De l'exécution structurée de programmes OpenMP sur architectures hiérarchiques

François Broquedis

Mémoire de MASTER RECHERCHE

présenté par

François Broquedis

De l'exécution structurée de programmes OPENMP sur architectures hiérarchiques

Encadrement : Brice Goglin et Raymond Namyst

Février – Juin 2007

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
Université Bordeaux 1

Table des matières

Introduction

Depuis le Mark I, inventé par Howard Aiken en 1937, permettant de calculer cinq fois plus vite que l'homme, et mesurant 17 mètres de long sur 2 mètres de haut, les ordinateurs n'ont cessé d'être miniaturisés et leur puissance de calcul augmentée, la densité d'intégration des transistors et l'augmentation de fréquence allant de pair.

Bien que la finesse de gravure continue de progresser, réduisant de plus en plus la taille des circuits, des contraintes physiques empêchent les constructeurs de dépasser les fréquences des processeurs actuels du marché. Après avoir essayé plusieurs combinaisons de composants, en utilisant notamment l'espace offert par une gravure plus fine pour ajouter des unités fonctionnelles ou des niveaux de caches supplémentaires, les architectes ont récemment renoncé à complexifier les architectures monoprocesseur, et utilisent désormais cet espace pour dupliquer les processeurs et en graver plusieurs côte à côte sur une même puce.

Ces architectures *multicœurs* envahissent le marché et leur puissance de calcul se compte désormais en nombre de cœurs intégrés. Parallèlement à cette évolution, les architectures de machines se sont elles aussi complexifiées. Les constructeurs proposent des architectures basées sur plusieurs processeurs ou encore distribuent la mémoire, auparavant centralisée, à plusieurs endroits de la machine, rompant l'isométrie entre les différentes parties de la mémoire et les processeurs qui tentent d'y accéder. Il n'est donc plus rare de trouver des machines de calcul composées de plusieurs blocs, eux-mêmes constitués de plusieurs processeurs, eux-mêmes composés de plusieurs cœurs. Cette hiérarchisation des architectures actuelles complexifie la répartition du travail sur les nombreuses unités de calcul qu'elles comportent et fait apparaître de nouvelles problématiques, comme la prise en compte de l'éloignement physique de flots d'exécution travaillant sur les mêmes données.

Classiquement, la communauté distingue deux principales approches pour programmer ce type d'ordinateur. Le programmeur peut insérer le code nécessaire à la gestion des flots d'exécution et les faire exécuter par les différents processeurs disponibles. On parle alors de programmation parallèle *explicite*. Ces flots d'exécution peuvent communiquer soit par envoi de messages, soit plus efficacement par partage direct de mémoire. L'API POSIX Threads est dans ce dernier cas la plus fréquemment utilisée et des dizaines d'implémentations ont vu le jour telles la Native POSIX Threads Library (NPTL) de LINUX, THR (la bibliothèque de SOLARIS) ou encore MARCEL, la bibliothèque de threads de la suite logicielle PM2 développée dans l'équipe-projet RUNTIME. Une particularité de la bibliothèque MARCEL est de présenter, en plus de l'interface POSIX, une interface étendue permettant, entre autre, de structurer de façon arborescente un ensemble de flots de calcul afin qu'un ordonnanceur spécialisé optimise leur exécution à sa façon.

La programmation parallèle *implicite* repose quant à elle sur l'emploi de langages de haut niveau masquant la plupart des opérations intervenant dans le contrôle du parallélisme,

comme la création, terminaison des flots de calculs ou encore leur synchronisation. Dans ce cadre, OPENMP [1] est un standard dont les implémentations permettent de paralléliser des programmes C, C++ ou FORTRAN annotés sans en modifier la structure (dans l'idéal). Son support d'exécution est capable d'adapter le nombre de threads créés en fonction du nombre de cœurs présents, apportant ainsi un minimum de portabilité. Depuis son apparition en 1997, OPENMP n'a pas réussi à s'imposer en tant qu'outil de référence pour les ordinateurs à mémoire partagée. Ce constat s'explique par le fait que ce standard n'apporte qu'une couche de portabilité et d'abstraction au-dessus des processus légers au détriment de beaucoup de possibilités d'optimisations "à la main". Ou encore que l'approche «envoi de message» est plus universelle car elle permet de programmer les machines multiprocesseurs comme on programme des grappes par communication explicite, ce qui peut paraître techniquement parlant regrettable mais économiquement compréhensible. Aujourd'hui, la donne change. La démocratisation et la densification des architectures parallèles apportent un nouvel élan à OPENMP. Par exemple, les compilateurs les plus populaires (GCC[1] et ICC[2]) l'intègrent dans leur dernière version.

Cependant peu d'outils, de langages et de compilateurs prennent réellement en compte la hiérarchisation accrue des ordinateurs. L'objectif de ce mémoire est de jeter les bases d'un environnement de programmation OPENMP performant, particulièrement bien adapté aux architectures parallèles hiérarchiques. Le choix d'OPENMP est pertinent, non seulement parce qu'OPENMP connaît aujourd'hui un regain d'intérêt avec l'explosion du nombre de cœurs par machines, mais aussi parce qu'un compilateur OPENMP est très bien placé pour extraire, transmettre et exploiter des informations sur la structure du parallélisme du programme considéré.

Concrètement, l'idée est de coupler OPENMP et la bibliothèque MARCEL, ce dernier exploitant les informations sur le parallélisme de l'application obtenues par un compilateur OPENMP. Dans un premier temps, il s'agit d'instrumenter le code généré/utilisé par un compilateur OPENMP en y insérant des appels à la bibliothèque MARCEL afin que le programme structure dynamiquement ses flots parallèles de calcul. Puis, au sein de la bibliothèque MARCEL, il s'agit de développer un nouvel ordonnanceur de façon à répartir efficacement les flots de calculs sur une machine hiérarchique en tenant compte à la fois de la structure arborescente des flots de calcul et de la topologie de la machine utilisée. Ces deux étapes accomplies, on obtient alors une solution alliant la portabilité d'OPENMP à la souplesse et la précision des ordonnanceurs de MARCEL, permettant ainsi de programmer de façon simple des applications parallèles efficaces sur des machines de calcul hiérarchiques.

Ce mémoire présente donc la démarche et les expériences que nous avons entreprises pour améliorer l'exécution des programmes OPENMP, et plus spécifiquement ceux compilés par GCC, base de notre plateforme expérimentale. Après avoir rappelé les clefs de l'architecture et de la programmation parallèles, nous détaillons notre démarche pour rendre efficace l'ordonnancement des programmes OPENMP puis présentons notre implémentation avant d'en évaluer le comportement et les performances.

[1] GNU Compiler Collection
[2] Intel C Compiler

Chapitre 1

Programmation des machines multiprocesseurs contemporaines

Ce chapitre présente les évolutions, en termes d'architecture, des machines de calcul contemporaines et les outils dont nous disposons pour les programmer.

1.1 Des architectures parallèles fortement hiérarchisées

Encore très récemment, les constructeurs de processeurs pouvaient compter sur l'augmentation régulière de la fréquence pour améliorer les performances de leurs produits. Mais depuis 2005, cette montée en fréquence se heurte à des problèmes physiques de dissipation thermique, en dépit de la taille plus faible des composants. Des innovations architecturales, toujours plus complexes, ont depuis permis de continuer à gagner en puissance de calcul, en intégrant du parallélisme depuis l'agencement même de la machine, jusque dans les processeurs. Cette section présente ces innovations. Le lecteur pourra se reporter à l'ouvrage de référence d'Hennessy et Patterson [8] sur le sujet.

1.1.1 Architectures multiprocesseurs

Bien que l'augmentation de la fréquence des processeurs ait subit un coup d'arrêt[1], la puissance de calcul des machines contemporaines ne cesse d'augmenter, grâce à l'apparition de nouvelles architectures permettant la multiplication des entités de calcul disponibles dans un ordinateur, autorisant par exemple l'exécution en parallèle de plusieurs parties indépendantes d'un même processus. Nous présentons ici les deux architectures parallèles les plus couramment utilisées lors de la conception de machines de calcul.

Des architectures monoprocesseur aux architectures SMP

Une architecture monoprocesseur se compose d'un unique processeur comportant une mémoire cache, très rapide mais de capacité réduite. Ce processeur accède à la mémoire vive

[1] IBM a cependant annoncé le 22 mai 2007 un processeur POWER6 cadencé à 4,7 GHz et espère atteindre 6 GHz à terme.

de la machine, plus lente mais disponible en plus grande quantité, via un bus mémoire. L'architecture *Symmetric Multiprocessing* (SMP) est une évolution directe des architectures monoprocesseur. Pour augmenter la puissance de la machine, on connecte plusieurs processeurs identiques au même bus mémoire comme on peut le voir sur la figure 1.1. Ces processeurs disposent chacun de leur propre mémoire cache, et pourront travailler en parallèle pour accélérer l'exécution de processus, ou en exécuter plusieurs de manière concurrente.

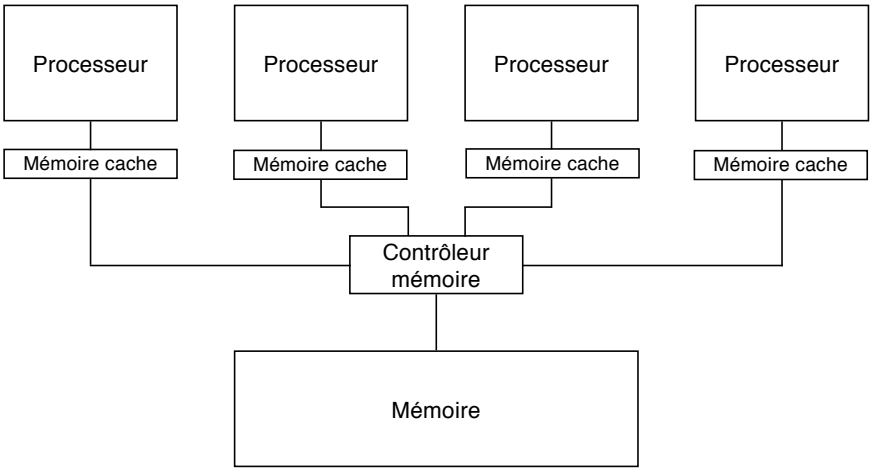


FIG. 1.1 – Exemple d'architecture SMP.

Peu onéreuse, cette architecture passe cependant mal à l'échelle. Le protocole d'accès au bus mémoire interdit le recouvrement des communications. Autrement dit, ce bus ne peut être utilisé que par un seul processeur à la fois. Ainsi, plus on augmente le nombre de processeurs, plus la contention au niveau du bus est potentiellement importante. En pratique, les architectures SMP comportent rarement plus de 4 processeurs.

Des architectures SMP aux architectures NUMA

Afin de contourner le problème des contentions mémoire inhérent aux architectures SMP, et pouvoir ainsi créer des machines de plusieurs centaines de processeurs, il est nécessaire de renoncer au principe de mémoire monolithique uniforme. Les architectures NUMA (*Non-Uniform Memory Access*) abolissent la mémoire centralisée des architectures monoprocesseurs et SMP, pour l'éclater en plusieurs morceaux, accessibles plus ou moins rapidement selon la proximité géographique des processeurs. Concrètement, une machine NUMA est composée d'une interconnexion de blocs basés sur l'architecture SMP. Chaque bloc est constitué d'un nombre limité de processeurs et de leurs mémoires cache, connectés à un banc mémoire local. Un dispositif matériel permet à ces bancs de mémoire distribués d'apparaître au programmeur comme une unique mémoire globale. Les performances du réseau interconnectant les blocs sont limitées. C'est pourquoi un accès par un processeur à la mémoire de son bloc est plus rapide qu'un accès à la mémoire d'un bloc distant. On parle d'architecture à accès mémoire non uniforme. Le rapport entre les temps d'accès distant et local est appelé facteur NUMA. Plus celui-ci est proche de 1, plus on s'approche du modèle d'une machine accédant uniformément à une mémoire globale. Aujourd'hui, on mesure un facteur NUMA compris entre 1,1 et 3 selon les architectures.

1.1.2 Du parallélisme à l'intérieur du CPU

Malgré la stagnation constatée en terme de fréquence, les processeurs continuent tout de même d'évoluer. La finesse de gravure progresse encore de nos jours. La miniaturisation qui en résulte offre plus d'espace sur une puce, permettant aux architectes de revoir l'agencement des composants et d'en ajouter de nouveaux, développant ainsi de nouvelles architectures de processeurs potentiellement plus puissantes. Nous présentons cette évolution dans cette partie.

Le parallélisme d'instruction (Instruction Level Parallism, ILP)

Depuis 1985, l'architecture de tous les processeurs courants est basée sur la technique dite du *pipeline* qui consiste à diviser l'exécution d'une instruction en une séquence d'étapes, chacune des étapes étant réalisée par une unité fonctionnelle dédiée et matériellement indépendante des autres unités. À l'image du travail à la chaîne, il est alors possible de traiter en parallèle plusieurs instructions issues d'un même flot en faisant travailler simultanément les différentes unités fonctionnelles sur différentes instructions. Certaines étapes étant plus complexes que d'autres ou certaines instructions réclamant des traitements spécifiques (opérations entières ou opérations flottantes, par exemple), on a multiplié les unités de certaines étapes permettant l'exécution simultanée de plusieurs instructions : on parle alors d'architecture superscalaire.

La principale limite de la technique du pipeline est l'inactivité forcée des unités fonctionnelles causée par des dépendances entre les instructions (deux instructions dépendantes ne peuvent pas se suivre de trop près) et par des branchements conditionnels (on ne sait pas déterminer assez tôt la prochaine instruction à exécuter).

Pour améliorer l'efficacité des pipelines, des techniques avancées ont été introduites telles que le ré-ordonnancement dynamique des instructions, le renommage à la volée de registres ou encore la prédiction de branchement. Désormais, une bonne part du silicium d'un processeur est consacrée à ces techniques. Cependant, elles sont inefficaces lorsque les instructions sont fortement dépendantes les unes des autres ou lorsque des défauts de cache apparaissent.

Le parallélisme de flot (Thread Level Parallelism, TLP)

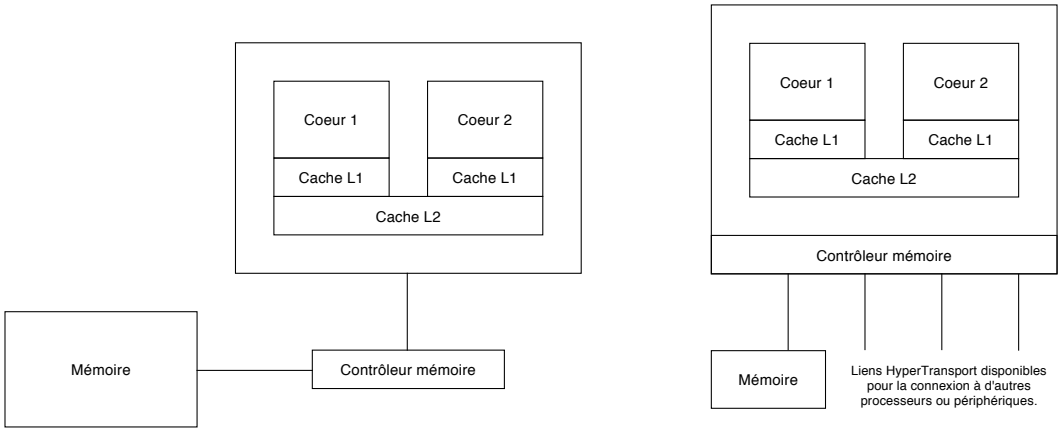
Afin d'optimiser encore plus le taux d'occupation des unités fonctionnelles, les architectes ont rendu possible le traitement de plusieurs flots d'instructions (indépendants) de manière concurrente par un même pipeline. Pour ce faire, on introduit autant de jeux de registres qu'on veut traiter de flots d'instructions en parallèle. On parle ici de processeur *logique* et d'architecture SMT, pour *Simultaneous MultiThreading* (HYPERTHREADING chez INTEL). Ainsi, une architecture SMT composée de deux processeurs logiques basée sur un seul pipeline sera vue comme une machine biprocesseur par le système d'exploitation.

La miniaturisation des composants est telle qu'il est aujourd'hui possible de graver plusieurs processeurs sur une même puce, formant ainsi ce que l'on appelle un processeur multicœurs. Ces processeurs ont déjà envahi le marché grand public et les principaux fondeurs que sont AMD, IBM, INTEL, ou encore SUN, proposent chacun leur façon d'agencer ces cœurs à l'intérieur d'une puce. En voici deux illustrations :

L'approche INTEL Le processeur INTEL Core 2 Duo comporte deux cœurs, disposant chacun de leur propre mémoire cache. Un cache commun de niveau 2 permet une meilleure communication entre les cœurs. La puce est connectée à la mémoire vive par un contrôleur (*chipset*) comme on peut le voir sur la figure 1.2(a).

L'approche AMD Le processeur Dual-Core AMD Opteron comporte aussi deux cœurs qui disposent de leur propre mémoire cache. La particularité de cette architecture réside dans la connectivité du processeur. En effet, une puce AMD Opteron possède plusieurs liens HYPERTRANSPORT utilisés pour les connexions avec les autres processeurs ou les périphériques (figure 1.2(b)). Parmi ces liens, un est toujours connecté à un banc mémoire attaché à la puce. Les autres peuvent être reliés à d'autres processeurs Opteron, permettant ainsi la conception d'un processeur dont l'architecture peut être vue comme celle d'une machine NUMA.

Il faut noter toutefois qu'INTEL proposera une technologie d'interconnexion proche de l'HYPERTRANSPORT d'AMD dans sa prochaine génération de processeurs.



(a) Architecture du processeur Intel Core 2 Duo

(b) Architecture du processeur AMD Dual-Core Opteron

FIG. 1.2 – Deux approches différentes d'architectures multicoeurs.

1.1.3 Tendances architecturales

Les architectures des machines contemporaines se complexifient. Les constructeurs sont désormais capables de produire des machines constituées d'une interconnexion, plus ou moins hiérarchique, de blocs de plusieurs processeurs agencés en parallèle, dont les architectures peuvent être multicœurs et SMT, comme l'illustre la figure 1.3. Ces agencements font apparaître de nouvelles contraintes à prendre en compte lorsqu'on programme une application parallèle, comme le placement respectifs des données et des flots de calcul sur une machine NUMA, les temps d'accès aux différentes mémoires, ou encore la répartition efficace des processus légers pour occuper toute la machine. Il faut donc revoir les environnements de programmation voire même les techniques de programmation pour tirer parti de ces architectures.

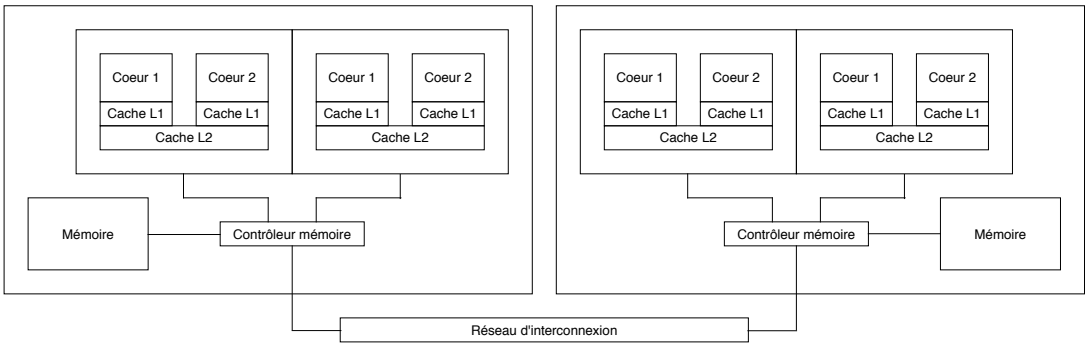


FIG. 1.3 – Une architecture de machine NUMA hiérarchique à 4 processeurs bi-cœurs.

1.2 Comment programme-t-on ces machines aujourd'hui ?

Hormis la remarquable exception de la parallélisation automatique de boucles imbriquées [4], le programmeur doit indiquer, lors de la conception de son application, quelles portions de code pourront être exécutées de façon concurrente sur les différents processeurs de la machine. Pour ce faire, il peut utiliser des bibliothèques spécialisées pour créer explicitement des processus légers ou utiliser un langage masquant certaines opérations de la parallélisation d'un programme. Cette section présente ces deux approches.

1.2.1 L'approche bibliothèque de processus légers

Les bibliothèques de processus légers fournissent des méthodes de gestion des threads, et permettent la création explicite de tâches dans les programmes. La gestion de ces dernières est souvent complexe et fastidieuse, mais cette approche permet d'obtenir les meilleures performances pour un problème donné. Elle autorise en effet un grand nombre d'optimisations qui, pour être portables efficacement demandent cependant un réel savoir-faire et un investissement conséquent en temps de développement. Ces bibliothèques proposent aussi plusieurs modes d'ordonnancements afin d'orchestrer l'exécution de ces tâches. Nous présentons dans cette partie les trois grands types de bibliothèques existantes. Le lecteur pourra se référer à la thèse de Vincent Danjean [3] pour une description plus approfondie.

Les différents types de bibliothèques

Les processus légers peuvent être gérés entièrement en espace utilisateur, au sein même du noyau (en espace protégé), ou encore de façon mixte. Ce choix de conception, fondamental, a des répercussions importantes sur de nombreuses caractéristiques et propriétés de la bibliothèque.

Les bibliothèques de niveau utilisateur Dans le cas d'une bibliothèque de niveau utilisateur, les processus légers sont gérés par un ordonnanceur opérant intégralement en espace utilisateur, à l'intérieur même du processus créant des threads.

Les primitives de gestion des processus légers sont performantes, puisqu'elles ne nécessitent aucun appel système. Cette approche permet d'adapter le comportement de l'ordonnan-

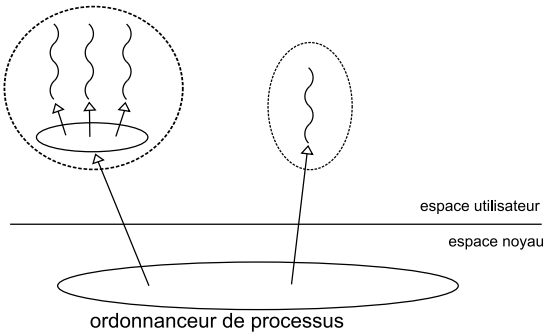


FIG. 1.4 – Un ordonnanceur de niveau utilisateur.

ceur aux besoins de l'application, autorisant de nouvelles fonctionnalités comme la migration de threads, particulièrement utile pour de l'équilibrage de charge. De plus, les primitives de création et de gestion de threads sont si peu coûteuses qu'il est possible d'en créer plusieurs centaines sans écrouler la machine. En revanche, une telle approche n'autorise pas plusieurs threads d'un même processus à s'exécuter sur des processeurs différents, puisque l'ordonnanceur système sous-jacent n'a pas connaissance de ces threads et n'est capable d'ordonnancer que des processus lourds, de son point de vue. La gestion des entrées/sorties devient elle aussi un problème, puisqu'un thread appelant une fonction potentiellement bloquante, comme une écriture sur le disque, par exemple, provoque le passage de l'état du processus entier dont il fait partie de «prêt» à «bloqué», interdisant de cette façon toute exécution d'autres threads prêts de ce même processus.

Les bibliothèques de niveau noyau Dans une bibliothèque de niveau noyau, les processus légers sont généralement gérés par l'ordonnanceur de processus du système, en espace protégé. Ils sont appelés threads lourds, et correspondent physiquement à des processus dont la mémoire serait partagée avec d'autres. Du point de vue de l'ordonnanceur, un thread lourd équivaut à un processus.

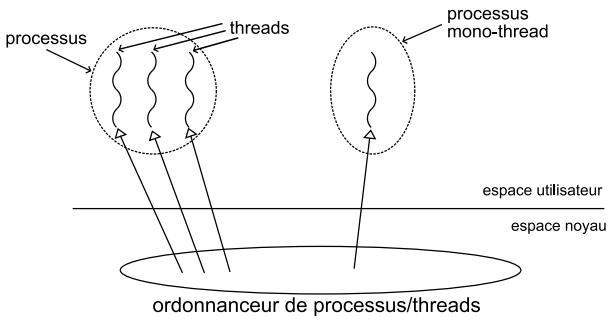


FIG. 1.5 – Un ordonnanceur de niveau noyau.

Cette approche permet la gestion d'architectures SMP, puisque l'ordonnanceur système voit de la même façon les threads et les processus. Il a donc connaissance de l'existence de ces processus légers et peut les faire exécuter par n'importe quel processeur de la ma-

chine. De plus, ce type de bibliothèque offre une bonne interaction avec les entrées/sorties, puisqu'un thread exécutant un appel bloquant ne gène en rien l'exécution des threads du processus dont il fait partie. En revanche, la gestion de ces threads est très chère, la plupart des primitives d'une telle bibliothèque nécessitant un appel système, et on peut apparenter ces threads aux véritables processus.

Les bibliothèques hybrides Une bibliothèque hybride déploie des threads utilisateur sur un ensemble de threads lourds gérés par l'ordonnanceur système de niveau noyau. En pratique, plusieurs threads noyau sont attribués à un processus créant des threads utilisateur. On parle d'ordonnanceur N : M dans le cas où N threads utilisateurs s'exécutent par l'intermédiaire de M threads noyau, comme l'illustre la figure 1.6.

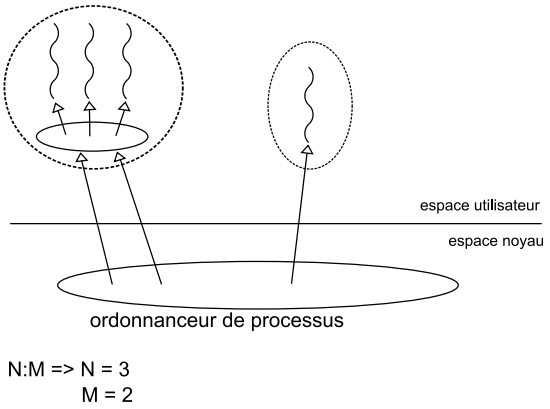


FIG. 1.6 – Un ordonnanceur hybride 3 : 2.

Bien qu'un petit peu moins rapide qu'un ordonnanceur utilisateur pur, cette approche offre de bonnes performances et supporte les architectures SMP. Dans l'exemple de la figure, deux threads utilisateur, parmi les trois contenus dans le processus de gauche, pourront s'exécuter en parallèle sur deux processeurs différents, puisque le processus dispose de deux threads noyaux, qu'on peut assimiler comme deux rampes de lancement vers un des processeurs. Cette approche bénéficie aussi de la flexibilité des bibliothèques utilisateur. Cependant, un tel ordonnanceur est plus complexe à mettre en œuvre, et n'obtient pas les résultats d'une bibliothèque de niveau noyau concernant les interactions avec les entrées/sorties.

Présentation de quelques bibliothèques de processus

La norme POSIX, créée en 1995, a standardisé l'interface des processus légers (interface PTHREAD). Ainsi, tous les systèmes d'exploitation UNIX en disposent et de nombreuses applications et bibliothèques l'utilisent. Comme toute interface normalisée, PTHREAD ne répond pas à certains besoins spécifiques[2]. Aussi des bibliothèques de processus légers aux fonctionnalités avancées continuent d'être développées. De nombreux codes peuvent alors être facilement portés vers ces bibliothèques et légèrement modifiés pour tirer parti de ces

[2]L'interface PTHREAD ne fournit notamment aucune fonctionnalité de placement des processus légers sur une architecture de machine NUMA, ni aucune primitive permettant de regrouper des processus partageant des ressources.

nouvelles fonctionnalités. Nous présentons dans cette partie deux bibliothèques de processus légers.

NPTL Native POSIX Thread Library est la bibliothèque de processus légers intégrée à Linux. Cette bibliothèque de niveau noyau a été écrite avec pour principaux objectifs une conformité parfaite à la norme POSIX. Les fonctionnalités offertes par NPTL sont donc strictement celles décrites dans la norme POSIX. L'exploitation des nouvelles fonctionnalités introduites dans les noyaux Linux récents permettent à NPTL d'être plus performant que ses prédécesseurs (LINUX-THREADS notamment). De plus, cette bibliothèque intègre un mécanisme de synchronisation efficace [6], minimisant les sollicitations du noyau.

MARCEL MARCEL est une bibliothèque de threads utilisateurs faisant partie de la suite logicielle PM2 [9], développée dans l'équipe RUNTIME du LaBRI. Elle est dotée d'une interface POSIX et est configurable pour fonctionner en espace utilisateur ou en fonctionnement hybride. À l'origine, cette bibliothèque a été pensée pour faciliter la migration de threads sur une machine, ou à travers un réseau, permettant ainsi la mise en place de solutions d'équilibrage de charge. Aujourd'hui, cette bibliothèque fournit bien plus que cette simple migration. Elle introduit notamment le concept de bulle [15], structure de données permettant à l'utilisateur de regrouper des tâches par affinité. Ces bulles peuvent exprimer des relations comme le partage de données, la participation à des opérations collectives, ou plus généralement un besoin de politique d'ordonnancement particulier. Ces politiques sont mises en œuvre à travers différents ordonnanceurs qui prennent en charge la répartition de ces bulles selon différents critères. Il existe notamment un ordonnanceur spécifiquement créé pour une répartition des bulles maximisant l'équilibrage de charge, ou encore un qui alimente les processeurs par vol de tâches. Notons aussi que cette bibliothèque met à disposition un ensemble de primitives permettant la création de son propre ordonnanceur, et un module de traces permettant l'analyse post-mortem de son comportement, sur des programmes d'exemples aussi bien que sur une exécution en conditions réelles.

1.2.2 L'approche langage de programmation parallèle

Les langages de programmation parallèle masquent certains aspects fastidieux de la gestion des processus légers, en intégrant par exemple des mécanismes de création et synchronisation implicites de tâches. Ils apportent un confort de développement supérieur à celui proposé par l'approche bibliothèque de processus, mais parfois au détriment des performances, puisque de nombreuses optimisations ne sont accessibles que lorsqu'on gère des processus légers «à la main». Nous présentons ici quelques langages couramment utilisés pour exprimer le parallélisme des applications.

Cilk

Le but de Cilk est de permettre l'écriture de programmes parallèles efficaces quel que soit le nombre de processeurs de la machine. Cilk-5 [7] est une extension du langage C permettant de lancer des tâches de manière extrêmement efficace : le surcoût du lancement d'une nouvelle tâche n'est que de 2 à 6 fois le coût d'appel d'une fonction C classique. Pour

atteindre une telle efficacité, seuls quelques processus légers (créés au démarrage du programme) s'exécutent et aucune synchronisation entre eux n'est utilisée au cours du programme : ce ne sont que des «conteneurs» du point de vue de Cilk. Les tâches ou fonctions à exécuter en parallèle sont rassemblées dans une file de travaux. Ceux-ci sont choisis puis exécutés dès qu'un processeur est disponible.

Cette approche permet une parallélisation de programme très efficace et pratiquement sans surcoût sur machine monoprocesseur, mais ne s'applique qu'à des algorithmes récursifs, du type «diviser pour régner».

UPC

Le langage UPC (*Unified Parallel C*) créé en 2001 repose sur le concept de mémoire virtuellement partagée (MVP) et s'adresse principalement aux grappes, en particulier aux grappes de multiprocesseurs. L'approche MVP apporte un grand confort d'utilisation puisqu'elle masque au programmeur les accès distants à la mémoire, contrairement à l'approche communication par messages explicites. Utilisée sans connaissance de la répartition des données par rapport aux flots de calculs, un effet « ping-pong » peut apparaître conduisant alors à des performances catastrophiques. L'originalité du langage UPC est de permettre l'exploitation du modèle MVP en distribuant dynamiquement le calcul par rapport au placement des données. Pour cela le programmeur explicite les variables à partager, indique la répartition et le type de cohérence mémoire[3] qu'il désire leur appliquer. La distribution du calcul se fait par rapport aux données grâce à la définition par le programmeur d'une information d'affinité au niveau de la boucle à distribuer. Ainsi, dans l'exemple illustré par la figure 1.7, pour tout indice `k` la somme `a[k] = b[k] + c[k]` sera réalisée par le thread a proximité de la variable partagée `a[k]`.

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

FIG. 1.7 – Cas d'école de l'utilisation du langage UPC.

Les extensions introduites dans le langage UPC devraient permettre d'exploiter efficacement les multiprocesseurs hiérarchiques, il serait donc intéressant d'en réaliser une version optimisée à cette fin.

OPENMP

Le standard industriel OPENMP [1] définit un ensemble d'annotations pour la parallélisation de programmes séquentiels. Ces dernières permettent de créer un nombre concerté de threads, en relation avec la machine sur laquelle le programme est exécuté. L'utilisateur

[3]On dispose de deux types de cohérence : *séquentielle* (toute écriture ou lecture d'une donnée demande la synchronisation de la mémoire) et *relâchée* (la mémoire n'est synchronisée qu'aux points de synchronisation du programme, comme lors des barrières).

dispose de mots-clés qu'il peut passer en paramètres d'une directive OPENMP, afin notamment :

- d'exprimer la visibilité (privée ou partagée) de variables ;
- de spécifier le type de répartition d'indices lorsqu'on parallélise une boucle ;
- de contrôler le nombre de threads créés pour exécuter une section parallèle.

Le standard définit aussi des primitives automatisant les mécanismes de synchronisation, d'exclusion mutuelle ou encore de terminaison de threads. Son comportement, illustré en figure 1.8, est basé sur le modèle *Fork/Join*[4] qui décrit la création de threads par le processus invocant une section parallèle (*fork*), et l'attente de ces threads par ce même processus avant de continuer son exécution (*join*).

```
int
main (int argc, char **argv) {
  int matrice[MAX][MAX];

  ...

#pragma omp parallel shared(matrice) num_threads(4)
  {
    // Le contenu de ce bloc sera exécuté en parallèle,
    // par 4 threads partageant la donnée "matrice"

    ...

  }

  return 0;
}
```

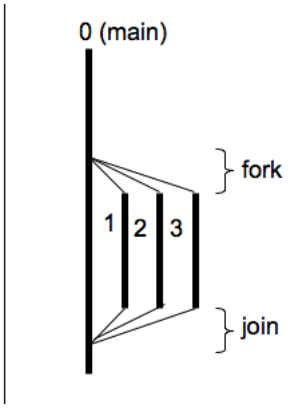


FIG. 1.8 – Exemple naïf de programme OPENMP.

Les programmes parallélisés avec OPENMP peuvent être écrits en C, C++ ou FORTRAN, et les annotations ajoutées seront ignorées par un compilateur non compatible. Notons que le consortium «*OpenMP Architecture Review Board*» réfléchit actuellement à la prise en compte par le standard des architectures NUMA, mais rien n'existe actuellement.

Nous présentons maintenant quelques implémentations de ce standard.

OMNI et OMNIST Le compilateur de la suite logicielle OMNI [12] a été initialement pensé pour fournir une implémentation d'OPENMP fonctionnant sur des ordinateurs à mémoire partagée, ou sur des grappes de machines, de façon transparente pour l'utilisateur. OMNIST [13] est un portage du compilateur OMNI sur la bibliothèque de processus légers STACKTHREADS/MP [14], développée dans l'optique de supporter de façon efficace les sections parallèles imbriquées. Pour ce faire, OMNIST met en œuvre un processus de création paresseuse de tâches, permettant de différer l'allocation réelle d'un processus léger au moment de son exécution effective. Cette approche réduit considérablement le coût de création d'un thread. De plus, OMNIST utilise un vol de travail actif, nécessitant l'appel à une fonction de la bibliothèque de façon régulière pour distribuer les threads entre les différents processeurs.

[4] graphes série parallèle.

GOMP Intégré à la récente version 4.2 de GCC, le compilateur GOMP [2] est capable de compiler des applications OPENMP développées en C, C++ ou FORTRAN95, sur de multiples architectures. Le support exécutif de GOMP autorise les sections parallèles imbriquées, et ne limite pas le nombre de threads créés comme OMNI. Il permet donc de surcharger les processeurs, c'est-à-dire définir plus de threads que de processeurs, mais n'offre aucun mécanisme adapté au bon déroulement de cette surcharge, n'assurant en rien la performance d'une application à parallélisme imbriqué.

1.3 Discussion

La plupart des implémentations d'OPENMP actuelles se contentent de fixer un thread par processeur et d'organiser des tâches au-dessus de ces threads. Contrairement à l'idée répandue que l'utilisation d'un grand nombre de threads écroule fatalement la machine, des études récentes [11] ont montré qu'on pouvait gagner à surcharger la machine, pour plusieurs raisons.

Tout d'abord, plus les threads sont nombreux, plus forte est la probabilité que la taille des données sur lesquelles ils travaillent soit petite. Ces données ont donc potentiellement plus de chances de tenir dans la mémoire cache ce qui permet d'accélérer le calcul.

De plus, la surcharge facilite l'équilibrage de charge (migration de threads vers les processeurs les moins chargés) et permet l'utilisation de sections parallèles imbriquées, rendant plus naturel le développement d'applications.

À l'heure actuelle, les quelques rares implémentations d'OPENMP qui autorisent la surcharge restent fortement dépendantes de l'ordonnanceur système sous-jacent et on sait qu'un mauvais placement sur une machine hiérarchique détériore notablement les performances. Malheureusement, les fonctionnalités offertes par les système d'exploitation quant au placement des threads et à leurs données sont assez rudimentaires. Le système LINUX permet de cantonner l'exécution d'un thread à un ensemble défini de processeurs. Les systèmes TRU64 de HP et SOLARIS de SUN y adjoignent la notion de groupe de threads. Cependant, il nous semble que, pour être efficace, le nombre et la composition des groupes de threads sont fonctions de la charge de la machine et doivent suivre celle-ci dynamiquement. Par ailleurs, des systèmes dédiés à des machines NUMA (TRU64 de HP, IRIX de SGI et un patch LINUX de BULL) s'appuient sur un support matériel spécifique pour décider par des heuristiques de déplacer des processus ou des données en mémoire. Cependant ce type de procédé reste coûteux en temps et est souvent désactivé.

De façon générale, on peut dire que l'exécution des programmes multithreadés sur machines hiérarchiques souffre d'un manque de dialogue avec l'ordonnanceur du système d'exploitation. C'est à cette fin que la plateforme de développement d'ordonnanceur BUBBLESCHED de la bibliothèque MARCEL est développée dans l'équipe RUNTIME. Informée de l'affinité liant les différents threads et de la topologie de la machine, cette bibliothèque peut judicieusement répartir threads et données sur les machines hiérarchiques.

Ainsi, une collaboration entre OPENMP et MARCEL doit permettre d'obtenir une solution portable et performante pour le développement d'applications parallèles sur des machines de calcul hiérarchiques. La mise en œuvre de cette collaboration est présentée dans le chapitre suivant.

Chapitre 2

Ordonnancement des bulles dans les programmes OPENMP

En utilisant les annotations d'OPENMP pour indiquer quelles sections d'un programme sont exécutables en parallèle, le programmeur donne indirectement des indications sur le parallélisme de son application au support d'exécution. Ces informations sont exploitées par OPENMP pour créer une hiérarchie d'équipes de threads formant un arbre. Il est possible de modifier un compilateur OPENMP de façon à ce qu'il génère une hiérarchie de bulles MARCEL suivant cet arbre. De cette façon, le support d'exécution fournit alors une hiérarchie d'entités ordonnançables par MARCEL.

On peut alors créer dans MARCEL un ordonnanceur qui répartit cette hiérarchie de bulles sur une machine de calcul, en respectant les contraintes d'affinité exprimées par OPENMP. On obtient ainsi une solution portable et performante de parallélisation de programmes par OPENMP sur des machines de calcul hiérarchiques, mais aussi une plateforme d'expérimentations pour des stratégies d'ordonnancement à venir.

Ce chapitre présente les outils qui nous ont permis d'y parvenir, et la façon dont nous avons abordé le problème.

2.1 La plateforme de développement d'ordonnanceurs de threads BUBBLESCHED

Pour accélérer l'exécution d'une application sur une machine hiérarchique, la bibliothèque MARCEL, outre son interface POSIX, dispose d'un ensemble fonctionnalités supplémentaires, dénommé BUBBLESCHED permettant de modéliser la machine, et de structurer l'ensemble des threads de l'application.

2.1.1 Modélisation des machines de calcul

Nous avons vu précédemment que les machines de calcul d'aujourd'hui deviennent de plus en plus complexes. Il devient plus que jamais nécessaire de trouver une façon de les modéliser afin de pouvoir développer des algorithmes sachant répartir efficacement des entités sur cette représentation.

De la subtilité de représenter les machines de calcul actuelles

Avec la complexification croissante des architectures de machines actuelles, les systèmes contemporains tentent tant bien que mal de proposer des modèles permettant à des applications parallèles d'occuper toutes les unités de calcul. Ils mettent ainsi en place des listes de threads, sur lesquelles les processeurs viennent «piocher» du travail lorsqu'ils sont inactifs.

La version 2.4 du noyau LINUX proposait par exemple une liste globale à tous les processeurs de la machine. L'augmentation du nombre de processeurs et de la contention sur cette liste allait donc de pair. La nécessité de prendre en compte des machines à plusieurs centaines d'unités de calcul imposa un changement de modélisation.

C'est pourquoi la version 2.6 du noyau LINUX propose une liste de threads par processeur. Un mécanisme avancé rééquilibre la charge entre les différentes listes. En revanche, rien n'assure que ce rééquilibrage n'entraîne pas l'éloignement d'entités travaillant sur les mêmes données, le mécanisme ne tenant pas compte des distances entre les différents blocs de mémoire d'une machine NUMA par exemple.

D'une manière plus générale, aucun système actuel ne dispose d'une modélisation dont puissent tirer parti les outils de placement de processus légers. Certains d'entre eux, en particulier la bibliothèque MARCEL, proposent leur propre modélisation.

La solution apportée dans MARCEL : une hiérarchie de listes de threads prêts

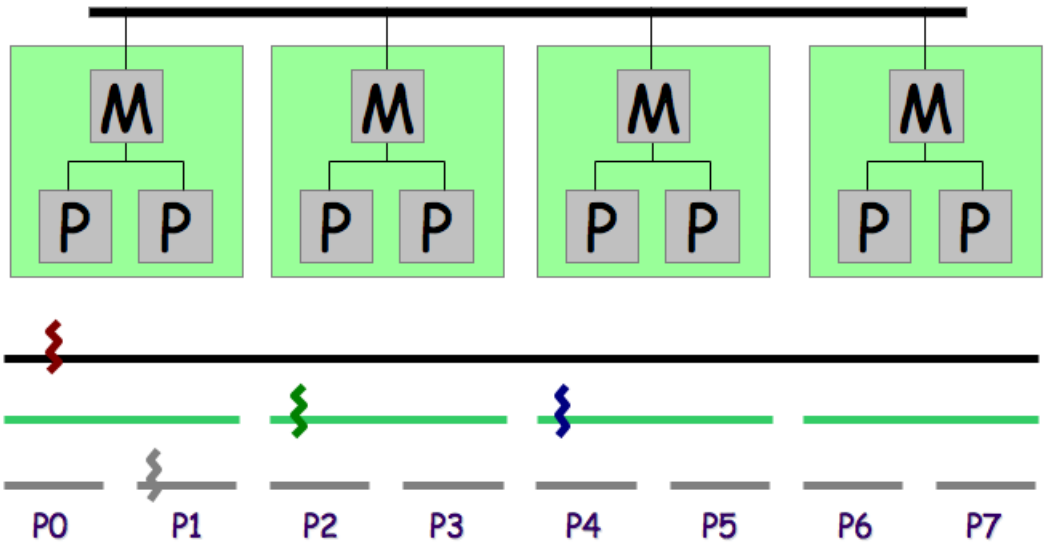


FIG. 2.1 – Modélisation d'une architecture de machine hiérarchique à l'aide d'un arbre de listes.

Une machine hiérarchique peut être déclinée en niveaux, correspondants à des points de vue plus ou moins proches du matériel. On considère par exemple la machine complète comme le niveau racine de la hiérarchie. Plus on regarde de près l'architecture de la machine, plus on voit apparaître des niveaux, correspondant à des entités de calcul toujours plus spécialisées.

Certaines machines sont ainsi constituées de plusieurs nœuds NUMA, qui caractérisent des distances entre les mémoires attachées aux différentes entités de calcul de la machine. Typiquement, un accès mémoire intra-nœud sera plus rapide qu'un accès inter-nœuds. Ces nœuds peuvent être composés de plusieurs puces, sur lesquelles on peut trouver plusieurs cœurs, qui eux-mêmes peuvent comporter plusieurs processeurs logiques.

De manière similaire à la hiérarchie de listes de Nano-Threads [10], cette hiérarchie d'entités est modélisée dans MARCEL à l'aide d'une hiérarchie de listes de threads. La machine en entier, chaque nœud NUMA, chaque puce, chaque cœur et chaque processeur logique sont ainsi représentés par une telle liste, comme l'illustre la figure 2.1.

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.

2.1.2 Encapsulation de threads dans des bulles

L'utilisateur donne des informations sur le parallélisme de son programme en regroupant des processus dans des structures de bulle, introduites en section 1.2.1. Ces structures sont récursives, c'est-à-dire qu'une bulle peut contenir d'autres bulles, comme le montre la figure 2.2.

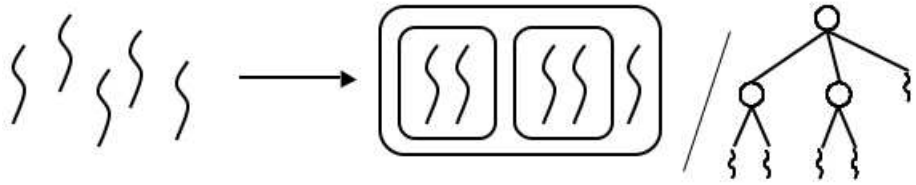


FIG. 2.2 – Encapsulation de processus légers dans des bulles.

Les bulles ont également un ensemble d'attributs qui peuvent être spécifiés par le programmeur pour donner des informations sur l'ensemble des threads contenus, comme par exemple :

- La *priorité* : elle permet d'évaluer le caractère prioritaire des threads contenus dans la bulle.
- L'*élasticité* : une bulle est élastique si les threads qu'elle contient peuvent être exécutés n'importe où sur la machine sans diminuer fortement les performances du programme.
- Des estimation de l'utilisation processeur ou mémoire.

Des compteurs peuvent également permettre de connaître le nombre total de threads, de threads actifs, ou la quantité de mémoire allouée.

2.1.3 Répartition de ces bulles sur la machine

Par défaut les bulles créées sont déposées sur la liste la plus haute de la topologie. Cela signifie que tous les processeurs de la machine peuvent ordonnancer les threads contenus

dans ces bulles. Cette répartition maximise l'occupation des processeurs, mais ne prend cependant pas compte des affinités entre threads et processeurs, puisqu'aucune relation entre eux n'est utilisée.

À l'inverse, les threads peuvent être distribués en prenant fortement les relations entre eux en compte, puisqu'ils peuvent être correctement répartis sur les processeurs selon leurs affinités. Cependant, si certains threads s'endorment, les processeurs correspondant deviennent inactifs, et l'utilisation des processeurs est donc potentiellement partielle.

On peut donc définir un ordonnanceur, dans la bibliothèque MARCEL, comme étant une interface de programmation permettant de répartir facilement bulles et threads sur les listes, afin d'obtenir des distributions telles que présentées en 2.1.1. Un ordonnanceur, c'est donc un ensemble de fonctions qui prend des décisions, en rapport avec le placement des entités sur les listes de threads, lorsqu'on demande de répartir de nouvelles entités, lorsque des entités meurent, ou encore lorsqu'un processeur n'a rien à faire.

2.2 Intégration des bulles dans les programmes OPENMP

OPENMP permet à l'utilisateur d'indiquer, par des annotations interprétables par le compilateur, quelles parties de son programme sont exécutables en parallèle. Un compilateur compatible OPENMP est donc capable d'appeler des fonctions spécifiques pour la création et l'exécution de processus légers se répartissant un travail désigné par l'utilisateur. On appelle «équipe» un groupe de processus légers créés lors d'une section parallèle. Une équipe regroupe donc plusieurs processus légers qui, pour parvenir à l'accomplissement d'un objectif commun, partagent des ressources et se synchronisent régulièrement.

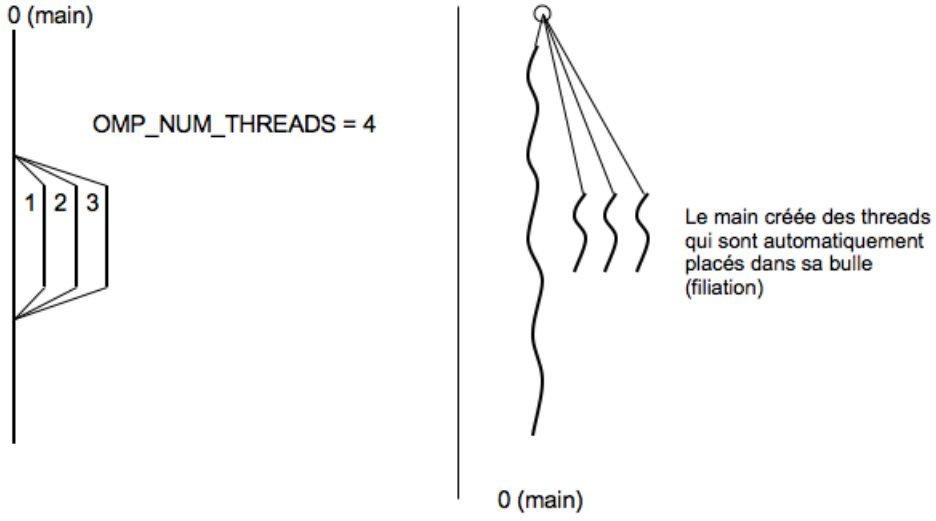


FIG. 2.3 – Comportement désiré du support exécutif d'OPENMP.

Faire générer des bulles à un support exécutif d'OPENMP revient donc à regrouper les membres d'une équipe, lors de sa création, dans une bulle, comme illustré par la figure 2.3.

2.3 Un ordonnanceur pour la répartition de ces bulles

Les applications OPENMP génèrent maintenant automatiquement des bulles, à partir des informations fournies par l'utilisateur sur les relations entre les tâches de son application. Pour qu'elles soient efficaces sur des architectures de machines hiérarchiques, il est nécessaire de créer un ordonnanceur spécifique dans MARCEL, baptisé *Affinity*, dont le rôle sera de répartir ces bulles sur la machine, en respectant les relations d'affinité exprimées entre les tâches.

2.3.1 Réflexions sur le comportement souhaité de l'ordonnanceur *Affinity*

Dans les applications de calcul parallélisées avec OPENMP, beaucoup de groupes de threads travaillent sur des données partagées, comme des indices de matrice par exemple. Afin d'évaluer les conséquences d'un mauvais placement, sur une machine hiérarchique, d'entités partageant des données, nous avons écrit un programme synthétique, dans lequel deux équipes de deux threads sont créées explicitement. Chaque équipe a la charge de lire, un grand nombre de fois, toutes les entrées d'un tableau qui occupe la totalité de la mémoire cache d'un processeur.

Nous utilisons une fonctionnalité de MARCEL permettant de forcer l'exécution d'un thread sur un processeur désigné, afin de comparer les performances de deux placements distincts illustrés par la figure 2.4.

Nous observons que l'exécution du programme pour lequel les entités sont placées selon le schéma 1 est presque deux fois plus longue que si elles sont placées selon le schéma 2. En effet, la répartition 2 maximise l'utilisation du cache, alors que les threads positionnés comme présenté par le schéma 1 se l'invalident continuellement.

L'ordonnanceur *Affinity* ne doit donc écarter les threads travaillant sur les mêmes données qu'en cas de force majeure, c'est-à-dire retarder au maximum l'éclatement des bulles, comme l'illustre la figure 2.5, pour que les entités d'un même groupe s'éloignent le moins possible.

2.3.2 Au cœur de l'algorithme

Cette section présente le comportement de l'algorithme récursif *Affinity*, permettant de répartir un ensemble de bulles sur une hiérarchie de listes de threads, en perçant le moins possible.

Appel à *Affinity* lorsque la machine n'est pas chargée

Après avoir placé de nouvelles bulles au sommet de la hiérarchie de listes, *Affinity* est appelé avec ce sommet en paramètre puis s'applique récursivement sur les niveaux inférieurs. La récursion porte ainsi sur les listes de threads de la topologie. Un appel récursif est invoqué sur chacune des listes sous-jacentes à la liste considérée, ce qui permet de parcourir toute la hiérarchie de listes à partir d'un appel à *Affinity*.

À chaque appel, on récupère l'ensemble des entités positionnées sur la liste considérée. On doit ensuite répartir ces entités sur les listes sous-jacentes, en évitant de percer des bulles s'il y en a. On commence donc par les compter, pour savoir si leur nombre est suffisant pour alimenter les processeurs sous-jacents dans la topologie. Si c'est le cas, une répartition

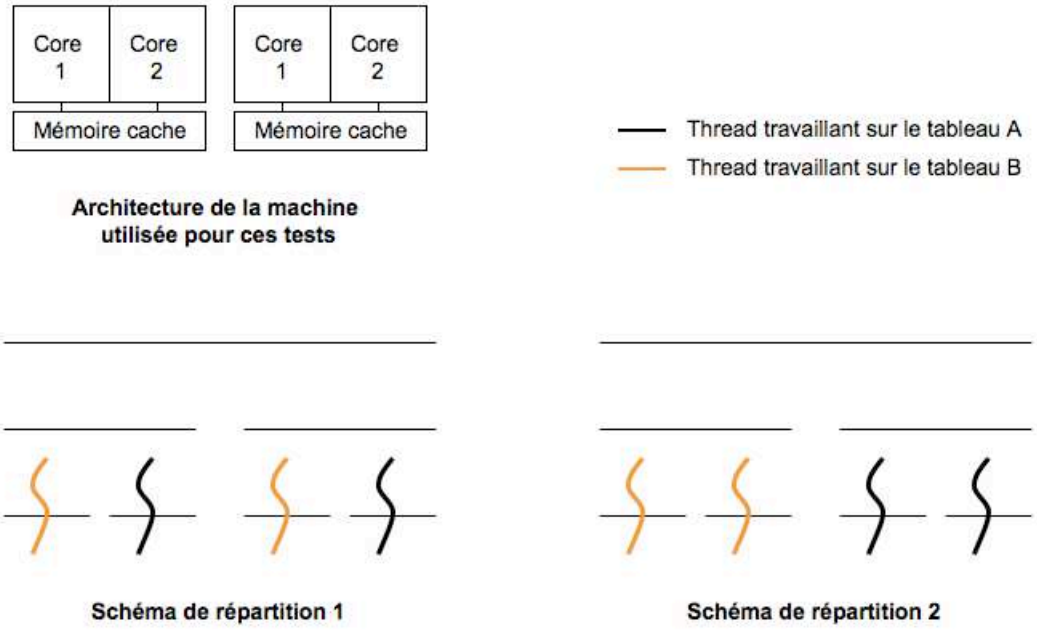


FIG. 2.4 – Deux placements différents pour un même programme.

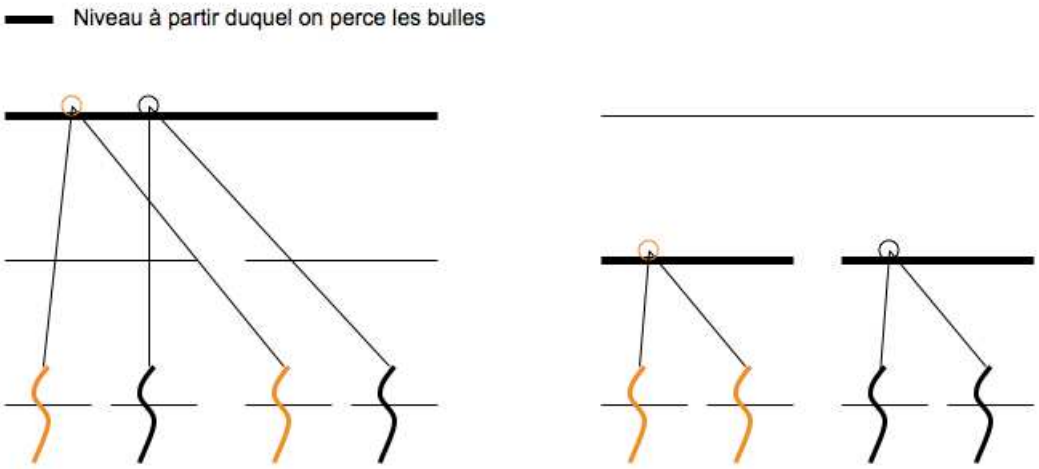


FIG. 2.5 – Retarder l'éclatement des bulles maximise la localité.

gloutonne suffit à donner du travail à tous les processeurs. On a donc assez d'entités sur la liste considérée, et on peut les distribuer sans les éclater.

Si leur nombre est insuffisant pour alimenter tous les processeurs sous-jacents, cela signifie qu'on sera forcé, à un certain moment de la récursion, de percer des bulles pour augmenter le nombre d'entités à répartir, et occuper ainsi tous les processeurs. On peut cependant essayer de retarder au maximum le moment de l'explosion, si le nombre d'entités à répartir est supérieur à l'arité de la liste qu'on considère. On examine le contenu des bulles pour déterminer si certaines d'entre elles contiennent assez d'entités pour alimenter un sous-arbre de la topologie. Si c'est le cas, on peut descendre ces entités intactes, retardant ainsi l'explosion de certaines d'entre elles.

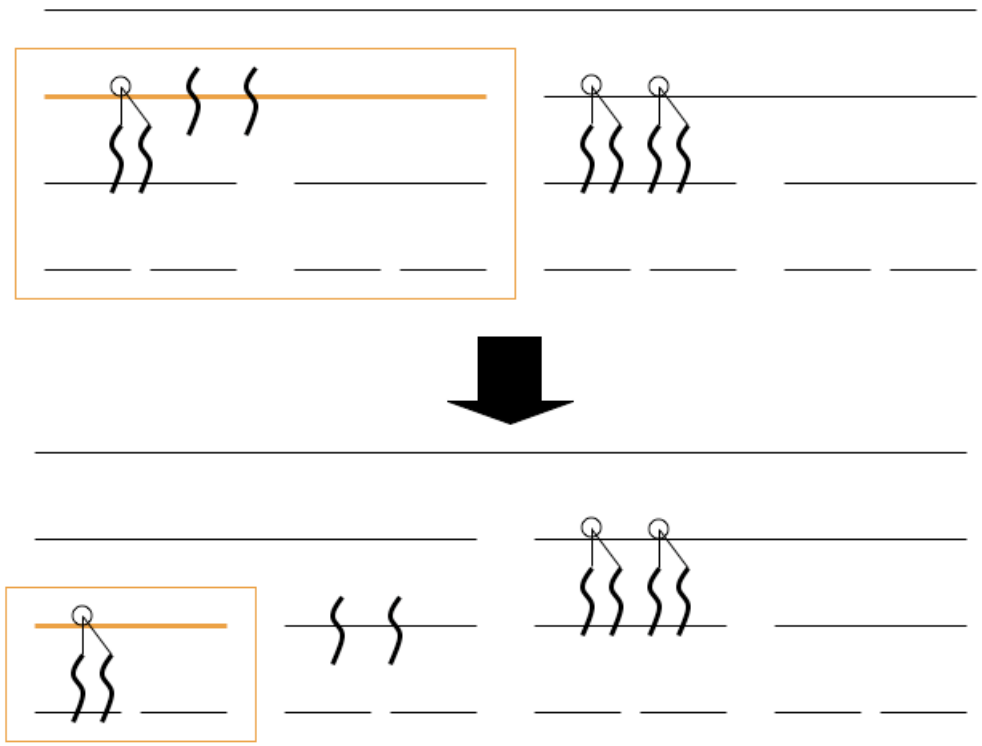


FIG. 2.6 – *Affinity* retarde l'éclatement des bulles.

La figure 2.6 illustre la situation où le nombre de d'entités est inférieur au nombre de processeurs à alimenter. La liste de threads représentée en gras correspond à celle considérée par *Affinity* à cet instant. Le rectangle de couleur met en évidence la sous-hiérarchie à laquelle s'intéresse présentement l'algorithme. La figure montre qu'*Affinity* trouve trois entités sur la liste, pour alimenter les quatre processeurs de cette sous-hiérarchie. Parmi ces entités se trouve une bulle, contenant deux threads. L'algorithme décide donc de retarder l'explosion de cette dernière, puisqu'elle contient assez d'entités pour alimenter les quatre processeurs sous-jacents, avec l'aide des deux autres threads présents sur la liste considérée.

En revanche, si le nombre d'entités à répartir est inférieur à la fois au nombre de processeurs sous-jacents à alimenter et à l'arité de la liste qu'on considère, ou si, après examen du contenu des bulles, on constate qu'elles ne suffiront pas à alimenter tous les processeurs, on se trouve dans l'obligation de percer une ou plusieurs bulles pour occuper toute la machine. Dans ce cas, on choisit de percer la bulle la plus grosse. On rappelle alors l'algorithme sur la même liste de threads, puisqu'elle contient maintenant, en plus des threads et bulles qui étaient déjà présents, les entités qui étaient contenues dans la bulle victime.

```
fonction affinity(liste_de_threads l) {
  entier nb_processeurs, arité;
  ensemble d'entités E;
  ensemble de listes Filles;

  nb_processeurs = récupérer_processeurs_sousjacents(l);
  Filles = récupérer_listes_filles(l);
  arité = |Filles|;
  E = récupérer_entités(l);

  si arité == 0
    retourner;
  finsi
  si |E| == 0
    retourner affinity(Filles);
  finsi

  si |E| < nb_processeurs
    si |E| >= arité et charge_est_suffisante(E)
      distribuer_gloutonnement(E, Filles);
      retourner affinity(Filles);
    sinon
      si percer_une_bulle(E) == succès
        retourner affinity(l);
      sinon
        retourner affinity(Filles);
      finsi
    finsi
  sinon
    distribuer_gloutonnement(E, Filles);
    retourner affinity(Filles);
  finsi
}
```

FIG. 2.7 – L'algorithme *Affinity* décrit en pseudo-code.

L'algorithme *Affinity* peut être exprimé à l'aide du pseudo-code présenté figure 2.7. Ce comportement permet de répartir efficacement un ensemble de bulles sur une machine non chargée. L'algorithme doit aussi être capable d'adapter son fonctionnement s'il est appelé sur une topologie sur laquelle des processus sont déjà en cours d'exécution. Nous développons ce cas dans la section suivante.

Appel à *Affinity* lorsque des tâches s'exécutent déjà sur la machine

Lorsque la machine est déjà chargée, même partiellement, le nombre de processeurs à alimenter est moindre. Le nombre d'entités nécessaires pour les alimenter est donc moins important, et la part des cas où l'on doit percer des bulles se réduit. Afin de pouvoir prendre des décisions quant à l'éclatement d'une bulle, l'algorithme doit connaître le niveau de charge de la machine, et quels sont les niveaux les moins chargés. Pour cela, on commence par évaluer de façon récursive le nombre de threads que comportent chaque niveau. Chaque liste renseigne donc la charge globale de l'arborescence sous-jacente. Dès qu'on place une entité sur une liste, sa charge est ajoutée à celle précédemment calculée. De cette façon, les nouvelles entités à répartir sont naturellement attirées vers les listes les moins chargées de la topologie.

En plus de la simple gestion de la répartition d'entités sur la machine, un ordonnanceur doit aussi être capable de prendre des décisions adéquates lorsqu'un processeur devient inactif. La section suivante traite du vol de travail mis en œuvre dans *Affinity*.

Comportement d'*Affinity* lorsqu'un processeur devient inactif

Afin de respecter la localité d'entités potentiellement en relation, un processeur inactif cherche à voler du travail de proche en proche, en commençant par les listes voisines, puis en remontant si la recherche n'a pas porté ses fruits. La découverte d'entités à voler provoque leur remontée vers le niveau le plus bas à partir duquel on doit rappeler *Affinity*, c'est-à-dire les répartir à nouveau. Ce niveau racine de la nouvelle répartition est déterminé au fur et à mesure du parcours de l'arbre des listes de tâches, lors de la recherche d'entités à voler. Voler du travail avec cet ordonnanceur revient donc à déterminer le niveau jusqu'auquel on doit remonter les entités sous-jacentes pour les répartir à nouveau en appelant la fonction de répartition d'*Affinity*.

2.4 Discussion

Pour le moment, lorsqu'on manque d'entités pendant la répartition, l'algorithme *Affinity* choisit de percer la bulle contenant le plus de sous-entités. Ce choix minimise le nombre de bulles éclatées, puisqu'en perçant la bulle contenant le plus de threads, on a plus de chances d'obtenir assez d'entités à répartir après cet éclatement. Cependant, on manque actuellement de critères de décisions permettant de définir une «épaisseur» de bulle, indiquant quelles bulles percer en premier. Un fort coefficient d'épaisseur pourrait par exemple indiquer que les threads contenus dans la bulle considérée travaillent de manière intensive sur une zone de données partagée, ou encore que ces threads se synchronisent très souvent. À l'inverse, les bulles de faibles coefficient d'épaisseur correspondraient à des bulles contenant des threads collaborant moins, et constitueraient ainsi des victimes privilégiées lorsqu'on doit éclater une bulle. Des travaux sont en cours dans l'équipe RUNTIME pour attacher ces critères de décision aux bulles sous la forme d'un coefficient, mais il est à noter que tout est d'ores et déjà prêt, dans l'algorithme *Affinity*, pour en tenir compte.

De plus, lors d'une nouvelle répartition orchestrée par *Affinity*, l'algorithme déplace des entités d'un processeur à un autre, mais ne déplace pas les zones mémoire sur lesquelles elles travaillaient. Cet aspect fait aussi l'objet de travaux, actuellement menés dans le projet RUNTIME en collaboration avec le laboratoire ID de Grenoble. À terme, lors d'une redistribution, et grâce au développement d'une bibliothèque spécifique de gestion de la mémoire

sur des machines NUMA, l'algorithme *Affinity* pourra prendre la décision de replacer les bulles et threads sur les processeurs sur lesquels ils ont déjà travaillé, ou décider de déplacer des zones mémoire accédées par ces entités vers leur nouvelle destination.

Enfin, il est possible, dans MARCEL, d'être prévenu à chaque fois qu'un processeur n'a rien à faire. On se sert de cet évènement pour appeler la fonction de vol de travail de l'ordonnanceur *Affinity*. Mais il existe des phases pendant lesquelles cet évènement doit être ignoré, pour que le vol de travail n'interfère pas sur d'autres mécanismes de MARCEL, par exemple la répartition des processus légers. En effet, un appel à la fonction de vol de travail pendant une répartition peut altérer le placement des entités sur la machine, handicapant fortement l'exécution du programme. De la même façon, en fin d'exécution, le programme ne comporte plus assez d'entités pour alimenter tous les processeurs de la machine. Les processeurs inactifs passent donc leur temps à tenter de voler du travail, sans succès. Mais la simple heuristique utilisée dans *Affinity*, indiquant de ne rien faire lorsque le nombre de processus légers devient inférieur au nombre de processeurs, a ses limites. Dans le cas illustré par la figure 2.8, on gagnerait à répartir les entités sur une moitié de la machine.

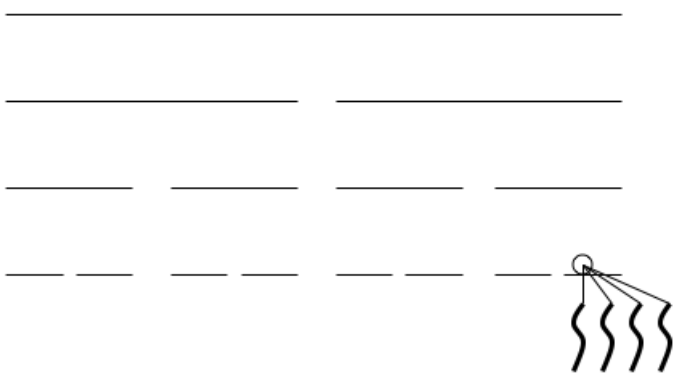


FIG. 2.8 – Exemple de positionnement des processus légers en fin de programme.

Chapitre 3

Mise en œuvre et évaluation

Ce chapitre décrit quelques points techniques concernant l'intégration de MARCEL dans le compilateur GOMP et la conception de l'ordonnanceur *Affinity*, puis présente les performances de cette solution sur deux programmes de test.

3.1 Extension d'un compilateur OPENMP pour créer des bulles

Cette section développe certains détails de l'intégration des bulles de MARCEL dans le compilateur GOMP.

3.1.1 Le choix du compilateur GOMP

Parmi les compilateurs OPENMP disponibles, GOMP est le plus portable. Utilisable sur des programmes C, C++ et FORTRAN95, il intègre des optimisations héritées de GCC pour bien des architectures. De plus, son intégration dans GCC lui assure d'être disponible, de base, dans de nombreux systèmes UNIX. Enfin, son architecture rend aisée l'intégration des bulles MARCEL.

Le code ajouté à ce compilateur pour générer automatiquement des bulles est très concis et pourrait facilement s'appliquer à plusieurs autres implémentations d'OPENMP.

3.1.2 L'architecture de GOMP

Un programme compilé avec GOMP est lié à l'aide d'une bibliothèque appelée `libgomp`. Cette dernière contient le code des fonctions appelées lorsque les annotations d'OPENMP sont interprétées par le compilateur. La figure 3.1 représente la pile logicielle associée au support exécutif de GOMP.

Intégrer MARCEL dans OPENMP revient à modifier le code de ces fonctions de façon à ce qu'elles appellent des méthodes de MARCEL, et ainsi générer une nouvelle bibliothèque `libgomp` avec laquelle lier les programmes OPENMP.

3.1.3 Comment faire générer des bulles à GOMP ?

La bibliothèque `libgomp` utilise des fichiers spécifiques pour chaque bibliothèque de thread qu'elle utilise, regroupés dans des dossiers indépendants. Une première étape consiste

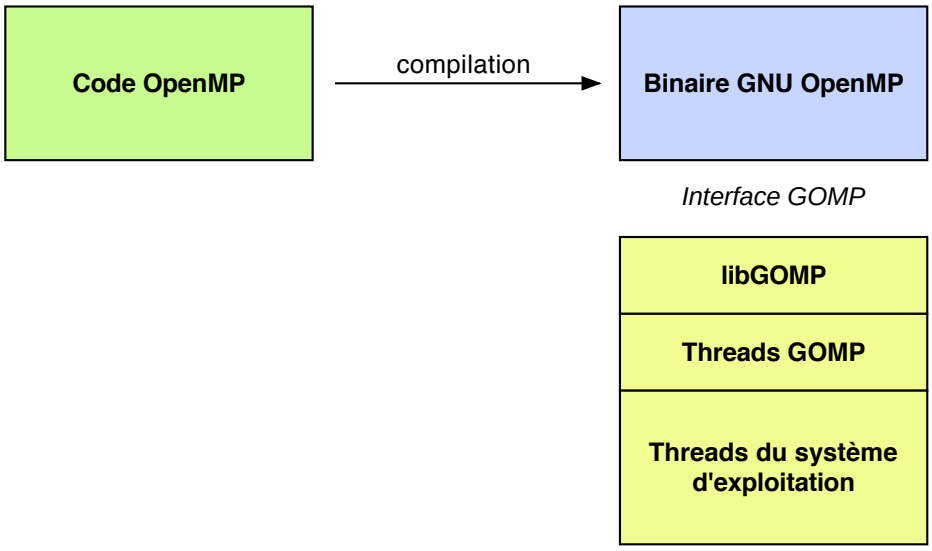


FIG. 3.1 – Pile logicielle associée au support exécutif de GOMP.

donc en la création de fichiers spécifiques à MARCEL, où les appels à la bibliothèque PTHREAD sont remplacés par leurs équivalents MARCEL. Une fois GOMP générant des threads MARCEL, il faut ensuite modifier une partie du code «commun» de `libgomp`, afin d'intégrer la création de bulles.

Lorsque, dans un programme OPENMP, l'utilisateur invoque une section parallèle, GOMP crée une équipe de threads, que l'on aimerait regrouper dans une bulle de la bibliothèque MARCEL. À la génération d'une équipe, on créé donc une structure de bulle, dans laquelle on place le thread appelant. On marque alors ce thread de façon à ce que la bulle créée apparaisse comme «structure mère» de ce processus léger. Cet attribut sera hérité lors de la création de nouveaux threads, ce qui signifie que chaque nouveau processus léger créé par le thread appelant sera placé dans la bulle nouvellement créée. Une terminaison d'équipe au sens OPENMP se traduit maintenant comme l'attente par le thread appelant de la fin d'exécution de tous ses fils, puis la destruction de la structure de bulle qui l'englobe et enfin le retour dans la bulle supérieure.

Ces modifications, très locales, du compilateur GOMP lui permettent de générer automatiquement des bulles en fonction des annotations indiquées par l'utilisateur. La section suivante présente l'implémentation de l'ordonnanceur qui permet de répartir ces bulles sur une topologie de machine hiérarchique.

3.2 Implémentation d'un ordonnanceur à bulles conservant l'affinité

Cette section développe quelques points techniques relatifs à la conception de l'ordonnanceur *Affinity*. On y évoque notamment la création d'une interface générique pour les ordonnanceurs à bulles, la mise en œuvre de structures et de fonctions spécifiques à la répartition, et le parcours particulier de la hiérarchie de liste, utilisé lors d'un vol de travail.

3.2.1 Création d'une interface générique pour ordonnanceurs à bulles

À l'origine, un ordonnanceur à bulles correspondait à un fichier source, dans lequel était définie une fonction de répartition, que l'utilisateur devait appeler explicitement dans ses programmes. Le passage d'une stratégie d'ordonnancement à une autre nécessitait donc une modification du code utilisateur dans les meilleurs cas, voire du code de MARCEL lorsqu'on redéfinissait la fonction de vol de travail.

Une première étape a donc été de définir une interface générique pour la création d'ordonnanceurs à bulles. Nous avons redéfini un ordonnanceur comme étant une structure, dont les points d'entrée sont constitués de pointeurs de fonctions qui définissent son comportement. Écrire son propre ordonnanceur revient donc à créer une nouvelle structure, et redéfinir les méthodes qu'elle contient pour préciser son comportement. Un ordonnanceur ressemble désormais à une structure de la forme :

```
/* Initialization */
typedef int (*ma_bubble_sched_init)(void);
/* Termination */
typedef int (*ma_bubble_sched_exit)(void);
/* Submission of a set of entities to be scheduled */
typedef int (*ma_bubble_sched_submit)(marcel_entity_t *);
/* Called when a processor is idle */
typedef int (*ma_bubble_sched_vp_is_idle)(marcel_vpmask_t);

struct ma_bubble_sched_struct {
  ma_bubble_sched_init init;
  ma_bubble_sched_exit exit;
  ma_bubble_sched_submit submit;
  ma_bubble_sched_vp_is_idle vp_is_idle;
};
```

Les champs de cette structure mis à NULL représentent des cas de figures pour lesquels l'ordonnanceur décide de ne rien faire. Par exemple, l'ordonnanceur my_sched définit ci-dessous ne possède pas de variables particulières à initialiser, ce qui se traduit par des pointeurs init et exit à NULL, et ne désire pas être prévenu lorsqu'un processeur n'a rien à faire (vp_is_idle à NULL).

```
int my_sched_submit(marcel_entity_t *e) {
  ...
}

struct ma_bubble_sched_struct marcel_bubble_my_sched = {
  .init = NULL,
  .exit = NULL,
  .submit = my_sched_submit,
  .vp_is_idle = NULL,
};
```

Cette nouvelle approche transforme un ordonnanceur en véritable plug-in interchangeable. Il est maintenant possible, dans MARCEL, de changer de politique d'ordonnancement au cours de l'exécution d'un programme.

3.2.2 Détails d'implémentation de la répartition par *Affinity*

Cette section présente la structure de données utilisée pour connaître le niveau de charge de chaque niveau de la machine, et la fonction déterminant si les entités sont suffisamment chargées pour alimenter tous les processeurs.

Le Load Manager

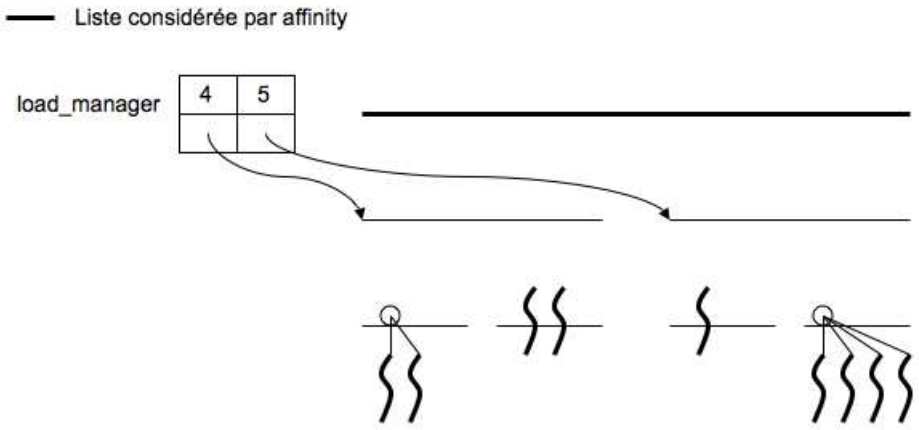


FIG. 3.2 – Le load_manager indique la charge des listes sous-jacentes.

Le `load_manager` est une structure introduite dans *Affinity* pour connaître la charge récursive des listes sur lesquelles on veut répartir des entités. Elle est représentée par un tableau à deux entrées, dont chaque indice désigne une liste sous-jacente.

```
struct load_indicator {
  int load;
  struct marcel_topo_level *l;
};

typedef struct load_indicator load_indicator_t;

...

load_indicator_t load_manager[arity];
```

La variable `arity` contient le nombre de listes «filles» de la liste considérée par *Affinity*. Une entrée de ce tableau, représentant une liste, contient donc le nombre récursif des entités se trouvant dans la sous-hiérarchie dont cette liste est racine, et un pointeur direct vers cette liste, comme le montre la figure 3.2.

Ainsi, pour accéder à la liste la moins chargée de la sous-hiérarchie, il suffit de trier le tableau `load_manager` en fonction des charges. Un tri croissant de ces charges nous assure que la liste pointée par le premier élément du tableau est la moins chargée.

La fonction Has enough entities

Nous avons vu précédemment que l'algorithme *Affinity* commençait par compter le nombre d'entités présentes sur la liste qu'il considérait. Nous avons aussi précisé que si ce nombre était inférieur au nombre de processeurs à alimenter, on pouvait retarder l'explosion d'une bulle en analysant le contenu des entités considérées. La fonction `has_enough_entities` permet cette analyse. Son fonctionnement est simple, puisqu'elle simule une répartition sans percer de bulles, sur la machine. À la fin de cette simulation, la fonction est capable d'exprimer s'il est nécessaire de percer une bulle pour alimenter les processeurs sous-jacents. Au lieu de déplacer les entités physiquement, on applique des modifications simulant cette répartition à une copie du `load_manager` présenté précédemment. Une fois la répartition achevée, on parcourt le `load_manager` pour déterminer si chacune des listes est suffisamment chargée pour alimenter les processeurs sous-jacents. Si c'est le cas, la fonction renvoie 1 et on évite l'éclatement d'une bulle.

3.2.3 Points techniques relatifs au vol de travail

On a coutume d'assimiler la hiérarchie de liste modélisant une topologie de machine à un arbre, dont les feuilles sont les processeurs. Alors qu'une répartition par *Affinity* provoque le parcours de la topologie de la racine vers les feuilles de cet arbre, le vol de travail, lui, demande une remontée progressive des feuilles vers la racine, afin de rechercher des tâches à voler. Chaque fois qu'une liste est parcourue, elle doit être verrouillée pour éviter qu'un autre processeur vienne la corrompre au même moment en y plaçant des entités par exemple. Ce verrouillage peut s'avérer coûteux, surtout pour des listes proches de la liste racine, potentiellement accédées par tous les processeurs en même temps. C'est pourquoi le parcours inversé initié par le vol de travail est effectué en deux temps. Un premier parcours, n'employant que des verrous légers, utilisés pour la lecture seulement, repère la ou les entités qui seront volées par le processeur inactif. Dans un deuxième temps, le voleur verrouille la liste source et destination de son vol, en lecture et écriture, et rapatrie les entités pour pouvoir les exécuter.

3.3 Évaluation

Nous évaluons les performances de notre implémentation à l'aide de deux programmes de test. Le premier, `omp-unbalanced`, est un programme synthétique créé pour l'occasion, qui a servi à tester les différents comportements de l'ordonnanceur au cours de son développement. Le second, BT-MZ (*Block Tri-diagonal, Multi-Zone*), est un programme de test faisant partie d'une suite de *benchmarks* développée par la NASA [5], spécifiquement créé pour évaluer les performances des compilateurs OPENMP.

Pour chacun de ces programmes, nous comparons les performances obtenues par l'implémentation originelle de GOMP fournie avec la version 4.2 du compilateur gcc, la version de GOMP modifiée pour créer des threads MARCEL, la version de GOMP modifiée pour générer automatiquement des bulles et les répartir avec *Affinity*.

3.3.1 Comportement détaillé d' *Affinity* sur une application synthétique

Le programme `omp-unbalanced` simule la réalité d'une application de calcul parallélisée avec OPENMP. Le thread principal invoque une section parallèle, dans laquelle il appelle une méthode s'apparentant aux fonctions de calcul des applications scientifiques. Cette méthode déclare un tableau d'un grand nombre d'entiers, et invoque à son tour une section parallèle afin de le parcourir. La tableau est suffisamment important pour occuper la mémoire cache d'un processeur. De cette façon, l'exécution de ce programme permet de mettre en évidence l'importance du placement des entités travaillant sur le même tableau, et de tester le comportement des différentes implémentations sur un programme OPENMP doté de parallélisme imbriqué.

De plus, nous avons fait en sorte de créer un nombre différent de threads à chaque appel à la méthode de calcul, afin de traduire le caractère irrégulier de nombreuses applications de calcul. Pour résumer, la première équipe, créée dans le `main`, est composée de quatre threads. Chacun des équipiers crée lui aussi une équipe, composée de deux ou quatre threads selon le numéro du créateur.

Ces tests ont été effectués sur une machine bi-Opteron Dual-Core, disposant de 1 Mo de mémoire cache par processeur.

TAB. 3.1 – Performances de `omp-unbalanced` sur les différentes implémentations.

Bibliothèque utilisée	Temps d'exécution
`libgomp` originale	32 s
`libgomp` + MARCEL	32 s
`libgomp` + MARCEL + *Spread*	28 s
`libgomp` + MARCEL + *Affinity*	18 s

Tout d'abord, les performances obtenues avec la version originale de GOMP et celle qui créé des threads marcel sont sensiblement les mêmes, et sont très inférieures à celles obtenues par la version de GOMP générant des bulles, réparties sur la machine par l'algorithme *Spread* dont le rôle est d'équilibrer au maximum la charge, elles-mêmes inférieures à celles obtenues avec *Affinity*. Étant de niveau utilisateur, la bibliothèque MARCEL bénéficie de primitives de gestion des processus légers plus performantes que NPTL. Cependant, les versions sans répartition de `libgomp` obtiennent les mêmes résultats sur `omp-unbalanced`, le nombre de threads créés par l'application n'étant pas suffisamment élevé pour constater cette différence. Les accélérations observées sur les versions appelant un algorithme de répartition ont donc été obtenues grâce à de l'ordonnancement pur. Pour comprendre ces résultats, analysons le placement de ces derniers sur la machine.

La figure 3.3 montre la répartition des entités lors de l'exécution du programme compilé avec la version originale de GOMP. Les douze threads créés par les différentes sections parallèles sont ordonnançables sur toute la machine, sans tenir compte des données sur lesquelles ils travaillent. Un processeur exécute un thread au hasard, qui charge le tableau sur lequel il travaille dans sa mémoire cache. Par souci d'équité entre les tâches, l'ordonnanceur du système préempte de façon régulière les processus qui s'exécutent, pour donner la main à d'autres processus prêts. Ainsi, lorsqu'une tâche reprend son exécution sur un processeur sur lequel elle s'est déjà exécutée, elle a de grandes chances de ne plus retrouver les données

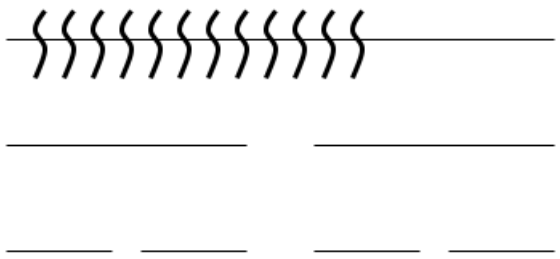


FIG. 3.3 – Répartition des entités d'`omp-unbalanced` par LINUX.

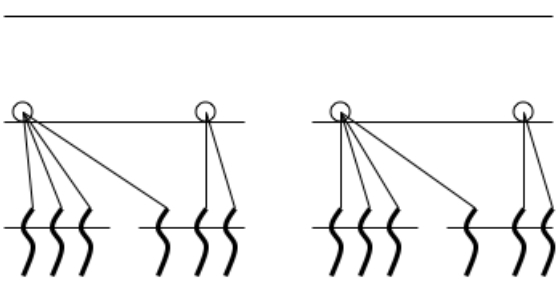


FIG. 3.4 – Répartition des entités d'`omp-unbalanced` par *Spread*.

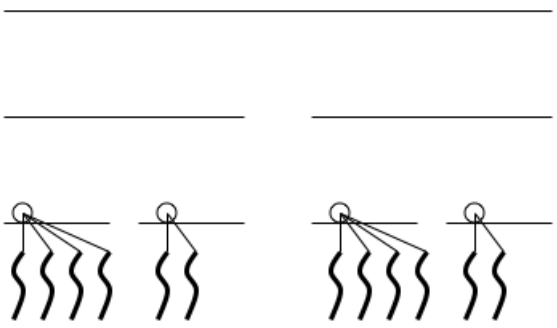


FIG. 3.5 – Répartition des entités d'`omp-unbalanced` par *Affinity*.

auxquelles elle accède dans la mémoire cache, puisque rien n'assure que les threads ordonnancés depuis sa préemption travaillaient sur le même tableau qu'elle. Les performances de l'application s'en trouvent déteriorées.

La figure 3.4 montre la répartition des entités, qui ont été regroupées dans des bulles, puis distribuées sur la machine par l'algorithme spread. Ce dernier cherche à équilibrer au maximum la charge sur la machine. Le programme créé 12 threads à répartir sur une architecture à 4 cœurs, *Spread* perce donc des bulles afin d'obtenir une répartition à 3 threads par cœur. Ainsi, bien qu'une telle répartition soit meilleure que celle proposée par NPTL, ou MARCEL sans bulles, certains membres d'une même bulles sont exécutés sur des cœurs différents. Ils utilisent donc des mémoires cache différentes, ce qui ralentit l'exécution du programme.

La figure 3.5 montre enfin la répartition des bulles, générées par GOMP, par l'algorithme *Affinity*. Ce dernier répartit les bulles sans les percer. Tous les cœurs n'ont pas la même charge de travail, mais l'utilisation des mémoires cache est optimale. C'est cette solution qui offre de loin les meilleurs performances.

L'algorithme *Affinity* se comporte très bien sur cette application synthétique, maximisant les effets de cache. Nous allons maintenant voir qu'il obtient aussi de bonnes performances sur une application de calcul moins prévisible.

3.3.2 Performances d'*Affinity* sur l'application BT-MZ

Les tests suivant ont été effectués sur une machine composée de huit processeurs AMD Opteron Dual-Core cadencés à 1,8 GHz, disposant de 64 Go de mémoire vive.

BT-MZ est un programme de la suite de *benchmarks* developpée par la NASA, couramment utilisée pour évaluer les performances des compilateurs OPENMP. Son but est de simuler, dans l'espace, des phénomènes physiques relatifs à la dynamique des fluides. Pour ce faire, la scène 3D est calculée à l'aide d'un maillage, découpé en plusieurs zones de travail. Le calcul à l'intérieur de chacune de ces zones est effectué en parallèle par plusieurs processus légers. On distingue donc deux niveaux de parallélisme dans cette application :

- On crée tout d'abord une équipe de threads pour découper le maillage en zones. On parle de parallélisme externe de l'application. Chaque thread créé devient alors responsable d'une zone de travail.
- Chaque responsable crée ensuite une équipe à son tour, à l'intérieur même d'une zone de travail, caractérisant le parallélisme interne de BT-MZ.

Le découpage de l'espace fait en sorte de créer des zones pour lesquelles la charge de travail est équivalente. BT-MZ constitue donc une application comportant du parallélisme imbriqué, dont les tâches ont approximativement la même durée.

L'accélération d'un programme est définie par la division du temps d'exécution de sa version séquentielle par celui de sa version parallèle.

La figure 3.6 compare les performances obtenues par la version originale de GOMP créant des threads de la bibliothèque NPTL, avec celle créant des threads de la bibliothèque MARCEL, sur plusieurs exécutions du programme BT-MZ, pour lesquelles on fait varier :

- le nombre de zones qui constituent le maillage, de 1 à 16, en abscisses sur la figure ;
- le nombre de threads créés dans chacune des zones, indiqués sur la légende par «*n».

Une exécution créant 16 zones, dans lesquelles on crée 8 threads, lancera donc au total 128 processus légers. La différence de performances entre MARCEL et NPTL s'explique en com-

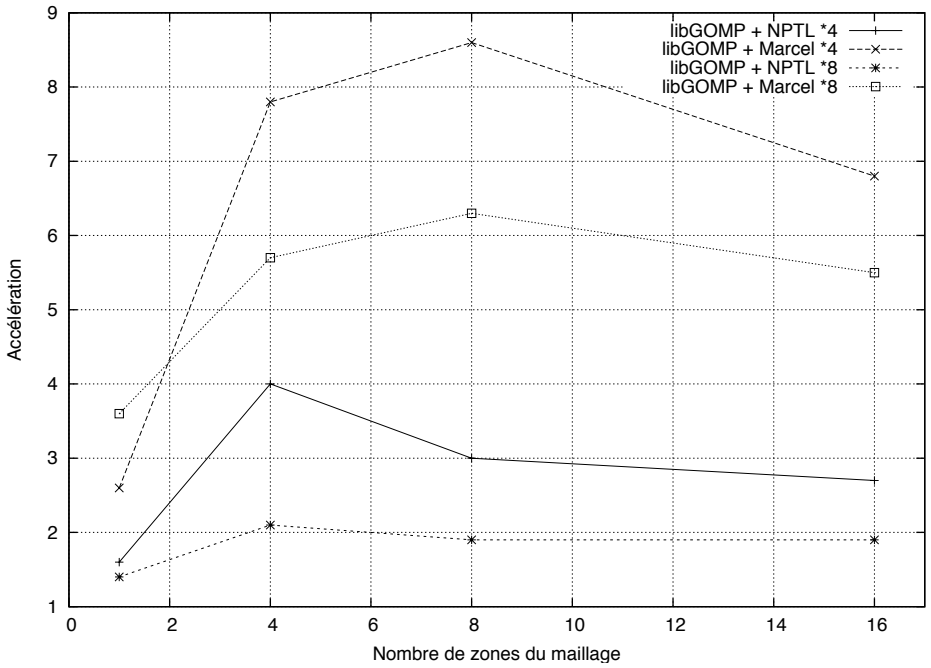


FIG. 3.6 – Comparaisons des performances des versions NPTL et MARCEL de `libgomp`, sur plusieurs exécutions de BT-MZ.

parant le coût de création et de gestion des threads pour chacune de ces deux bibliothèques. En effet, nous avons vu précédemment que la bibliothèque NPTL était de niveau noyau. Les threads créés par cette bibliothèque sont presque aussi lourds que de véritables processus, et leur gestion implique de nombreux appels systèmes coûteux. À l'inverse, MARCEL est une bibliothèque hybride, qui fixe un nombre de threads noyau à l'initialisation, et ne créé que des threads de niveau utilisateur ensuite. C'est pourquoi, à nombre de threads égal, la bibliothèque MARCEL obtient de meilleures performances qu'une bibliothèque de niveau noyau comme NPTL, alors qu'omp-unbalanced ne comportait pas assez de processus légers pour montrer cette différence.

Les figures 3.7 et 3.8 comparent les résultats obtenus par la version de GOMP modifiée pour générer des bulles réparties avec *Affinity* aux résultats précédemment obtenus. Les équipes de threads travaillent toutes sur des zones distinctes du maillage. En répartissant ces équipes sur la machine avec *Affinity*, on distribue implicitement ce maillage sur la topologie, et on assure que la majorité des accès mémoire invoqués par les membres d'une équipe, quelle qu'elle soit, sont locaux. De plus, augmenter le nombre de threads dans ce cas-là signifie simplement agrandir l'effectif d'une équipe, déjà placée sur un processeur, ce qui explique le bon passage à l'échelle de la solution appelant *Affinity*.

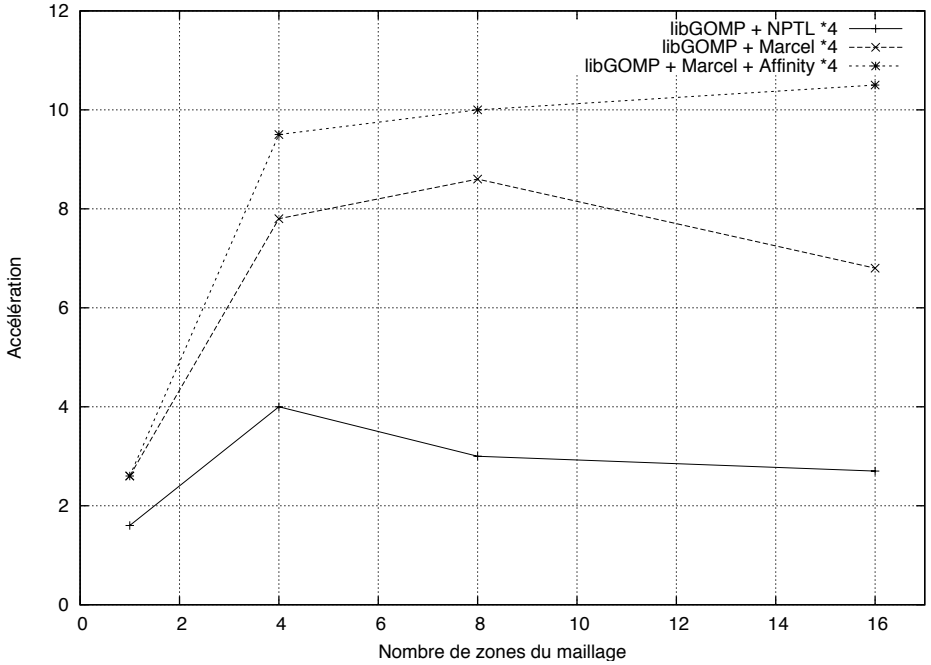


FIG. 3.7 – Comparaisons des performances des versions NPTL, MARCEL et MARCEL + *Affinity* de `libgomp`, sur plusieurs exécutions de BT-MZ créant 4 threads par zone.

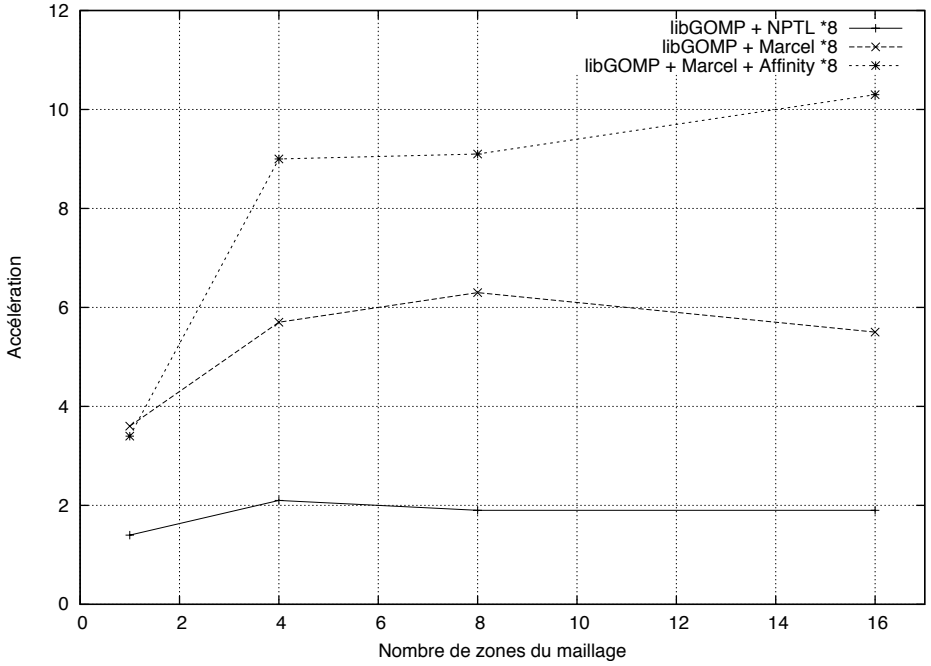


FIG. 3.8 – Comparaisons des performances des versions NPTL, MARCEL et MARCEL + *Affinity* de `libgomp`, sur plusieurs exécutions de BT-MZ créant 8 threads par zone.

Conclusion

Trente ans durant, les fondeurs de microprocesseurs se sont livrés une course à la fréquence. Confrontés à la barrière technologique de la dissipation thermique, ceux-ci se sont tournés depuis 2005 vers le parallélisme de flot en commercialisant des processeurs basés sur les technologies SMT et multicœurs. L'introduction de ces technologies renforce plus encore le caractère hiérarchique des calculateurs contemporains basés sur une interconnexion de briques constituées de processeurs et de bancs mémoires.

Les outils de programmation parallèle classiques, tels OpenMP, permettent d'exploiter de façon simple et portable ces architectures émergentes ; cependant leur efficacité est loin d'être optimale. En effet, ceux-ci, bien adaptés aux machines symétriques, ne prennent pas en compte la structure hiérarchique des calculateurs contemporains et, mis à part le langage CILK et ses émules[1], ne tirent pas parti de la structure hiérarchique du parallélisme de l'application.

Dans ce mémoire, nous avons jeté les bases d'un environnement de programmation OpenMP bien adapté aux machines hiérarchiques. Pour ce faire, nous avons modifié le code utilisé/généré par le compilateur GOMP pour qu'à l'exécution l'application exhibe dynamiquement la structure arborescente des flots de calcul par l'intermédiaire des bulles de la bibliothèque de threads MARCEL. Conjointement, afin de contrôler la répartition des bulles sur machines hiérarchisées, nous avons élaboré une stratégie d'ordonnancement de bulles spécifique aux programmes OpenMP et l'avons intégrée à la bibliothèque MARCEL.

Cette approche offre déjà des résultats très encourageants sur des applications à parallélisme imbriqué, comme le programme BT-MZ de la suite de benchmarks de la NASA. Naturellement, il est nécessaire de poursuivre les évaluations pour valider voire améliorer notre stratégie d'ordonnancement et son implémentation, puis il s'agira ensuite d'en étoffer les fonctionnalités. Voici quelques pistes :

À court terme, il s'agira de s'intéresser aux problèmes soulevés par l'apparition de distances entre mémoire et processeurs dans les architectures contemporaines. La technique de vol de travail peut provoquer un éloignement topologique entre un processus léger et les données auxquelles il accède. Cependant, comme des bibliothèques de gestion mémoire permettent de déplacer physiquement les pages mémoires, il est possible de migrer les pages accédées par tel processus léger sur le nœud NUMA l'exécutant. Ainsi, un problème intéressant est la conception d'un ordonnanceur capable de décider de l'ensemble des données à accompagner une bulle donnée lors d'une migration. Formellement, cet ensemble est fonction de la quantité de données à déplacer et du nombre de fois qu'on y accédera. Cette dernière information est difficile à obtenir, surtout lorsqu'on considère une application de calcul de nature irrégulière. L'algorithme devra donc être capable de prendre des décisions

[1] Le langage ATHAPASCAN-1 reprend certaines des idées de CILK

pertinentes relatives à la mémoire allouée par les entités qu'il déplace, en intégrant et poursuivant les travaux actuellement menés par Sylvain Jeuland dans l'équipe RUNTIME autour du thème « ordonnancement et placement mémoire sur machine NUMA » dans notre stratégie.

De plus, des études récentes ont montré que le modèle de création paresseuse de threads s'appliquait bien aux applications OPENMP [13]. Nous pourrons, dans la bibliothèque MARCEL, retarder l'allocation effective de la pile d'un processus léger au moment où il est exécuté pour la première fois sur la machine, et même réutiliser des piles déjà allouées par des threads ayant terminé leur exécution. Déplacer de telles entités sur une machine reviendra à copier un descripteur de processus léger d'un point à un autre de la topologie. En particulier, ces threads non-alloués formeront une cible privilégiée lors d'un vol de travail, accélérant sensiblement les routines de répartition des ordonnanceurs MARCEL.

Certaines applications OPENMP très irrégulières peuvent provoquer des déséquilibres sur la machine, de façon répétée, et ainsi nécessiter plusieurs appels aux fonctions de vol de travail des ordonnanceurs MARCEL. Ces répartitions successives peuvent s'avérer coûteuses, pour parfois, au final, ne déplacer qu'une bulle d'une liste à une autre. Ce genre de résultat n'est pas prédictible au moment où l'on demande une répartition. Aussi, pour résoudre ce problème, nous pourrons envisager de maintenir une copie «légère» de la topologie de la machine, sur laquelle on appliquera les algorithmes de répartition de manière relâchée (sans verrou), sans interférer sur le déroulement de l'exécution. Le résultat de cette simulation, comparé à l'occupation actuelle de la machine, permettra d'identifier les entités s'étant déplacées. Nous n'aurons alors qu'à appliquer un nombre limité de migrations, plutôt que répartir à nouveau l'ensemble des processus légers de la machine.

À plus long terme, une analyse statique du code des programmes utilisateurs permettrait de déterminer certains facteurs de la parallélisation, comme le taux de partage de certaines données, ou encore la répartition d'indices dans les boucles. Ce genre d'analyse permet déjà aux compilateurs performants tels que GCC d'opérer des optimisations structurelles sur les codes séquentiels. Une telle approche permettrait à un compilateur OPENMP modifié d'affecter un coefficient d'*élasticité* (évoqué en 2.1.2) aux bulles MARCEL générées, à partir des informations issues d'une telle analyse. L'algorithme *Affinity* privilégierait alors l'éclatement des bulles les plus élastiques. Il existe de plus un certain nombre de compteurs matériels, accessibles via des bibliothèques spécialisées, permettant par exemple de connaître le nombre de défauts de cache au cours de l'exécution d'un programme. L'analyse de ces informations pourrait engendrer la création d'évènements, interprétables par MARCEL. Nous pourrions imaginer demander une nouvelle répartition à l'ordonnanceur *Affinity* si le nombre de défauts de cache dépasse un certain seuil sur un processeur donné.

Ces modifications pourraient, dans un premier temps, être mises en oeuvre dans un compilateur tel que GOMP, pour être diffusées à la communauté.

Nous aimerions enfin influencer les futures versions du standard en collaborant avec l'*Architecture Review Board*, afin qu'elles permettent à l'utilisateur de préciser le comportement de son application, qu'il maîtrise parfaitement. Par exemple, il pourrait spécifier quel type d'ordonnancement s'applique à quelle section parallèle de son programme. Nous ferions ainsi intervenir différents ordonnanceurs MARCEL pour répartir efficacement un sous-ensemble d'entités. L'utilisateur est aussi le seul capable de spécifier des changements de phase dans son programme. Lui donner la possibilité de les exprimer par un mot clé permettrait à l'ordonnanceur MARCEL sous-jacent d'adapter son comportement, selon par exemple que le programme est en phase d'initialisation, de calcul ou de terminaison.

Bibliographie

[1] http ://www.openmp.org/.

[2] GOMP – An OpenMP implementation for GCC. http ://gcc.gnu.org/projects/gomp.

[3] Vincent Danjean. *Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs*. PhD thesis, École normale supérieure de Lyon, December 2004.

[4] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization*. Birkhauser Boston, 2006.

[5] Rob F. Van der Wijngaart and Haoqiang Jin. NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, 2003.

[6] Ulrich Drepper. Futexes are tricky. April 2004.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann editions, 1990. 4ème édition.

[9] Raymond Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. PhD thesis, Univ. de Lille 1, January 1997.

[10] Dimitrios S. Nikolopoulos, Eleftherios D. Polychonopoulos, and Theodore S. Papatheodorou. Efficient runtime thread management for the nano-threads programming model. In *IPPS/SPDP Workshops*, pages 183–194, 1998.

[11] Marc Pérache. *Contribution à l'élaboration d'environnements de programmation dédiés au calcul scientifique hautes performances*. PhD thesis, CEA/DAM Île de France, Université de Bordeaux 1, October 2006.

[12] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of OpenMP compiler for an SMP cluster. In *EWOMP '99*, pages 32–39, September 1999.

[13] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.

[14] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. Stackthreads/MP : Integrating futures into calling standards. In *Principles Practice of Parallel Programming*, pages 60–71, 1999.

[15] Samuel Thibault. Un ordonnanceur flexible pour machines multiprocesseurs hiérarchiques. In *16ème Rencontres Francophones du Parallélisme*, pages 77–88, March 2005.