives de certains problèmes rencontrés lorsque l'on considère la programmation de ces architectures.

Mots-clés : OpenMP, Architecture NUMA, Mémoire partagée répartie logicielle, Privatisation, HPF

1. Introduction

OpenMP [3] propose un standard pour la programmation parallèle d'architectures à mémoire partagée. Il propose une approche relativement simple pour créer des codes parallèles, permettant en particulier un développement incrémental de codes parallèles depuis des codes séquentiels existants.
Mais il est extrêmement difficile de construire des systèmes à mémoire réellement partagée de taille quelconque. Ainsi les systèmes actuels peuvent avoir 16 processeurs connectés à la mémoire partagée, et des systèmes à 32 processeurs devraient bientôt être disponibles. Leur bus partagé entre les processeurs et la mémoire représente une réelle limitation. Pour solution, certains vendeurs ont construit des machines constituées d'un certain nombre de noeuds, chacun ayant sa propre mémoire partagée interne ; ces noeuds sont connectés de facon à créer une plate-forme extensible avec un accès à la mémoire non uniforme (NUMA). Si un tel système a un espace d'adressage global et si le matériel supporte une cohérence de cache sur l'ensemble de la machine, OpenMP peut être implémenté sur ce système.
De plus, il y a un intérêt croissant à programmer des machines à mémoire partagée en grappes, ou même des machines à mémoire distribuée comme un simple super-calculateur en utilisant par exemple un système de mémoire partagée répartie par support logiciel (SDSM en anglais). Cependant, OpenMP n'a aucun dispositif pour supporter l'optimisation des codes sur des systèmes NUMA, et il n'est pas non plus un bon candidat comme modèle de programmation pour de tels systèmes NUMA étendus, puisqu'il ne peut pas s'étendre au-delà d'un espace d'adressage unique.
Dans la suite, nous présentons d'abord un ensemble d'extensions à OpenMP (section 2) qui est destiné à améliorer l'exécution sur des systèmes NUMA et qui pourrait également être mis en application à travers un système à mémoire distribuée (par l'intermédiaire de SDSM ou de techniques de compilation empruntées à HPF [8]). Ces extensions utilisent principalement le concept de la privatisation de données. Nous présentons des expériences montrant l'intérêt de cette approche sur deux architectures (section 3), une machine cc-NUMA avec un Origin 2000 de SGI, et une machine à mémoire distribuée avec un

IBM SP2 en utilisant une approche SDSM avec TreadMarks [13, 1]. Ces expériences (sections 4 et 5) sont représentatives de problèmes liés à la programmation d'architecture à hiérarchie mémoire multiple.

2. Programmation d'architectures NUMA

Afin d'être efficace, le programmeur doit faire attention à la hiérarchie mémoire de l'architecture sous-jacente. Ainsi, on peut écrire un programme dans un style SPMD employant la privatisation des données de facon à placer toutes les données dans un niveau de mémoire aussi proche que possible du processeur. Mais cette approche est difficile et exige une attention spéciale puisque le code n'est plus lié à la version séquentielle originale, et donc difficile à maintenir, développer ou déboguer.
Pour faciliter l'effort de programmeur et rester aussi près que possible de la version séquentielle originale, nous proposons un ensemble d'extensions à OpenMP, appelé NUMA-OMP pour produire automatiquement ces codes. Ces extensions sont principalement empruntées à HPF [8] et concernent le placement des données et l'accès local aux données.

Nous proposons d'abord une directive `DISTRIBUTE` avec trois formats, $\star$, `BLOCK` et `CYCLIC`. Elle spécifie comment les données sont placées (privatisées) sur les mémoires des processeurs. À ce jour, nous ne permettons que seule une dimension soit distribuée parmi toutes les dimensions d'un tableau, et ce pour chaque tableau partagé.
La directive `ON HOME` permet, via le placement de la donnée spécifiée dans la directive, d'indiquer quel thread doit exécuter le bloc d'instructions qui suit.
Nous proposons aussi d'employer la directive `SHADOW` pour spécifier quelles parties des tableaux distribués vont être accédées non localement fréquemment. L'utilisation de cette directive permet au compilateur d'allouer un stockage complémentaire, des zones de recouvrement sur chaque processeur, afin de permettre un accès local aux données voisines distantes.

À ce jour, toutes nos expériences sont faites à l'aide de codes produits à la main, mais nous projetons d'obtenir directement ces codes en employant plus tard notre compilateur d'étude. Nous emploierons alors la sentinelle `!$NMP` pour indiquer les directives NUMA-OMP, comme suit:

```
!$NMP DISTRIBUTE A(*,BLOCK)
!$NMP SHADOW A(0,1)

!$OMP PARALLEL DO
!$NMP ON HOME (A(:,I))
DO I = 1, N
...
END DO
!$OMP END PARALLEL
```

3. Architectures utilisées

Nous avons employé deux sortes d'architectures : d'une part une architecture cc-NUMA avec un SGI Origin 2000 et d'autre part un système à mémoire distribuée combiné à un SDSM avec un IBM SP2.

3.1. Architecture cc-NUMA

Le modèle de programmation à mémoire partagée développé pour les SMPs (Symmetric Multi-Processor) s'est avéré être très populaire, cependant les SMPs purs ne peuvent pas atteindre de grande taille en raison du bus partagé entre les processeurs et la mémoire. Pour résoudre cela, les vendeurs ont concu des systèmes divisés dans des modules plus petits. Chaque noeud est un SMP pur avec un réseau haute débit connectant tous les noeuds. De tels systèmes ont un espace d'adressage mémoire global et le modèle de programmation à mémoire partagée s'applique. Ces machines sont aussi appelées des machines NUMA à cause de la non-uniformité dans les temps d'accès à la mémoire distante et locale [18]. Les systèmes qui fournissent la cohérence de cache sont appelés cc-NUMA. Des exemples commerciaux incluent l'Origin 2000 de SGI et les AplhaServer GS80, GS106 et GS320 de Compaq.

Le modèle de programmation à mémoire partagée assume des temps d'accès uniformes pour l'espace d'adressage entier. L'application donc directe du modèle SMP aux machines NUMA peut ne pas amener des performances optimales en raison d'éventuelles mauvaises propriétés de localité spatiale.
Néanmoins, les systèmes NUMA peuvent délivrer cette haute performance pour des applications basées sur la mémoire partagée si le placement des données est local au processeur sur lequel le calcul est exécuté. OpenMP lui-même ne fournit pas de facilité pour le placement des données, néanmoins on s'attend sur des machines NUMA à ce que la localité des données augmentant, les performances pour les applications OpenMP augmenteront elles aussi.

Comme SGI était conscient des problèmes de localité des données sur une machine NUMA, ils ont proposé [10] plusieurs facons de distribuer les données (et les calculs en fonction des données), en employant une politique de premier contact ("*First Touch*", le premier processeur accédant à un élément d'une page se verra attitrer cette page dans sa mémoire locale) ou l'utilisation explicite de directives de distribution. Bien que distribués, il faut noter que les tableaux demeurent toujours partagés via le support fourni par le système d'exploitation. La clause `AFFINITY` permet d'indiquer, tout comme la directive `ON HOME`, le thread associé pour chaque itération.
Selon la documentation SGI, *aucune distribution* (*First Touch*) et les directives *Distribute* sont presque semblables. La seule différence est que la distribution via `DISTRIBUTE` est faite à la compilation alors que le *First Touch* a lieu à l'exécution. Par défaut, le compilateur OpenMP de SGI utilise le *First Touch* [5].
La directive `DISTRIBUTE` de SGI distribue à la HPF les tableaux, mais elle emploie une distribution basée sur la pagination de la mémoire (basée sur la taille d'une page).
La directive `DISTRIBUTE_RESHAPE` emploie une autre facon de distribuer un tableau, en se basant sur les éléments et non sur les pages. De ce fait, les tableaux sont réorganisés et l'arithmétique des pointeurs est employée pour accéder à un élément. Cette facon de faire est très semblable à la version que nous avons produite à la main, excepté que le tableau reste toujours partagé parmi les processeurs.

Nous avons donc mis en oeuvre pour chaque exemple 4 versions, 1 *Sans Distribution*, 1 avec un `DISTRIBUTE`, 1 avec un `DISTRIBUTE_RESHAPE` et finalement 1 produite à la main appelé *Private*.
Cette dernière est semblable au style de programmation SPMD. Les tableaux sont distribués sur l'ensemble des processeurs et déclarés privés. Nous employons chaque fois un buffer partagé de petite taille pour réaliser les communications entre les threads. Ainsi les calculs de chaque thread portent uniquement sur les parties privées des tableaux et le partage est réalisé à travers ces petits buffers partagés.

Les expériences ont été conduites sur l'Origin 2000 de SGI au NCSA. La configuration actuelle du NCSA consiste en une machine de 48 processeurs nommée modi4, et de 11 autres de 64 à 256 processeurs. modi4 est disponible pour l'accès interactif, tandis que les autres machines sont utilisées via la soumission de jobs. Chaque machine est homogène, mais entre elles opèrent des différences : le type de processeur varie de 195MHz R10000 à 250MHz R10000 et la mémoire globale varie entre 16 et 128 Go. Sur toutes ces machines, le niveau 1 de cache pour les données et les instructions sont tous les deux de 32 Ko, tandis que le niveau 2 de cache est de 4 Mo. Toutes nos expériences ont été conduites dans un mode partagé. Nous avons employé le compilateur MIPSPRO 7 Fortran 90 de SGI avec l'option *-mp* pour toutes les applications, sauf LU où nous avons ajouté l'option *-Ofast*.

3.2. Mémoire partagée répartie par support logiciel

Les systèmes dans lesquels la cohérence de la mémoire est gérée par un logiciel sont à l'autre bout du spectre. De telles machines consistent généralement en un certain nombre de processeurs reliés par un réseau rapide, où toutes les mémoires sont physiquement distribuées avec une couche logicielle imitant un espace d'adressage global. Les caractéristiques de telles machines sont semblables au cc-NUMA. Pour obtenir de bonnes performances, il est impérieux que l'accès aux données soit local au processeur exécutant le calcul. La non-uniformité dans l'accès à la mémoire est plus marquée avec les SDSM que dans les machines cc-NUMA. Ainsi, la localisation des données devrait avoir un effet beaucoup plus prononcé.
Nous avons employé le système SDSM TreadMarks. TreadMarks est basé sur le protocole de cohérence paresseux [12] et le protocole d'écrivains multiples pour réduire l'effet de faux partage.

Nous mettons en oeuvre pour chaque exemple deux versions, une partagée, une privée. La mémoire est par défaut privée dans TreadMarks et seulement la mémoire allouée explicitement à l'aide d'appels à la bibliothèque est partagée [15]. Pour la version partagée, l'ensemble du tableau est déclaré partagé.
La version privée n'alloue qu'un buffer nécessaire pour permettre la mise à jour de la zone de recouvrement (ou shadow) (Jacobi en section 4.1 et LBE en section 4.2) ou la diffusion de la colonne normalisée (LU en section 4.3). Ainsi, tous les calculs emploient des données privées. Ces codes SPMD avec TreadMarks sont très proches des codes SPMD écrit pour les versions privées d'OpenMP sur l'Origin 2000.

Les expériences ont été réalisées sur le SP2 d'IBM situé à l'Université de Houston. Ce système comporte 64 noeuds assemblé en 5 parties contenant : 8 noeuds fins avec une mémoire de 256 Mo chacun, 48 noeuds fins avec une mémoire de 128 Mo, 4 noeuds larges avec une mémoire de 512 Mo et enfin 4 noeuds larges avec 2 Go de mémoire. Les noeuds fins sont des P2SC (Power2 Superchip) à 120MHz, tandis que les noeuds larges sont des P2SC à 135MHz. Toutes les expériences ont été faites sur les noeuds fins, dans le mode non-dédié, c'est à dire que le système était partagé avec d'autres utilisateurs. Tous les noeuds utilisent le système IBM AIX 4.3. Le compilateur C d'IBM C version 3.6.6 a été utilisé pour tous les programmes qui ont été liés avec la version 1.0.3.3-BETA de la bibliothèque TreadMarks. Les options de compilation suivantes ont été utilisées : `-O3 -qstrict -qarch=pwr2 -qtune=pwr2` .

4. Programmes utilisés

Nous avons évalué les performances de notre approche sur des noyaux typiques à savoir Jacobi, la factorisation LU et l'équation de Boltzmann.

4.1. Jacobi

Jacobi est une méthode courante pour résoudre des équations différentielles partielles. L'algorithme de Jacobi possède une localité spatiale excellente (Programme 1 et Figure 1(a)). Tous les accès aux données peuvent être rendus locaux exceptés pour les éléments à la frontière accédés en lecture partagée. Ainsi, c'est un exemple typique où des architectures NUMA peuvent obtenir une bonne performance puisque l'accès partagé est en lecture seule et de plus sur une faible portion des données.

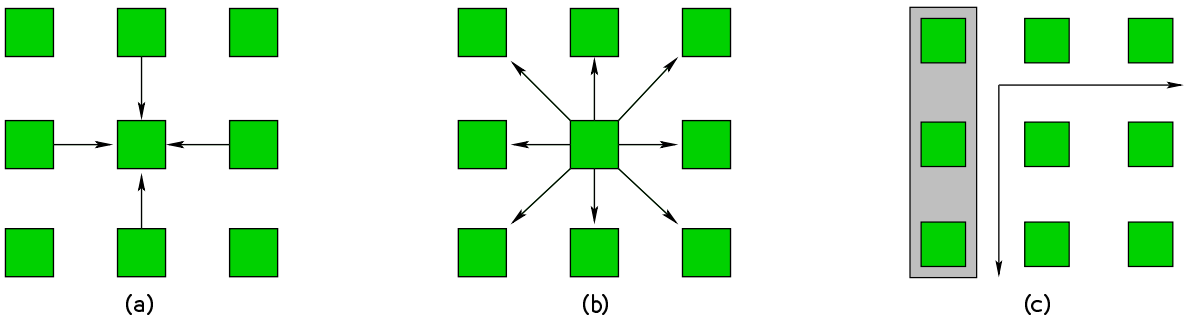

Figure 1. Schémas des accès pour Jacobi (a), LBE (b) et LU (c)

```
!$OMP PARALLEL DO
  do j = 1,n
     do i = 1,n
        A(i,j) = (B(i-1,j)+B(i+1,j)+B(i,j-1)+B(i,j+1)) * c
     end do
  end do
!$OMP END PARALLEL
```

Programme 1. Noyau du Jacobi

Une distribution `BLOCK` dans la deuxième dimension des deux tableaux `A` et `B` permet d'obtenir une bonne localité et un bon équilibrage de charge. Dans la version privée, nous avons employé une zone *shadow* pour avoir accès aux éléments situés à la frontière. Le partage des données le long de la frontière est réalisé via 2 buffers partagés de la taille d'une colonne par processeur, à part pour les processeurs 0 et

N-1 qui ont un seul buffer partagé. Ainsi le tableau effectivement partagé utilisé pour implémenter ces buffers est de taille (2*N-2) * taille d'une colonne, où 'N' est le nombre de processeurs.

4.2. Équation de Boltzmann

LBE est un code de dynamique des fluides qui résout l'équation de Boltzmann (Lattice Boltzmann Equation, LBE [9]). L'algorithme est semblable à Jacobi avec un schéma d'accès en étoile excepté que les éléments voisins sont mis à jour et non pas lus (Fig. 1(b)). Ainsi c'est un cas d'accès partagé en écriture plutôt qu'en lecture, soit un fort potentiel de prendre en défaut le fonctionnement de la mémoire partagée. En effet, il est possible d'avoir de multiples lectures sur la même ligne de cache (ou page), mais pas de multiples écritures. C'est donc un bon exemple pour analyser une faiblesse des architectures NUMA.

```
!$OMP PARALLEL
  do iter = 1, niters
     Calculate_ux_uy_p
     Collision_advection_interior
     Collision_advection_boundary
  end do
!$OMP END PARALLEL
```

```
Collision_advection_interior:
!$OMP DO
  do j = 2, Ygrid-1
     do i = 2, Xgrid-1
        f(i,0,j)   = Fn(fold(i,0,j))
        f(i+1,1,j) = Fn(fold(i,1,j))
        f(i,2,j+1) = Fn(fold(i,2,j))
        f(i-1,3,j) = Fn(fold(i,3,j))
        .....
        f(i+1,8,j-1) = Fn(fold(i,8,j)
     end do
  end do
!$OMP END DO
```

Programme 2. Algorithme LBE et noyau de Collision_advection_interior

`Collision_advection_interior` (Prog. 2) est le seul sous-programme de LBE qui possède un accès partagé aux données. Les matrices sont distribuées par `BLOCK` dans la dernière dimension pour assurer de nouveau une bonne localité et un bon équilibrage de charge. Dans la version privée, nous avons employé une zone shadow pour avoir accès aux éléments voisins et un buffer partagé de la taille de ces frontières (comme dans Jacobi) pour la mise à jour distante.

4.3. Factorisation LU

La factorisation LU (Progr. 3) est souvent employée comme une partie d'une résolution de système. Elle factorise la matrice en deux sous matrices, triangulaires supérieure et inférieure.

```
!$OMP PARALLEL
  do k = 1,n-1
!$OMP SINGLE
     lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$OMP END SINGLE
!$OMP DO
     do j = k+1, n
        lu(k+1:n,j) = lu(k+1:n,j) - lu(k,j) * lu(k+1:n,k)
     end do
!$OMP END DO
  end do
!$OMP END PARALLEL
```

Programme 3. Factorisation LU

Semblable à Jacobi, il n'y a que des accès partagés en lecture et seule la colonne courante normalisée est concernée (Fig. 1(c)). Mais pour être efficace, tous les threads doivent participer à la mise à jour presque jusqu'à la fin. D'où, pour un bon équilibrage de charge, une distribution `CYCLIC` sur la deuxième dimension est nécessaire. C'est un bon exemple où la norme OpenMP n'a aucune réponse pour un besoin si spécifique et montre donc un intérêt fort de la capacité à spécifier la distribution des données et donc la distribution des calculs associés. Dans la version privée, nous employons seulement un tableau partagé de la taille d'une colonne afin de rendre accessible la colonne normalisée à tous les processeurs.

5. Résultats expérimentaux

Pour chaque application, nous présentons les résultats et analyses obtenus à chaque fois avec l'Origin 2000 de SGI et le SP2 d'IBM. Nous terminerons par un résumé des analyses.

5.1. Jacobi

Sur l'Origin 2000, comme un processeur lit essentiellement des données locales, à l'exception d'une ligne provenant des processeurs voisins, les rendements(speedups) observés sont linéaires (et même super-linéaires) (Fig. 2(a)). Le "*First Touch*" et `DISTRIBUTE` montrent une efficacité comparable puisque, dans cet exemple, la pagination est la même après la phase d'initialisation. L'écart observé sur 64 processeurs est dû justement à cette initialisation où le système d'exploitation déplace les pages selon la politique du "*First Touch*", tandis que dans la version `DISTRIBUTE`, ce placement est réalisé lors de la compilation.

À ce jour, il est difficile de dire pourquoi le `DISTRIBUTE_RESHAPE (*,BLOCK)` donne une accélération inférieure, alors que le placement en mémoire est fondamentalement le même. Parmi les explications possibles, il y a les surcoûts induits par la réorganisation des tableaux ou par l'arithmétique des pointeurs utilisée pour accéder aux éléments. Mais cela devrait être également observé par ailleurs, comme avec LBE ou LU, ce qui n'est étonnamment pas le cas.

La version *Privée* montre des accélérations super-linéaires (75 pour 64 processeurs), principalement en raison de la privatisation des données qui impliquent un meilleur effet de cache (niveau 1) et de la moindre intrusion du système d'exploitation dans la gestion des variables partagées, puisqu'elles ne sont uniquement employées que pour mettre à jour les zones shadow après chaque itération du Jacobi, et non pas durant les calculs. Ces mises à jours sont donc vectorisées, d'où ces meilleures performances.

Avec plus de processeurs (jusqu'à 128), nous observons toujours une grande efficacité, mais qui n'est plus linéaire car il n'y a plus assez de calcul pour chaque processeur pour une matrice de taille 1024×1024 .

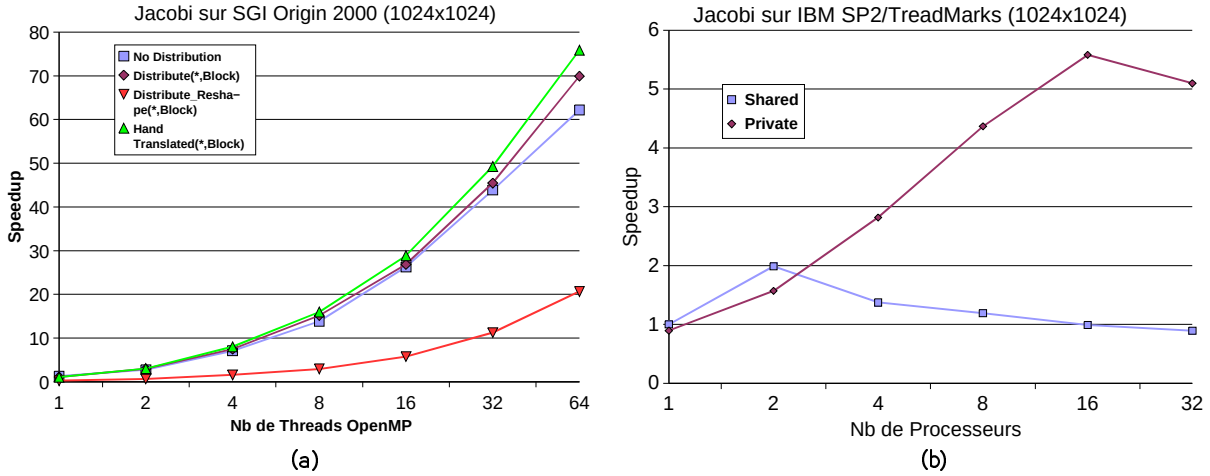

Figure 2. Accélération du Jacobi sur SGI O2000 (a) et IBM SP2 (b) (1024×1024 en double précision)

Avec TreadMarks sur IBM SP2, les performances pour la version avec un tableau partagé sont très faibles et ce dès 2 processeurs (Fig. 2(b)). En fait, nous remarquons un fort ralentissement avec plus de 8 processeurs. Pour le cas privé, nous obtenons une accélération quasi linéaire jusqu'à 16 processeurs. Avec 32 processeurs, la courbe s'affaisse. Le faible rendement pour la version partagée est attribué au coût élevé induit par le maintient de la cohérence sur un grand nombre de pages partagées, contrairement à la version privée qui possède un nombre plus faible de pages partagées (seulement un buffer).

5.2. Équation de Boltzmann

Sur l'Origin 2000, nous pouvons voir cette fois que toutes les versions employant l'extension SGI d'OpenMP ou le "*First Touch*" ont des résultats quasi identiques, avec un léger avantage pour `DISTRIBUTE_RESHAPE` (Fig. 3(a)). Cette fois, les rendements sont sous-linéaires, car chaque processeur doit accéder à la mémoire du voisin en écriture, contrairement à Jacobi. Ce partage en écriture prend en défaut le support du système d'exploitation pour la mémoire partagée, avec un effet de ping-pong des lignes de cache associées. L'architecture cc-NUMA peut également être rendue responsable, car, pour 16 ou 32 processeurs, l'accès

à une donnée non-locale peut se faire en traversant 2 noeuds, obtenant donc de plus long temps d'accès.

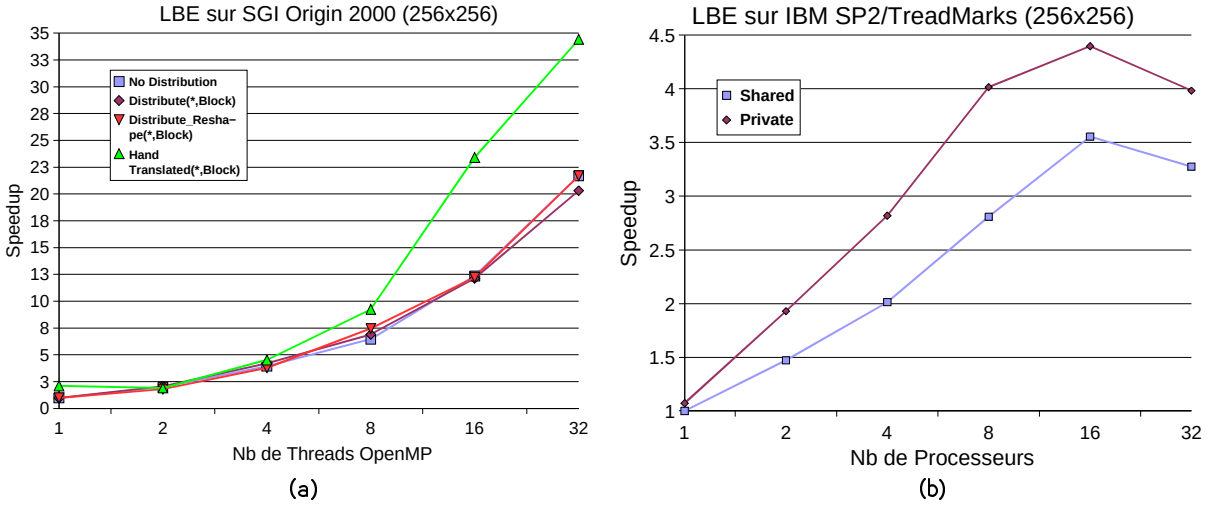

Figure 3. Accélération de LBE sur SGI O2000 (a) et IBM SP2 (b) (256 × 256)

La version *Privée* de LBE montre une amélioration notable de l'accélération pour 16 et 32 processeurs. D'une part, il y a une augmentation de la localité des données (effet de cache) et d'autre part la synchronisation est faite seulement à la fin de chaque itération. Tous les processeurs font donc leur calcul seulement sur des données privées jusqu'à la fin de l'itération (vectorisation des messages).

Avec TreadMarks sur IBM SP2, nous observons une moindre amélioration de la version privée comparée à la version partagée (Fig. 3(b)). Les deux versions ont des accélérations raisonnables (3.5 sur 16 processeurs pour la version partagée, 4.5 pour la version privée) puisque la quantité de mémoire partagée n'est pas trop importante, d'où un coût faible pour le maintien de la cohérence sur les processeurs. Néanmoins la privatisation permet d'obtenir une meilleure efficacité puisque le nombre de pages maintenues y est moindre que dans la version partagée (seuls les buffers sont partagés). Avec plus de 16 processeurs, les efficacités diminuent en raison d'un manque de calculs par rapport aux communications requises.

5.3. Factorisation LU

Sur l'Origin 2000, toutes les versions, après 16 processeurs (Fig. 4(a)), ont un rendement sous-linéaire en raison, principalement, du manque de travail sur chaque processeur (ratio calculs/communications élevé).

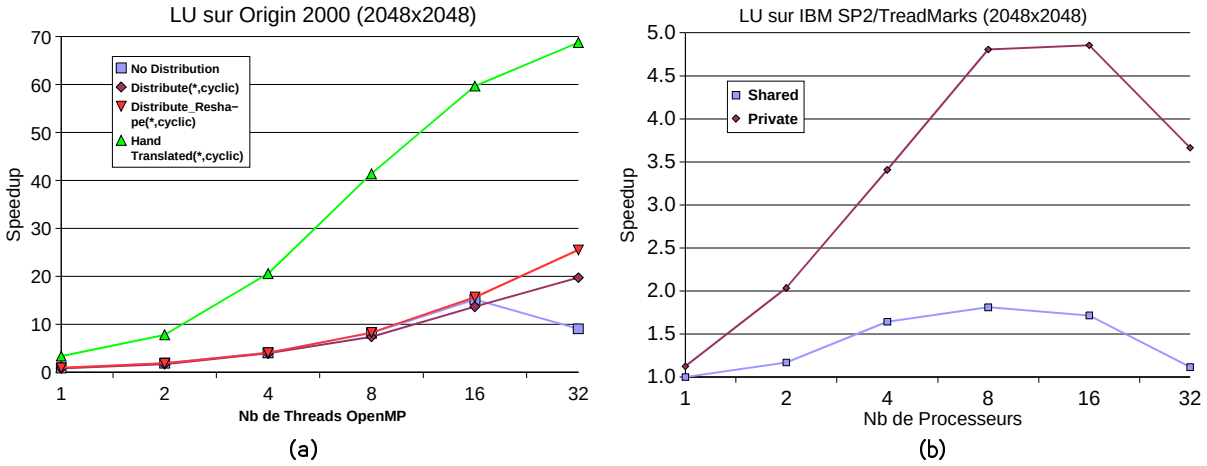

Figure 4. Accélération pour LU sur SGI O2000 (a) et IBM SP2 (b) (2048 × 2048)

Si nous considérons la version OpenMP sans distribution, son mauvais rendement est dû à une mauvaise répartition sur les 32 processeurs. En effet, par défaut, la répartition initiale (résultant du "*First Touch*")

ressemblera à une distribution `BLOCK` qui n'est pas appropriée pour pour l'équilibrage de charge avec LU. Les versions `DISTRIBUTE` et `DISTRIBUTE_RESHAPE` sont donc meilleures car la distribution reflète le besoin pour l'équilibrage de charge.
Dans la version *Privée*, la seule partie présentant une synchronisation entre les processeurs est lorsque chaque processeur recopie "la colonne normalisée partagée" dans son buffer privé. Le reste du calcul est alors parfaitement parallèle où chaque processeur fait ses calculs sur ses propres données sans aucune donnée partagée. Le rendement super-linéaire vient de nouveau de la privatisation des données, visant à la fois à un meilleur effet de cache et à une vectorisation des mises à jour des buffers partagés.

Avec TreadMarks sur IBM SP2, pour la version partagée, nous remarquons de faibles accélérations jusqu'à 16 processeurs maximum (Fig. 4(b)), supérieures à celles obtenues avec Jacobi, principalement en raison du fait que, dans LU, seule une colonne par itération est accèdée de manière partagée. De là, le nombre de pages à mettre à jour est moindre que dans Jacobi (2*N - 2 lignes par itération).
Pour la version *Privée*, nous remarquons des accélérations quasi linéaires jusqu'à 8 processeurs. Au-delà, la baisse de rendement peut être attribuée à la facon dont, dans LU, comme l'exécution progresse, la quantité de calcul à réaliser par itération diminue. Cet effet est d'autant plus marqué que le nombre de processeurs est grand, d'où le surcoût dû à la synchronisation prend le pas sur les calculs.

5.4. Résumé des expérimentations sur les deux plate-formes

Sur l'Origin 2000 de SGI, nous avons observé que prendre soin de la distribution des données (et des calculs selon les données) peut radicalement augmenter l'efficacité des applications, et ce, même si l'on admet que la machine possède un système de mémoire partagée efficace. L'architecture cc-NUMA est responsable de ce résultat. Nous observons par exemple que le "*First Touch*" n'est pas toujours suffisant pour obtenir une bonne efficacité, comme avec l'application LU.
Nous observons aussi que la privatisation des tableaux (en les placant explicitement sur la mémoire locale de chaque processeur) permet de meilleurs rendements, voire même super-linéaire. Ceci est principalement dû à deux facteurs. D'abord, la privatisation place les données dans le cache de niveau 2 dans le système SGI, donc ils sont plus proches du processeur et permettent de bénéficier des effets de caches du niveau 1 qui est souvent responsable d'accélération super-linéaire. Deuxièmement, cette privatisation implique un style de programmation qui a tendance à placer toutes les communications à l'extérieur des calculs, soit un processus de vectorisation des communications. Ce second effet, bien connu dans la recherche concernant HPF [8] est aussi responsable des meilleures performances.
Cependant, il y a toujours plusieurs problèmes à résoudre quant à l'utilisation de cette approche. Par exemple, à ce jour, la version privée sur l'Origin 2000 est limitée par la quantité de mémoire disponible sur la pile utilisée pour la mise en oeuvre des tableaux privatisés. Mais avec la seconde version d'OpenMP et les changements architecturaux attendus sur l'Origin 3000 de SGI, cette limitation ne devrait pas persister.

Avec la bibliothèque SDSM TreadMarks, le nombre total des pages maintenues pour les versions partagées est beaucoup plus grand que pour les versions privées. Par conséquent, le surcoût impliqué par le maintient de la cohérence de la mémoire partagée pour une version partagée est beaucoup plus grand que pour une version privée. De plus, pour cette dernière, comme le nombre de pages partagées est faible, le coût de marquage dans le système TreadMarks est lui aussi plus bas [7]. Ce faible coût est reflété par les améliorations de rendement observées. En effet, la quantité d'informations échangées pour la cohérence affecte aussi le temps passé dans les barrières et synchronisations, principalement en raison de la version centralisée du système implémenté dans TreadMarks.

Pour récapituler, nous observons dans les deux architectures NUMA, que ce soit cc-NUMA ou basée sur un SDSM, que la localité de données est un facteur important pour obtenir une bonne efficacité. Et nous voyons que la privatisation est une facon d'obtenir ce résultat.

6. Travaux relatifs

OpenMP[3] est un constitué de directives permettant de décrire le parallélisme de manière simple. Ces directives ont été concu pour l'exécution de programmes sur des machines parallèles à mémoire partagée et repose essentiellement sur la notion de boucles parallèles et de variables partagées ou privées. Au

contraire d'HPF[8], OpenMP ne propose aucune directive permettant d'orienter la distribution des données car OpenMP considère un accès uniforme de tous les processeurs à la mémoire partagée.
De nombreuses études ont déjà porté sur l'extension d'OpenMP à des architectures à accès non uniforme à la mémoire. Certains ont considéré des modèles de programmation mixtes pour les grappes de noeuds SMP, utilisant à la fois MPI et OpenMP pour paralléliser les applications [19, 4]. Il est généralement admis que l'efficacité de cette approche dépend fortement des propriétés sous-jacentes de l'application étudiée. Néanmoins, il apparaît également que ce style de programmation n'est pas aisé pour le programeur puisqu'il doit faire face à la programmation SMPD qu'impose MPI [6].
Nous pensons qu'il est possible avec un minimum d'extensions à OpenMP d'obtenir un langage à base de directives qui facilite la programmation et permette d'obtenir de bonnes performances sur de telles architectures. Il existe déjà quelques implémentations de compilateurs OpenMP dédiés pour des grappes de noeuds SMP comme le compilateur Omni OpenMP [17, 16], mais ce dernier n'inclue pas de directives de placement des données et repose essentiellement sur un SDSM pour implémenter la mémoire partagée le long des noeuds.

Comme nous l'avons déjà présenté dans cet article, SGI a considéré le placement des données et des calculs comme étant dans certains cas un point crucial pour obtenir de bonnes performances sur des systèmes cc-NUMA comme l'Origin 2000 [10] afin de réduire les écarts entre les temps d'accès à la mémoire distante et locale [18]. Nos expérimentations montrent que l'implémentation choisie par SGI pourrait sans doute bénéficier de quelques améliorations, notamment par la prise en compte de la directive `SHADOW`.
Compaq, pour ses AlphaServers de type cc-NUMA, a également introduit des extensions inspirées directement de HPF dans leur compilateur OpenMP [2]. Ces extensions couvrent un plus large spectre que celles de SGI, bien que très semblables - notamment pour ce qui est d'une distribution basée sur la pagination et une autre basée sur les éléments eux-mêmes -, en introduisant également les alignements entre les données et la possibilité de restreindre la distribution à un sous-ensemble de processeurs.
L'inconvénient de l'approche de Compaq est de proposer un langage sans doute trop riche pour pouvoir être facilement porté sur d'autres types d'architectures, et pose donc le problème de la portabilité des codes écrits pour les architectures Compaq vers d'autres architectures.

Enfin, Portland Group Inc. (PGI) propose également des extensions au langage OpenMP dans un langage nommé Distributed OpenMP (DoMP) [14, 11]. Ces extensions concernent principalement les grappes de noeuds à mémoire partagée, incluant aussi les cc-NUMA et les architectures à mémoire distribuée. Ils proposent également les directives `DISTRIBUTE` et `ON HOME` pour distribuer les données et les calculs, mais cette fois ci sur les mémoires des noeuds et non des processeurs de facon à refléter la structure de la mémoire sur les grappes de noeuds.
Cette approche, différente des deux premières, utilise comme modèle d'exécution celui de HPF en dehors de toute région `PARALLEL`, et à l'intérieur d'une région `PARALLEL`, celui d'un OpenMP distribué. Pour simplifier, un thread OpenMP distribué est en fait constitué de plusieurs threads répartis sur les noeuds, communiquant entre eux pour assurer les accès aux mémoires distantes.

7. Conclusion

Toutes ces approches proposent diverses extensions et interprétations afin de prendre en compte l'aspect distribué de la mémoire. Toutes ces approches sont attractives mais empêchent l'utilisateur de réaliser une version portable de son code puisque chacune d'entre elles constituent des extensions propriétaires. Notre but est donc de définir un ensemble minimum d'extensions qui seront vraiment utiles et aussi portables que possible.
Nous montrons dans cet article qu'un premier sous-ensemble de directives serait constitué des directives de distribution des données (`DISTRIBUTE`) et des calculs en fonction des données (`ON HOME`), mais aussi de la directive `SHADOW` permettant de considérer les données accédées à la frontière de ces zones distribuées. Les expérimentations ont montré que les performances étaient dramatiquement améliorées en utilisant une privatisation des données, et ce aussi bien pour des architectures cc-NUMA telle le Origin 2000 de SGI que des architectures a mémoire partagée répartie logicielle comme l'IBM SP2 avec la bibliothèque de support pour une mémoire partagée TreadMarks.

D'autres études sont encore nécessaires, et notamment pour mieux prendre en compte les architectures du type grappes de noeuds à mémoire partagée, en raison de l'écart important entre le temps d'accès aux données locales et aux données distantes via le réseau. Les techniques utilisées devront certainement hériter des études passées portant sur HPF, et notamment sur les capacités à analyser et optimiser les communications entre les noeuds. Même si le compilateur s'appuie sur un support SDSM pour implémenter le partage de la mémoire sur l'ensemble des noeuds, il demeure selon nous important de conserver la notion de localité des données de facon à limiter le surcoût induit par ce support.

Bibliographie

1. C. Amza, A. Cox, et et al. Treadmarks: Shared memory computing on networks of workstations. IEEE Computer, 29(2):18–28, Février 1996.
2. J. Bircsak, P. Craig, R. Crowel, J. Harris, C.A. Nelson, et C.D. Offner. Extending OpenMP For NUMA Machines: The Language. Dans *WOMPAT 2000, Workshop on OpenMP Applications and Tools*, San Diego Supercomputer Center, San Diego, California, Juillet 2000.
3. OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0, Novembre 2000.
4. F. Cappello et D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. Dans *SC2000, Supercomputing*, Dallas, Texas, USA, Novembre 2000.
5. R. Chandra, D.K. Chen, R. Cox, D.E. Maydan, N. Nedeljkovic, et J.M. Anderso. Data Distribution Support on Distributed Shared Memory Multiprocessors. Dans *Conference on Programming Language Design and Implementation*, 1997.
6. B. Chapman, J. Merlin, D. Pritchard, F. Bodin, Y. Mevel, T. Sorevik, et L. Hill. Tools for Development of Programs for a Cluster of Shared Memory Multiprocessors. Dans *PDPTA'99*, Las Vegas, USA, 1999.
7. E. de Lara, Y.C. Hu, H. Lu, A.L. Cox, et W. Zwaenepoel. The Effect of Memory Contention on the Scalability of Page-based Software Distributed Shared Memory Systems. Dans *Proceedings of Languages, Compilers, and Runtimes for Scalable Computing*, Mai 2000.
8. HPF Forum. High Performance Fortran language specification, Version 2.0, Janvier 1997.
9. X. He et L. Luo. Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. Phys. Rev. Lett. E,56(6) 6811, 1997.
10. Silicon Graphics Inc. MIPSPro Fortran 90 Commands and Directives Reference Manual. Document number 007-3696-003. Search keyword MIPSPro Fortran 90 on http://techpubs.sgi.com/library/.
11. J. Merlin (The Portland Group, Inc.). Distributed OpenMP: Extensions to OpenMP for SMP Clusters. Dans *EWOMP 2000, European Workshop on OpenMP*, Edimburgh, Scotland, U.K., Septembre 2000.
12. P. Keleher, A. Cox, et W. Zwaenepoel. Lazy release consistency for software distributed shared memory. Dans *19th Annual International Symposium on Computer Architecture*, pages 12–21, Mai 1992.
13. P. Keleher, S. Dwarkadas, A. Cox, et W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. Dans *Winter Usenix Conference*, pages 115–131, Janvier 1994.
14. M. Leair, J. Merlin, S. Nakamoto, V. Schuster, et M. Wolfe. Distributed OMP - A Programming Model For SMP Clusters. Dans *CPC2000, Compilers for Parallel Computers*, Aussois, France, Janvier 2000.
15. Concurrent Programming with TreadMarks. TreadMarks users Manual. http://www.cs.rice.edu/ willy/papers/doc.ps.gz.
16. M. Sato, H. Harada, et Y. Ishikawa. OpenMP compiler for a Software Distributed Shared Memory System SCASH. Dans *WOMPAT 2000*, San Diego, Juillet 2000.
17. M. Sato, S. Satoh, K. Kusano, et Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. Dans *EWOMP'99, European Workshop on OpenMP*, pages 32–39, Lund, Sweden, Septembre 1999.
18. S. Seidel. Access Delays Related to the Main Memory Heirarchy on the SGI Origin2000. Dans *Third European CRAY-SGI Workshop*, Paris, France, Septembre 1997.
19. L.A. Smith et J.M. Bull. Development of Mixed Mode MPI/OpenMP Applications. Dans *WOMPAT 2000*, San Diego, Juillet 2000.