BubbleSched, plate-forme de conception d'ordonnanceurs de threads sur machines hiérarchiques

Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier

BubbleSched, plate-forme de conception d'ordonnanceurs de threads sur machines hiérarchiques

Samuel Thibault*, Raymond Namyst*, Pierre-André Wacrenier*

* *LaBRI - Université Bordeaux 1*
351 cours de la Libération — F-33405 Talence cedex
{samuel.thibault,raymond.namyst,pierre-andre.wacrenier}@labri.fr

RÉSUMÉ. *L'efficacité de l'exécution d'une application multithreadée irrégulière sur une architecture multiprocesseurs hiérarchique repose essentiellement sur la qualité de l'ordonnancement des threads et du placement des données. Pour obtenir d'excellentes performances, les programmeurs sacrifient souvent la portabilité de leur application en câblant dans celle-ci des stratégies de placement* ad hoc *dépendant fortement de l'architecture. Pour garantir la portabilité des performances, nous avons défini des abstractions appelées « bulles » capturant la nature structurée du parallélisme du calcul d'une part et modélisant l'architecture de la machine cible d'autre part. Un ensemble de primitives de haut niveau permet alors de définir simplement des ordonnanceurs dédiés, efficaces et portables. Nous justifions l'intérêt de cette approche et décrivons les techniques mises au point pour définir simplement de tels ordonnanceurs.*

ABSTRACT. *Exploiting full computational power of hierarchical multiprocessor machines with irregular multithreaded applications requires a very careful distribution of threads and data. To achieve most of the available performance, programmers often have to forget about portability and wire down* ad hoc *placement strategies that highly depend on the architecture. To guarantee the portability of performance, we have defined abstractions called "bubbles" for capturing both the hierarchical structure of the application's parallelism, and the hierarchical architecture of the targeted machine. We have defined a set of high level primitives to ease the implementation of dedicated, efficient and portable schedulers. We show the relevance of our approach and describe the mechanisms we developped for easily implementing such schedulers.*

MOTS-CLÉS : *Threads, Ordonnancement, Bulles, NUMA, SMP, Multicore*

KEYWORDS: *Threads, Scheduling, Bubbles, NUMA, SMP, Multicore*

3e soumission à *Technique et Science Informatique RENPAR*, le 6 août 2008

1. Introduction

Après s'être livrés durant 30 ans à une course à la fréquence, les constructeurs de microprocesseurs rivalisent maintenant à coups de puces multicore SMT. Ainsi, SUN commercialise des puces capables d'exécuter simultanément 32 threads sur 8 cores et IBM en est à 4 threads sur 2 cores. De leur côté AMD et INTEL commencent à produire des puces 4-cores et réfléchissent même à l'intégration d'une centaine de cores sur une seule puce. De plus, les serveurs de calculs sont souvent à mémoire partagée et organisés hiérarchiquement (SUN WILDFIRE, SGI ALTIX, BULL NOVASCALE ou encore IBM P560Q).

Exploiter efficacement de telles machines hiérarchiques est complexe. En particulier, l'ordonnanceur de threads est face à quelques dilemmes. Ainsi, comme sur de telles architectures le temps d'accès à la mémoire dépend de la position relative du processeur et du banc mémoire (le facteur NUMA), l'ordonnanceur doit faire en sorte qu'un thread soit exécuté à proximité des données qu'il manipule. Par ailleurs, les cores d'une même puce se partageant la bande passante mémoire, l'ordonnanceur doit distribuer les threads les plus gourmands en bande passante sur des puces différentes. Le partage de cache induit également des effets de symbiose entre threads, qu'il faut essayer de privilégier. Enfin, les cores de type SMT permettent de définir des processeurs logiques qui, en réalité, se partagent la puissance de calcul. Pour bien exploiter la technologie SMT l'ordonnanceur doit former des équipes de threads compatibles.

Ainsi, ordonnancer une application multithreadée en tenant compte de la hiérarchie mémoire, de celle des unités de calcul et des impératifs de l'application est un vrai défi. Certes les systèmes d'exploitation munis d'ordonnanceurs généralistes peuvent, dans une certaine mesure, prendre en compte l'état de la machine (*via* des mesures de performance) mais ils ne tiennent pas compte du comportement de l'application exécutée. Or celui-ci peut être irrégulier, imprédictible avant l'exécution (cas du maillage adaptatif) ce qui peut provoquer d'importants problèmes d'équilibrage de charge. Du point de vue des performances, la solution idéale serait que l'application décide elle-même de l'ordonnancement de ses threads et donc d'utiliser un ordonnanceur dédié à l'application. Cependant écrire un ordonnanceur, qui plus est portable, est techniquement difficile.

Pour remédier à ce problème de portabilité des performances, nous proposons une plate-forme permettant d'établir un dialogue entre l'environnement d'exécution et l'application afin d'aboutir automatiquement à un ordonnancement optimisé. Dans ce cadre, les programmeurs d'application (voire le compilateur ou encore le support exécutif) décrivent dynamiquement l'organisation hiérarchique de leurs tâches en regroupant de manière récursive celles qui partagent des intérêts communs, l'utilisation d'une même zone mémoire par exemple (affinité mémoire). Une API de haut niveau permet ensuite à d'autres programmeurs, de bibliothèques scientifiques spécialisées ou d'environnements de programmation parallèle, d'écrire des ordonnanceurs performants et portables manipulant cette organisation hiérarchique, sans pour autant se plonger dans de sordides détails techniques. Les programmeurs d'applications peuvent

alors facilement essayer et combiner ces ordonnanceurs pour ordonnancer leur application.

Dans cet article, nous présentons les principales approches actuelles d'exploitation des machines hiérarchiques. Nous proposons ensuite notre plate-forme, composée de deux nouveaux modèles pour décrire les tâches d'une application et la hiérarchie d'une machine, ainsi que d'une interface de programmation d'ordonnanceurs. Nous abordons ensuite les points-clés nécessaires à la conception d'un ordonnanceur de threads spécialisé et quelques éléments d'implémentation. Nous présentons aussi quelques résultats d'évaluation en situation. Enfin, nous comparons notre solution aux travaux apparentés avant de conclure.

2. Exploitation de machines hiérarchiques

Qu'elles soient simples SMP, NUMA, ou encore NUMA de multicores multithreadés, les machines multiprocesseurs sont difficiles à exploiter de manière optimale. Il faut trouver un compromis entre de nombreuses contraintes : favoriser les affinités entre threads, cache et mémoire, profiter de toute la puissance processeur, réduire les coûts de synchronisation et respecter les contraintes applicatives. Différentes approches ont été envisagées.

2.1. *Indications utiles à l'ordonnancement*

L'ordonnanceur dispose de quantité de contraintes, indications, constantes et métriques.

A l'exécution, il dispose d'informations sur la structure de la machine cible, depuis l'organisation des bancs mémoire pour les machines NUMA jusqu'à l'organisation des caches sur les puces multicores. Il est également possible de savoir si l'application s'exécute correctement à l'aide de *compteurs de performances*. Le nombre d'accès mémoire distants permet d'effectuer par exemple des migrations de pages mémoire (Marathe *et al.*, 2006). Le nombre de défauts de cache et d'instructions par cycle (IPC) permettent également de mesurer la concurrence au sein des processeurs.

Le *compilateur* peut aussi fournir des indications sur le comportement de l'application. En analysant les schémas d'accès aux données (Shen *et al.*, 2005), on peut estimer comment répartir threads et données pour favoriser les accès locaux et s'assurer que l'occupation mémoire sera possible sur la machine cible.

Enfin, les *programmeurs* eux-mêmes détiennent *beaucoup* d'informations. Ils savent si le schéma d'accès aux données est plutôt orienté producteur/consommateur ou bien complètement irrégulier par exemple. L'ordonnanceur peut alors coopérer avec le gestionnaire de mémoire pour établir des stratégies telles que « allocations entrelacées + ordonnancement centralisé », ou bien « allocations localisées + ordonnancement hiérarchisé ». Les programmeurs ont aussi une assez bonne idée du com-

portement des threads vis-à-vis de l'ordonnancement : s'ils s'endorment souvent ou bien consomment beaucoup de temps processeur par exemple, ce qui permet de trouver de bonnes combinaisons de politiques d'ordonnancement. Parfois, les schémas de synchronisation sont connus : on peut savoir que la libération d'un sémaphore devrait ordonnancer tout de suite le thread réveillé. Un thread endormi pour un court laps de temps sur un mutex ne devrait par ailleurs pas être compté comme vraiment inactif pour l'équilibrage de charge.

2.2. *Placement et ordonnancement prédéterminés*

Pour une machine et une application données, il est parfois possible de déterminer un ordonnancement des tâches et un placement des données adaptés à la machine et ses niveaux de hiérarchie. En exigeant de l'environnement d'exécution cet ordonnancement et ce placement précis, on obtient alors des performances excellentes (voire optimales). Le solveur PASTIX (Hénon *et al.*, 2000) illustre particulièrement bien cette approche : il résout de (très) grands systèmes linéaires creux par une méthode directe en calculant d'abord un ordonnancement statique des calculs par blocs et des communications, par l'intermédiaire d'une simulation utilisant une modélisation des opérateurs BLAS et de la communication sur l'architecture cible.

Pour mettre en œuvre ces ordonnancements, de nombreux systèmes (AIX, LINUX, SOLARIS, WINDOWS, etc.) permettent de fixer les threads d'un processus sur des ensembles de processeurs ainsi que de préciser sur quel nœud mémoire les allocations de zones de données doivent être effectuées. Dans la mesure où la machine est dédiée à l'application, l'ordonnancement est maintenu totalement sous contrôle en fixant sur chaque processeur exactement un thread. Pour passer d'une tâche à une autre, on utilise alors des changements de contexte explicites, les threads étant utilisés comme simples conteneurs de fil d'exécution.

2.3. *Placement et ordonnancement opportunistes*

Etudiés depuis 20 ans, les algorithmes gloutons (appelés *Self-Scheduling* (Tang *et al.*, 1986)) constituent des solutions dynamiques, souples et portables pour la parallélisation des boucles. Quelle que soit la machine sur laquelle l'application s'exécute, un algorithme de *Self-Scheduling* s'occupe de l'ordonnancement des threads et du placement des données. Les ordonnanceurs des systèmes d'exploitation actuels reposent sur ces algorithmes.

Le principe de base est l'utilisation d'une liste unique de tâches prêtes : l'ordonnanceur se contente de « piocher » dans celle-ci le prochain thread à ordonnancer, ce qui permet de répartir le travail entre les différents processeurs. Le dernier placement de chaque thread est mémorisé afin de le rejouer le plus possible, pour éviter les contre-performances dues aux défauts de cache. Ces techniques sont utilisées dans les systèmes d'exploitation LINUX 2.4 et WINDOWS 2000 (Russinovich, 1997).

Cependant, une unique liste de threads pour toute une machine constitue un goulet d'étranglement, en particulier lorsque la machine possède de nombreux processeurs.

Pour éviter cette contention, les algorithmes *Guided Self-Scheduling* (GSS) (Polychronopoulos *et al.*, 1987) et *Trapezoid Self-Scheduling* (TSS) (Tzen *et al.*, 1993) font en sorte que chaque processeur s'approprie tout une partie du travail total lorsqu'il n'en a plus, au risque de provoquer d'éventuels déséquilibres. Les algorithmes *Affinity Scheduling* (AFS) (Markatos *et al.*, 1994) et *Locality-based Dynamic Scheduling* (LDS) (Li *et al.*, 1993) utilisent quant à eux des listes de tâches locales à chaque processeur. A court de travail, ce dernier en « vole » à un autre processeur, le plus chargé, par exemple. Ce dernier type d'algorithme est utilisé par les systèmes d'exploitation actuels (par exemple LINUX 2.6 (Lin, n.d.), CELLULAR IRIX (Whitney *et al.*, 1997), FREEBSD 5.0 (Roberson, 2003)) qui y ajoutent des éléments de rééquilibrage : tout processus créé est placé sur le processeur le moins chargé ; régulièrement, le processeur le plus chargé confie une partie de son fardeau au processeur le moins chargé.

Cependant, à partir d'un certain nombre de processeurs, et surtout pour les machines NUMA, ce n'est plus suffisant. WANG *et al*. proposent *Clustered Affinity Scheduling* (CAFS) (Wang *et al.*, 1997) qui regroupe les p processeurs par groupes de $\sqrt{p}$. Lorsqu'un processeur est inoccupé, plutôt que de chercher à « voler » le travail du processeur le plus chargé de la machine, il cherche à « voler » celui du processeur le plus chargé de son groupe seulement, localisant ainsi les accès aux listes. De plus, en alignant les groupes sur les nœuds NUMA, on localise aussi le placement des données. Enfin, dans *Hierarchical Affinity Scheduling* (HAFS) (Wang *et al.*, 2000), tout groupe entier inoccupé « vole » du travail au groupe le plus chargé. Cette dernière voie est en cours d'exploration pour le développement des systèmes d'exploitation tels que LINUX et FREEBSD.

2.4. *Placement et ordonnancement négociés*

Des approches intermédiaires entre ordonnancement prédéterminé et opportuniste existent. Les extensions de langage comme OPENMP (Mattson *et al.*, 1999), HPF (*High Performance Fortran*) (Schreiber, 1996) ou UPC (*Unified Parallel C*) (Carlson *et al.*, 1999) permettent en effet de programmer les machines parallèles à l'aide de simples indications. Une boucle `for` sera par exemple annotée pour que les différentes itérations soient automatiquement réparties entre différents threads, profitant ainsi des processeurs disponibles sur la machine. Une matrice de HPF pourra être annotée pour être automatiquement découpée en autant de blocs qu'il y a de processeurs, chaque processeur étant alors chargé d'effectuer les calculs concernant la partie qui lui a été attribuée.

C'est au compilateur que revient le travail de placement et d'ordonnancement. Pour cela, il ajoute le code déterminant l'environnement d'exécution (nombre de processeurs) et compile le programme de manière suffisamment générique pour pouvoir

s'adapter aux différentes architectures parallèles. En particulier, il faudra gérer les threads nécessaires aux boucles parallélisées ou aux calculs répartis, voire gérer les échanges de données entre processeurs (cas des matrices réparties de HPF)... L'expressivité est cependant limitée : on obtient un programme où le parallélisme est surtout une succession ou une imbrication de blocs « `Fork-Join` ». Le programmeur ne peut pas exprimer un parallélisme de contrôle irrégulier (avec des tâches asynchrones par exemple).

Il est aussi possible d'écrire des applications capables de s'adapter elles-mêmes à la machine supportant leur exécution. Les applications peuvent en effet prendre connaissance de leur environnement, les systèmes d'exploitations modernes fournissant une description complète de la machine les exécutant (voir les bibliothèques `lgroup` (Sun, n.d.) pour SOLARIS ou `numa` (Lin, n.d.) pour LINUX). L'application peut ainsi déterminer le nombre de processeurs (ce qui est commun), mais aussi obtenir la hiérarchie des nœuds NUMA, leurs nombres de processeurs et quantités de mémoire vive. De plus, ces mêmes systèmes d'exploitation permettent de déterminer la politique d'allocation mémoire (nœud mémoire précis, *first touch* ou *round robin*) et de fixer les threads sur des ensembles de processeurs (notion de *cpuset*). L'application contrôle alors à volonté (voire exactement) le placement des threads et de la mémoire, mais elle a alors l'entière responsabilité de la répartition des threads sur les processeurs.

2.5. *Discussion*

Les trois catégories dans lesquelles nous avons choisi de classer les approches existantes peuvent être caractérisées ainsi.

– *L'approche prédéterminée* permet d'obtenir des performances excellentes. Elle n'est cependant portable que lorsqu'elle est appliquée aux problèmes réguliers, c'est-à-dire que leur traitement dépend de la structure des données et non pas des données elles-mêmes.

– *L'approche opportuniste* passe en général assez bien à l'échelle, mais elle ne tient pas compte des affinités entre les tâches et n'obtient donc pas d'excellentes performances lorsque celle-ci deviennent importantes.

– *L'approche négociée* permet à l'application de s'adapter à la machine sous-jacente, mais demande, pour être flexible, l'écriture d'une partie de l'ordonnanceur. Cependant, il ne paraît pas raisonnable de demander à des programmeurs d'application de réaliser complètement de tels ordonnanceurs embarqués, performants et portables. En effet, la réalisation d'un ordonnanceur est un travail très technique qui réclame un savoir-faire de spécialiste.

Le problème est donc de tirer parti des informations disponibles afin d'établir un ordonnancement qui est nécessairement un compromis puisque, par exemple, distribuer les threads sur toute la machine permet de profiter des processeurs mais concentrer les threads autour des données traitées permet de gagner en affinités. Certaines

parties d'application peuvent même réclamer des politiques d'ordonnancement différentes. Les approches actuelles ne sont donc pas à même de prendre en compte ces contraintes, et de manière plus générale, manquent d'interaction avec les programmeurs et le compilateur.

Notre objectif est de proposer une nouvelle approche négociée en distinguant clairement d'une part, le rôle de l'environnement d'exécution (ou du compilateur), qui peut fournir des informations permettant de guider l'ordonnancement, et, d'autre part, le rôle de l'ordonnanceur qui, en contrôlant précisément l'allocation des ressources, est le mieux à même de prendre les décisions de placement appropriées lorsque c'est nécessaire (e.g. en réaction à un déséquilibre de charge par exemple).

3. BubbleSched : un modèle et une plate-forme pour l'élaboration d'ordonnanceurs portables

Notre proposition s'appuie sur de nouvelles abstractions pour renforcer la collaboration entre l'application, l'ordonnanceur et la machine : des outils pour structurer, qualifier le parallélisme de l'application et des outils pour piloter son ordonnancement (sans avoir à réécrire un ordonnanceur de bas niveau). Pour des raisons de portabilité, ces outils devraient évidemment s'appuyer sur des abstractions de haut niveau de la machine sous-jacente.

3.1. *Des bulles pour exprimer la structure d'une application*

Nous demandons à l'application de modéliser l'organisation générale de ses threads au moyen d'ensembles imbriqués appelés **bulles**[1].

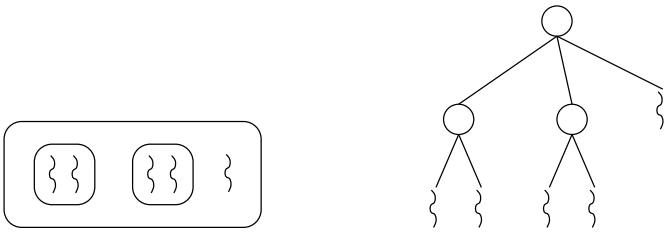
La figure 1 illustre une telle modélisation : l'application regroupe ses 4 threads de calcul par paires (car chacune travaille sur une matrice différente par exemple), et accompagne ces deux paires d'un thread de communication (vers d'autres machines par exemple).

La notion de bulle est à interpréter comme une *classe d'équivalence par rapport à une relation d'affinité donnée*, l'imbrication des bulles signifiant le *raffinement* de la relation par une autre relation. En effet, différentes relations d'affinités sont à considérer.

Partage de données

Il est profitable de rapprocher des threads travaillant sur les mêmes données afin de profiter des effets de cache ou du moins éviter qu'ils se retrouvent sur différents nœuds NUMA et soient alors pénalisés par le facteur NUMA.

1. De manière relativement analogue à certaines bibliothèques de communication telles que MPI qui demandent à l'application d'indiquer des *communicateurs* : les groupes de machines qui communiqueront ensemble.



a. Représentation ensembliste. b. Représentation arborescente.

```
marcel_t comp_thread[4], comm_thread;
marcel_bubble_t bubbles[2], bubble;

marcel_bubble_init(&bubbles[0]);
marcel_bubble_init(&bubbles[1]);
marcel_bubble_init(&bubble);

marcel_bubble_insertthread(&bubbles[0],&comp_thread[0]);
marcel_bubble_insertthread(&bubbles[0],&comp_thread[1]);
marcel_bubble_insertthread(&bubbles[1],&comp_thread[2]);
marcel_bubble_insertthread(&bubbles[1],&comp_thread[3]);

marcel_bubble_insertbubble(&bubble,&bubbles[0]);
marcel_bubble_insertbubble(&bubble,&bubbles[1]);
marcel_bubble_insertthread(&bubble,&comm_thread);
```

c. Code source correspondant.

Figure 1. *Expression des relations entre threads*

Opérations collectives

Il est intéressant d'optimiser l'ordonnancement des ensemble des threads concernés par une même opération collective telle une barrière de synchronisation qui permet de s'assurer que les threads concernés ont tous terminé la première partie d'un calcul avant d'entamer la seconde.

SMT

Certains couples de threads sont à même d'exploiter très efficacement, sans interférer, la technologie du *Simultaneous MultiThreading* lorsqu'ils sont exécutés en parallèle sur les deux processeurs logiques d'un même processeur physique.

D'autres relations d'équivalence sont possibles, on peut vouloir exprimer des qualités de parallélisme (voire de séquentialité), de préemption ou de priorité, par exemple.

Cette notion ressemble à ce que propose le système SOLARIS, qui permet aux applications de fixer leurs threads sur des *LWP* (*Light-Weight Processes*). Le système

s'occupe de répartir les *LWP* sur les différents processeurs, tandis que la librairie de thread utilisateur peut, sur chaque *LWP*, ordonnancer les threads qui ont été fixés dessus.

3.2. *Des listes de tâches pour coller à la structure de la puissance de calcul*

D'après Dandamudi *et al.* (1997), il est plus performant, en général, de hiérarchiser les listes de tâches prêtes que de les distribuer simplement entre processeurs. Aussi, des ordonnanceurs à deux niveaux de listes ont été développés (Fukuda *et al.*, 1993; Oguma *et al.*, 2001). Qui plus est, ces ordonnanceurs facilitent la fixation de tâches sur un processeur donné. Nous avons repris ce point de vue, en le généralisant.

En effet, nous modélisons les machines hiérarchiques à l'aide d'une hiérarchie de listes de tâches prêtes. A chaque élément de chaque étage de la hiérarchie de la machine correspond (*bijectivement*) une liste de tâches. La figure 2 montre une machine hiérarchique et sa modélisation. La machine en entier, chaque nœud NUMA, chaque puce (*core*), chaque processeur physique SMT et chaque processeur logique possède ainsi une liste de tâches prêtes. L'ordonnanceur de base est alors de type Self-Scheduling : chaque processeur examine les listes qui le « couvrent » à la recherche du thread le plus prioritaire.

L'identification entre élément physique et liste de tâches permet ainsi de déterminer l'espace d'exécution d'une tâche donnée : placée sur une liste associée à une puce physique, cette tâche pourra être exécutée par tout processeur de cette puce ; placée sur la liste globale elle pourra être exécutée par tout processeur de la machine.

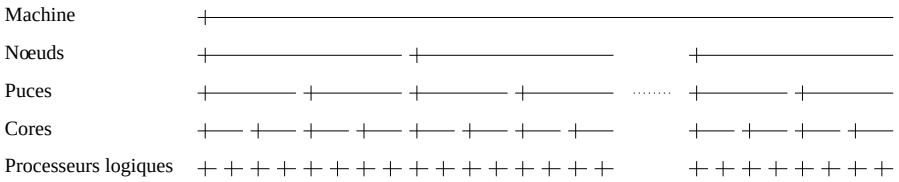
Toute organisation peut être prise en compte. Il est ainsi aisé de modéliser un ensemble de machines NUMA reliées par un réseau à capacité d'adressage (tel que SCI [2] (Solutions, 1996)) : il suffit simplement d'ajouter le niveau « réseau ». Bien entendu, suivre l'organisation physique de la machine n'est pas obligatoire. Par exemple, les *clusters* artificiels de processeurs introduits par CAFS, présenté en 2.3, peuvent eux aussi être modélisés.

3.3. *Des outils pour implémenter des stratégies d'ordonnancement*

Les applications et leur comportement sont si divers qu'il semble irréaliste d'espérer définir un ordonnanceur « universel » capable de satisfaire les besoins de la majorité des applications. En outre, le développement d'un ordonnanceur requiert un savoir-faire technique élevé et nécessite de prendre en considération un nombre important de paramètres qui n'ont pas de lien direct avec les aspects algorithmiques de l'ordonnancement. Pour ces raisons, nous avons donc développé une plate-forme fa-

2. Un réseau SCI permet de définir des segments de mémoire partagée par plusieurs machines de façon transparente, la machine SEQUENT NUMA-Q (Lovett *et al.*, 1996) utilise cette technique.

a. Une machine NUMA de multicores hyper-threadés.

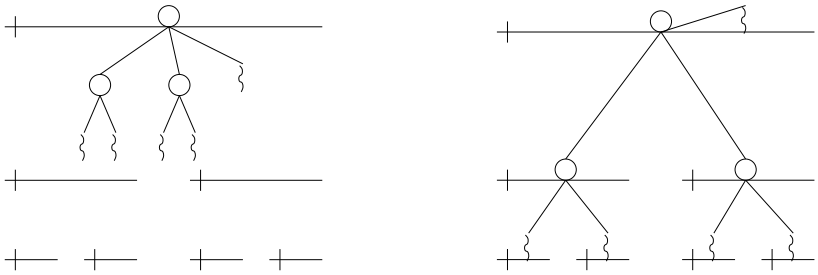


b. Modélisation par des listes de tâches.

Figure 2. *Une machine très hiérarchique et sa modélisation*

cilitant l'écriture d'ordonnanceurs efficaces, flexibles et portables sur les machines hiérarchiques. Cette plate-forme permet au programmeur de se focaliser uniquement sur les aspects algorithmiques de l'ordonnancement.

Voici un exemple illustrant toute la puissance du modèle des *bulles*. Par défaut, les bulles (et donc les threads contenus dedans) sont placées au démarrage sur la liste de la machine entière. Lorsque l'on exécute l'exemple précédent de la figure 1b sur une machine double bi-core, on obtient la figure 3a : la hiérarchie de bulles (et donc tous les threads) reste en haut de la hiérarchie de listes. Une telle distribution permet certes d'utiliser tous les processeurs de la machine, puisque tous ont l'opportunité d'exécuter n'importe quel thread. Cela ne prend cependant pas en compte les affinités entre threads et processeurs, puisqu'aucune relation entre eux n'est utilisée. A l'inverse, la figure 3b montre comment les threads peuvent être distribués spatialement en prenant fortement les affinités en compte, puisque les threads sont en fait 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.



a. Bonne utilisation processeur, affinités non prises en compte.

b. Utilisation distribuée des processeurs, affinités correctement prises en compte.

Figure 3. *Distributions possibles des threads et bulles sur la machine*

Nous avons développé une interface de programmation pour répartir facilement bulles et threads sur les listes afin d'obtenir des distributions telles que celle de la figure 3b. Threads et bulles sont indifféremment considérés comme **entités**, une bulle contenant simplement des entités (threads ou autres bulles). Des primitives sont alors fournies pour manipuler les entités sur les listes de la figure 3b (appelées `runqueues`), voir figure 4. On peut accéder aux listes de manière portable à l'aide d'un simple tableau de pointeurs (indexé en « hauteur ») sur des tableaux (indexés en « largeur ») de listes, ou directement certains niveaux physiques utiles, `node_rq` si l'on veut énumérer les nœuds NUMA pour gérer finement les allocations mémoire, par exemple. Des pointeurs père/fils sont également disponibles pour des parcours arborescents. Pour gérer la concurrence, il est possible de verrouiller finement les listes, mais les programmeurs pourront parfois préférer se contenter d'un verrouillage total pendant leurs manipulations (`all_lock`). On peut alors énumérer les entités d'une liste, et décider d'en enlever (`get_entity`) pour les ajouter sur une autre liste (`put_entity`), exprimant ainsi une *migration* d'entité entre deux listes de la machine.

Ecrire un ordonnanceur de haut niveau se réduit alors à écrire quelques fonctions-clés effectuant de telles migrations. La fonction `ma_bubble_schedule()` est appelée lorsque l'ordonnanceur de base rencontre une bulle lors de sa recherche du thread suivant à exécuter. L'implémentation par défaut recherche simplement un thread dans la bulle (ou l'une de ses sous-bulles), et l'exécute. La fonction `ma_bubble_tick()` est appelée lorsque le quantum de temps d'une bulle expire, et permet ainsi des opérations périodiques sur les bulles avec une notion de temps relative à la bulle. `ma_bubble_idle)`, enfin, est appelée sur un processeur lorsqu'il est inactif. Bien sûr, des threads peuvent aussi être utilisés pour effectuer des opérations en arrière-plan. Au final, les programmeurs peuvent manipuler la répartition des threads avec un haut niveau d'abstraction en décidant du placement des entités sur les listes.

```
/* Une entité est un thread ou une bulle
enum entity entity_type(entity_t *e);
bubble_t *bubble_entity(entity_t *e);
thread_t *thread_entity(entity_t *e);
bubble_foreach(bubble_t *b, entity_t **e)

/* Niveaux de listes. Accès portable. */
runqueue_t *runqueues[];
/* Accès direct à certains niveaux utiles. */
runqueue_t main_rq, *node_rq, *cpu_rq;
/* Verrouillage. */
void runqueue_lock(runqueue_t *rq);
void runqueue_unlock(runqueue_t *rq);
void all_lock(); all_unlock();

/* Opérations sur listes. */
int runqueue_empty(runqueue_t *rq);
entity_t *runqueue_entry(runqueue_t *rq);
runqueue_foreach(runqueue_t *rq, entity_t **e)
/* Retirer un élément. */
void get_entity(entity_t *e);
/* Déposer un élément sur une autre liste. */
void put_entity(entity_t *e, runqueue_t *rq);
```

Figure 4. *Interface de programmation*

3.4. *Exemples d'implémentations d'algorithmes d'ordonnancement*

Notre interface de programmation permet de développer des « ordonnanceurs à bulles » très variés.

3.4.1. *Ordonnancement à éclatement*

Un premier exemple d'ordonnanceur possible est un « algorithme à éclatement » : un algorithme de Self-Scheduling « tire » vers les processeurs les bulles, qui « éclatent » (*i.e.* déversent leur contenu sur la liste), d'une manière opportuniste, lorsqu'elles ont atteint un niveau hiérarchique donné par les programmeurs. Pour implémenter cela, la fonction `ma_bubble_schedule()` de la figure 5 tire la bulle vers le processeur, et le contenu est déversé. En quelques itérations, bulles et threads sont répartis comme sur la figure 3b. De plus, les bulles sont automatiquement régénérées à l'aide de la fonction `ma_bubble_tick()` : leur contenu originel y est replacé, et elles sont remises sur la liste de la machine, pour être de nouveau distribuées. Cela permet d'adapter automatiquement la distribution à une nouvelle charge de calcul, tout en prenant en compte les affinités.

```
thread_t *ma_bubble_schedule (entity_t *e, runqueue_t **sched_rq) {
  runqueue_t *rq = *sched_rq;
  if (e->level > rq->level) {
    /* Entité trop haute, la faire descendre. */
    ma_get_entity(e);
    runqueue_unlock(rq);
    for(rq = cpu_rq[cpu_self()]; e->level < rq->level;
        rq = rq->father) { }
    runqueue_lock(rq);
    ma_put_entity(e, rq);
    *sched_rq = rq;
  }
  /* Un thread, le retourner pour exécution. */
  if (entity_type(e) == THREAD) return thread_entity(e);
  /* Une bulle, l'éclater et laisser l'ordonnanceur de base
     recommencer. */
  bubble_t *b = bubble_entity(e);
  bubble_foreach(b, &e) { ma_get_entity(e); ma_put_entity(e, rq); }
  return NULL;
}
```

Figure 5. *Ordonnancement à éclatement*

3.4.2. *Ordonnancement de gangs*

Avec l'émergence dans les années 1980 des réseaux de machines multiprocesseurs, Ousterhout (1982) propose de regrouper les threads et données par affinités sous forme de *gangs* et d'ordonnancer non plus des threads sur des processeurs mais des gangs sur des machines. Dans notre environnement, pour réaliser un *gang-scheduler*, on associe une bulle à chaque gang, et l'on laisse tourner un démon écrit en une simple douzaine de lignes, voir figure 6a. Il utilise une liste `nosched_rq` supplémentaire, non consultée par l'ordonnanceur de base, où il met de côté toutes les bulles (i.e. les gangs) sauf une, qu'il laisse sur la liste principale pendant un certain laps de temps, avant de recommencer pour placer une autre bulle. Ceci revient à un ordonnancement *temporel*. Il nous semble qu'un programmeur peut modifier un tel algorithme sans connaissances techniques.

Une utilisation intéressante de ce gang scheduler est l'émulation équitable d'un réseau de machines virtuelles sur une même machine physique. Chaque machine virtuelle est représentée par une bulle. Puisque le gang scheduler, en arrière-plan, donne tour à tour à chaque bulle une tranche de temps, il ordonnance ainsi équitablement les machines virtuelles. Nous avons réalisé une expérience où trois gangs de calcul intensif se partagent une machine à quatre processeurs. On observe sur la figure 6b que la machine est bien partagée de manière équitable : le gang 0 formé de 5 threads (notés `0-[0-4]`) obtient, tout comme les autres gangs, à peu près 133 % de temps processeur (sur 400 %).

```
runqueue_t nosched_rq ;
while(1) {
  runqueue_lock(&main_rq) ;
  runqueue_lock(&nosched_rq) ;
  /* Remettre toutes les bulles de côté. */
  runqueue_foreach(&main_rq, &e) {
    get_entity(e) ;
    put_entity(e, &nosched_rq) ;
  }
  /* Placer la bulle suivante
  sur la liste principale. */
  if (!runqueue_empty(&nosched_rq)) {
    e = runqueue_entry(&nosched_rq) ;
    get_entity(e) ;
    put_entity(e, &main_rq) ;
  }
  runqueue_unlock(&main_rq) ;
  runqueue_unlock(&nosched_rq) ;
  /* Laisser cette bulle s'exécuter un peu. */
  delay(timeslice) ;
}
```

a. Code source.

```
            nom   pr   % proc   s   proc
 gang scheduler   42      0.0   I      0
            0-0   43     26.4   R      3
            0-1   43     26.6   R      0
            0-2   43     26.8   R      2
            0-3   43     25.6   R      1
            0-4   43     26.6   R      3
            1-0   43     21.9   R      2
            1-1   43     22.2   R      0
            1-2   43     22.5   R      3
            1-3   43     22.1   R      2
            1-4   43     21.9   R      0
            1-5   43     22.6   R      1
            2-0   43     19.2   R      3
            2-1   43     18.9   R      1
            2-2   43     19.3   R      0
            2-3   43     19.8   R      2
            2-4   43     19.1   R      3
            2-5   43     19.3   R      1
            2-6   43     19.2   R      2
```

b. Statistiques d'exécution (outil *a la* `top`)

Figure 6. *Un gang scheduler de base*

Dans ce modèle originel du gang scheduling, des processeurs peuvent rester inactifs parce qu'une même machine ne peut exécuter qu'un gang à la fois, même si celui-ci est « petit ». Feitelson *et al.* (1996) proposent un contrôle hiérarchique des processeurs, pour pouvoir exécuter plusieurs gangs sur la même machine. Une telle approche peut être facilement implémentée en exécutant un thread de gang scheduling pour chaque nœud de la hiérarchie de la machine. Une certaine synchronisation est nécessaire entre ces threads, mais cela peut être facilement effectué à l'aide de sémaphores usuels.

3.4.3. *Ordonnancement par vol de travail*

Un de nos algorithmes en cours de développement est basé sur le vol de travail : la hiérarchie de bulle est d'abord placée sur la liste du processeur 0. Lorsque la fonction `ma_bubble_idle()` est appelée par un processeur inactif, elle utilise les pointeurs père/fils des listes pour chercher du travail à voler localement d'abord (sur la liste de threads de l'autre processeur du même core par exemple), puis plus globalement, jusqu'à trouver du travail. Savoir quel « vol » effectuer est un problème algorithmique non trivial : seule une partie de la hiérarchie de bulles devrait être tirée vers le processeur inactif. De plus, la structure de la hiérarchie devrait être prise en compte : plus qu'une simple migration de threads, il s'agit donc véritablement d'une sorte d'« étirement local de l'arbre des bulles » le long de la machine. Tous les attributs et statistiques attachés aux bulles décrits dans la section précédente devraient également être soigneusement pris en compte de manière heuristique pour obtenir une répartition bien adaptée à l'application. En effet, selon les applications il faudra privilégier plutôt une répartition de l'occupation mémoire, ou parfois plutôt de la bande passante nécessaire ; un compromis est à trouver. Ces problèmes ne sont cependant que purement algorithmiques : il n'y a plus de problème technique.

3.4.4. *Ordonnancements mixtes*

Les programmeurs peuvent même combiner tous ces ordonnanceurs. Une partie de la machine peut exécuter une partie de l'application avec certaines politiques, pendant que d'autres parties de la machine exécutent d'autres parties de l'application avec d'autres politiques, obtenant ainsi une combinaison *spatiale*. En étendant l'implémentation du gang scheduling décrite ci-dessus, chaque gang peut avoir sa propre politique d'ordonnancement. Ou bien l'application peut demander différents ordonnancements selon ses phases de calculs. On obtient ainsi une combinaison *temporelle* d'ordonnanceurs. Des priorités fortes définies par le programmeur peuvent également fournir des *niveaux* d'ordonnancement : une partie L de faible priorité de l'application peut être préemptée par une partie H de priorité plus forte de l'application, qui peut alors être ordonnancée avec une autre politique ; mais si une partie de la machine n'est pas utilisée par la partie H, la partie L peut continuer à être ordonnancée sur les processeurs restants, avec sa politique habituelle.

4. Implémentation

D'un point de vue abstrait, écrire un ordonnanceur c'est avant tout écrire une fonction qui sera appelée en certaines occasions afin de déterminer le prochain thread à exécuter. La concrétisation de ce point a cependant retenu quelques attentions de la part de la communauté, de nombreux écueils étant à éviter. Un véritable savoir-faire s'est donc peu à peu dégagé. Aussi, avant de présenter l'implémentation de notre plate-forme, nous rappelons quelques éléments du savoir-faire commun quant à la conception des ordonnanceurs que nous avons, nous aussi, appliqués.

4.1. *Savoir-faire technique et algorithmique commun à tout ordonnanceur*

Un point essentiel est de prendre correctement en compte les appels concurrents aux fonctions de l'ordonnanceur. Pour cela, il faut déterminer les articulations nécessaires entre les fonctions de l'ordonnanceur. Par exemple, lorsqu'un thread T essaie de prendre un mutex occupé, il doit s'endormir par un appel à la fonction `sleep()` qui doit appeler une fonction `find_next()` pour véritablement suspendre le thread T en en trouvant un autre à exécuter. La fonction `sleep()` doit cependant faire attention à la possibilité d'un appel concurrent à la fonction `wake_up()`, effectué sur un autre processeur par un autre thread qui est précisément en train de libérer le mutex que T essayait de prendre ! De tels problèmes, courants dans les ordonnanceurs, induisent des bugs difficiles à cerner, au caractère particulièrement aléatoire lorsqu'il s'agit d'exploiter un nombre conséquent de listes de tâches comme sur une machine NUMA.

Ecrire un ordonnanceur avec une implémentation en elle-même *portable* est également difficile : pour faire fonctionner un ordonnanceur de threads, il faut de nombreuses connaissances sur le système d'exploitation et le processeur cibles. Par exemple, préempter un thread actif pour basculer l'exécution sur un autre est une opération dont l'implémentation dépend du système d'exploitation et du processeur. Pour respecter les conventions binaires (ABI) du système, il faut correctement sauver et restaurer l'état des threads : non seulement les registres du processeur, mais aussi les objets spéciaux tels que la variable `errno` [3].

Enfin, notons que la *qualité* de l'implémentation est en elle-même importante si l'on veut obtenir un ordonnanceur *efficace* et *passant à l'échelle* : comme le détaille Edler (1995), toutes les briques de bases doivent être soigneusement implémentées. Les verrous rotatifs, par exemple, devraient non seulement être écrits en assembleur (ce qui implique des problèmes de portage), mais doivent aussi être distribués afin d'éviter des contentions mémoire (Radović *et al.*, 2002).

3. L'introduction récente du support *Thread-Local Storage* (TLS) dans les systèmes d'exploitation n'a fait que complexifier encore ceci en impliquant par exemple le registre de segment `gs` sur les machines x86.

4.2. *La bibliothèque de threads* MARCEL

Afin de valider notre approche sur des architectures et des systèmes variés nous avons implémenté la plate-forme BubbleSched au sein de l'ordonnanceur MARCEL (Namyst, 1997; Danjean, 2004). MARCEL est une bibliothèque portable[4] de threads à deux niveaux : elle fixe un thread de niveau noyau sur chaque processeur, et effectue ensuite des changements de contexte entre des thread utilisateurs. Ainsi, en supposant qu'aucune autre application ne tourne sur la machine, MARCEL garde un contrôle complet sur l'ordonnancement des threads sur les processeurs depuis l'espace utilisateur et sans aucune aide du noyau. Ceci permet ainsi un déploiement aisé sur une grande variété de machine de test sans privilèges administratifs particuliers. Pour éviter des problèmes de blocage des threads noyaux lorsque l'application effectue des appels système bloquants, MARCEL utilise les *Scheduler Activations* (Danjean *et al.*, 2000) lorsqu'elles sont disponibles, ou bien détourne simplement ces appels bloquants.

4.3. *Implémentation de BubbleSched*

Implémenter la plate-forme BubbleSched consiste à intégrer, au sein de la bibliothèque visée, les appels aux fonctions cités dans la section 3.3. Dans le cadre de la bibliothèque MARCEL, ces fonctions sont ainsi exécutées au même niveau que l'application, en espace utilisateur. Les « threads d'arrière-plan » sont ainsi simplement des threads MARCEL. Bien sûr, cette implémentation pourrait aussi être faite dans d'autres cadres moins portables tels que SPIN, l'ExoKernel, etc.

Pour faciliter la programmation, de la même façon que threads et bulles sont indifféremment considérés comme entités, listes et bulles sont considérés comme *conteneurs* : ils contiennent une liste d'entités.

La gestion du verrouillage des conteneurs est particulièrement importante : pour des raisons de performances, il devrait être le plus distribué possible. Cependant, distribuer les verrous impose de faire attention aux interblocages. Notre convention pour verrouiller plusieurs conteneurs est de le faire de haut en bas : par exemple, une liste d'un core, puis les listes des processeurs, puis les bulles placées sur ces listes, puis les bulles contenues dans ces bulles, etc. Dans certaines situations, il est alors nécessaire de libérer un verrou pour pouvoir en prendre un autre. De telles situations se sont cependant révélées rares, dans l'implémentation actuelle cela ne se produit que deux fois. Une autre convention aurait permis d'éviter ces deux cas, mais en aurait par ailleurs apporté beaucoup d'autres.

Puisque l'on veut que les threads de haute priorité s'exécutent le plus tôt possible même lorsqu'ils sont situés sur la liste principale de la machine, un processeur inactif doit parcourir toutes les listes qui le couvrent. Deux passes sont en fait nécessaire pour

4. MARCEL fonctionne sur un grand nombre de systèmes (Linux, AIX, Darwin, Solaris, OSF, etc.) et de processeurs (x86, x86_64, Itanium, PowerPC, etc.).

trouver l'entité (thread ou bulle) qui a la priorité maximale parmi toutes ces listes. La première passe trouve rapidement la liste qui contient l'entité de priorité la plus forte, sans prendre de verrou. Cette liste est alors verrouillée, et une deuxième passe permet de s'assurer que la liste choisie contient bien encore une entité avec cette priorité, au cas où un autre processeur l'aurait prise entre-temps. Si l'entité sélectionnée est un thread, il est enlevé de la liste, qui peut alors être déverrouillée. Ce thread est ordonnancé et remet le thread précédent sur sa propre liste (avec verrouillage). Si par contre l'entité est une bulle, la fonction `ma_bubble_sched()` est appelée. Cette fonction peut soit retourner un thread à ordonnancer, soit effectuer quelques opérations de rééquilibrage, et l'ordonnanceur est redémarré avec le nouvel équilibrage.

La gestion du placement des bulles et des threads est un problème ardu. Migrer des threads peut être particulièrement délicat : il peut s'agir de placer sur une liste un thread contenu dans une bulle, alors qu'il est encore en train de s'exécuter sur une autre liste. Il est ainsi nécessaire de mémoriser ces trois informations pour pouvoir gérer tous les cas : le conteneur *initial* d'une entité est celui que les programmeurs ont défini initialement, et doit être préservé puisqu'il représente l'affinité entre les entités (les programmeurs peuvent cependant le changer si ces affinités elles-mêmes évoluent). Le conteneur d'*ordonnancement* d'une entité est le conteneur où l'entité devrait être ordonnancée. Elle n'est pas forcément déjà en train de s'y exécuter, mais elle devrait y parvenir dans un délai bref. Le conteneur d'*exécution* est enfin le conteneur où une entité s'exécute et pour lequel elle contribue aux statistiques.

Enfin notons que l'introduction de l'API de programmation des ordonnanceurs n'a pas eu d'impact négatif sur les performances. En effet, nous avons mesuré l'impact de notre implémentation sur un PENTIUM IV XEON à 2,66 GHz, voir table 1. On observe pour la recherche du prochain thread à exécuter (temps *Yield*) et pour le changement de contexte proprement dit (temps *Switch*) un léger surcoût, mais les résultats restent très bons comparativement à ceux de NPTL (LINUX 2.6).

	Yield (ns)	Switch (ns)	Total
MARCEL (originel)	240	270	510
MARCEL (bulle)	247	284	531
NPTL (Linux 2.6.15)	820	410	1230

Tableau 1. *Coût de l'ordonnanceur modifié de* MARCEL. *Yield : parcours des listes seulement ; Switch : synchronisation et changement de contexte*

5. Evaluation

Nous montrons d'abord sur une application relativement simple que l'utilisation de bulles permet d'atteindre les performances d'un ordonnancement *ad hoc*. Ensuite, sur une application plus complexe, nous montrons que paramétrer un ordonnanceur de bulles permet d'obtenir des résultats différents, l'utilisateur final pouvant ainsi choisir l'ordonnanceur le plus adapté à son application.

	temps	*speedup*
Séquentiel	250,2s	
MARCEL simple	23,65s	10,58
MARCEL fixé	15,82s	15,82
MARCEL bulles	15,84s	15,80

Figure 7. *Performances de l'application* CONDUCTION *selon l'approche. Le temps séquentiel sert de référence pour le* speedup

5.1. *Simulation de Conduction*

Dans le cadre d'un travail de thèse au CEA DAM, Pérache (2005) utilise notre ordonnanceur dans afin de comparer l'efficacité de différentes stratégies d'ordonnancement de l'exécution d'une application de calcul de conduction de chaleur. Cette application effectue une simulation physique en parallèle sur des brandes d'un grand maillage 3D, avec propagation des résultats intermédiaires entre les tranches à chaque pas de temps. Le placement relatif entre données et threads ainsi que les communications entre processeurs sont donc assez importants. Les résultats obtenus sont présentés figure 7. La machine cible est une machine ccNUMA BULL NOVASCALE totalisant 16 processeurs et 64 Go de mémoire. Ils sont répartis sur 4 nœuds NUMA de 4 processeurs et 4 Go de mémoire. Pour un processeur donné, accéder à la mémoire du nœud où il est situé est de l'ordre de 3 fois plus rapide qu'accéder à la mémoire d'un autre nœud (facteur NUMA).

Dans la version MARCEL simple, le maillage est découpé en autant de bandes qu'il y a de processeurs et un ordonnancement opportuniste est utilisé. Pour obtenir un placement des threads et du maillage optimisé, il est possible de fixer de manière non portable les threads sur les processeurs, ce qui localise les accès mémoire : un thread et ses données restent sur un même nœud. On obtient alors de bien meilleures performances. Nous avons enfin simplement modifié l'application pour qu'elle regroupe ses threads par 4 (par affinités par rapport aux communications des résultats intermédiaires) dans des bulles. L'utilisation d'un ordonnancement à éclatement détaillé dans la section 3.4.1 permet alors d'obtenir des performances très proches de la version « fixée » (voir figure 7).

L'utilisation de bulles permet donc d'atteindre des performances comparables à celles obtenues en réglant « à la main » les placements de threads.

L'application CONDUCTION est pour l'instant un cas simple où la charge de calcul entre les bandes est équilibrée, l'utilisation de bulles lui permet simplement de s'adapter automatiquement à l'architecture de la machine. Cette application devrait cependant prochainement être modifiée afin de bénéficier des méthodes de raffinement adaptatif (*AMR*) permettant d'augmenter la précision des calculs aux endroits intéressants. Ceci provoquera de gros déséquilibres en charge de calcul dans le maillage *au cours du calcul* et *selon les résultats obtenus*. Il sera intéressant de comparer les temps

de développement et d'exécution des versions ordonnancées « à la main » d'une part, de façon opportuniste d'autre part, et enfin dynamiquement à l'aide de bulles.

5.2. *Factorisation LU*

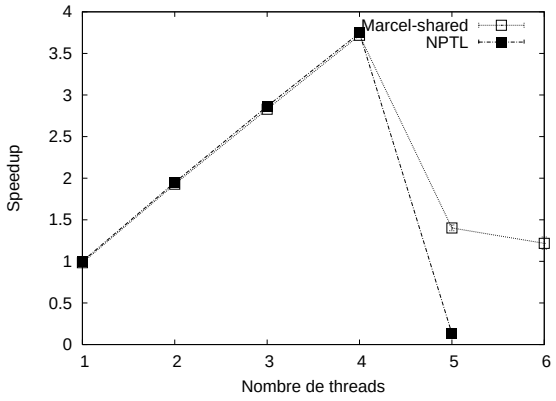
Nous avons testé quelques ordonnanceurs de bulles avec SuperLU_MT Version 1.0 (Demmel *et al.*, 1999) : c'est un solveur parallèle de grands systèmes d'équations linéaires creux et non symétriques (factorisation LU) qui s'appuie sur des BLAS pour profiter au mieux des capacités de calcul et de cache de la machine cible. Les affinités de cache sont ainsi prépondérantes pour obtenir une bonne efficacité. La machine cible est une machine avec deux processeurs dual-core Opteron (donc 4 voies). Le facteur NUMA entre ces deux processeurs est mesuré à peu près 1.4. La figure 8a montre combien SuperLU parvient à passer à l'échelle sur la machine cible. La courbe de *speedup* est presque parfaite tant qu'il n'y a pas plus de threads que de processeurs[5]. Au-delà les performances s'écroulent lorsque l'on utilise un ordonnanceur générique tel que celui de la NPTL (LINUX 2.6.17) ou la bibliothèque Marcel originelle avec une politique d'ordonnancement *shared* (une seule liste de threads prêts est partagée par tous les processeurs), car ceux-ci essaient de respecter l'équité entre threads, et pour cela les font tourner brièvement tour à tour sur les différents processeurs, ce qui empêche de conserver les affinités de cache des BLAS utilisés.

Considérons maintenant une situation où une grande application scientifique multiéchelle a besoin d'effectuer des tâches de factorisations LU d'une manière irrégulière. Il y a plusieurs manières de traiter ces tâches, selon le nombre de threads à lancer par tâche, et comment les ordonnancer. Utiliser un simple ordonnanceur par lot (lançant chaque tâche à l'aide de 4 threads sur les 4 voies jusqu'à sa terminaison), ou à l'inverse utiliser un ordonnanceur complètement distribué (lançant chaque tâche avec un seul thread sur une seule voie jusqu'à sa terminaison) ne sont pas nécessairement le plus adapté, lorsque d'autres parties parallèles de l'application (tournant sur d'autres machines) ont rapidement besoin des résultats de certaines tâches, par exemple.

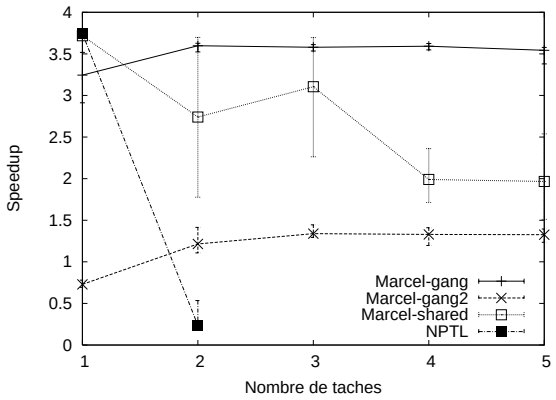
Les figures 8c et 8b montrent les résultats obtenus en utilisant plusieurs approches. Sur la figure 8b, toutes les tâches ont été lancées avec 4 threads, tandis que sur la figure 8c, elles ont toutes été lancées avec 2 threads. Les deux figures montrent clairement que les ordonnanceurs génériques de NPTL et de bibliothèque Marcel originelle obtiennent des performances mauvaises et même erratiques (les barres min-max sont très hautes), car ils considèrent de manière équitable tous les threads de toutes les tâches sans prendre en compte des notions d'affinités.

Nous avons donc fait en sorte que les threads d'une même tâche soient rassemblés au sein d'une même bulle (voir figure 9) et utilisé notre *gang-scheduler* (voir section 3.4.2) pour ordonnancer ces tâches avec des tranches de temps de $0,2ms$ (le temps typique de résolution d'une tâche avec un seul thread est de l'ordre de $10s$).

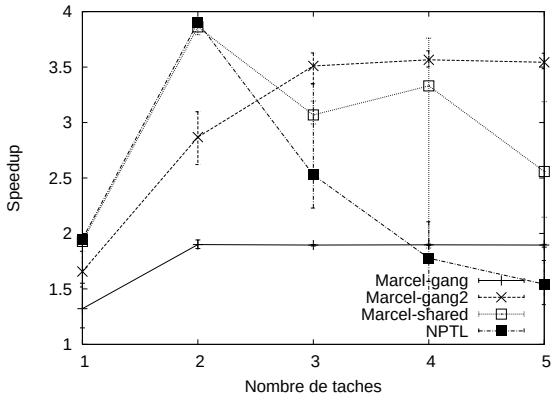
5. Sur une machine à 8 processeurs dual-core (donc 16 voies) équivalente, le *speedup* obtenu n'atteint cependant qu'environ $7,5$.



a. Speedup



b. Tâches 4 threads



c. Tâches 2 threads

Figure 8. *Résultats*

```
void tache_lu(int voies, ...) {
  int i ;
  marcel_t threads[voies] ;
  struct args[voies] ;
  marcel_bubble_t b ;
  marcel_bubble_init(&b) ;
  for (i=0 ; i<voies ; i++) {
    args[i] = ...
    marcel_create(&threads[i], NULL, sub_lu, &args[i]) ;
    marcel_bubble_insertthread(&b, &threads[i]) ;
  }
  marcel_bubble_join(&b) ;
}
```

Figure 9. *Création de bulles au sein des tâches LU*

Nous avons essayé d'utiliser un seul *gang-scheduler* pour l'ensemble de la machine 4 voies (courbes « Marcel-gang »), mais aussi d'utiliser deux *gang-schedulers*, un par nœud NUMA 2 voies (courbes « Marcel-gang2 ».

– La courbe « Marcel-gang » en bas de la figure 8c montre que lancer un seul *gang-scheduler* 4 voies pour des tâches à 2 threads limite évidemment le *speedup* à 2.

– La courbe « Marcel-gang2 » en bas de la figure 8b montre que lancer des tâches à 4 threads sur des nœuds NUMA 2 voies produit également un *speedup* limité (car il y a plus de threads que de processeurs, perdant ainsi les affinités de cache et de synchronisation).

– La courbe « Marcel-gang » en haut de la figure 8b montre que notre *gang-scheduler* arrive assez bien à ordonnancer les tâches sur la machine : chaque tâche obtient un *speedup* d'à peu près $3,5$, quel que soit leur nombre.

– La courbe « Marcel-gang2 » de la figure 8c montre qu'utiliser deux *gang-schedulers* à deux voies permet d'obtenir un résultat similaire, pourvu qu'il y ait suffisamment de tâches bien sûr.

Au final, lorsque l'on compare de plus près ces deux dernières courbes, on peut constater que pour cette application et sur cette machine, lancer un seul *gang-scheduler* pour la machine entière permet en fait d'obtenir des performances légèrement meilleures qu'en lançant un *gang-scheduler* sur chaque nœud NUMA. Cela est sans doute dû au fait que pour obtenir une meilleure répartition, nous avons choisi de laisser les *gang-scheduler* partager l'ensemble des tâches à traiter. Si l'application avait été légèrement différente (moins exigeante au niveau des caches, mais plus exigeante en termes d'accès mémoire), nous aurions observé l'inverse.

Il est à noter que cette expérience a été réalisée sans modifier l'application. Nous avons simplement fait en sorte que Marcel construise des bulles selon la filiation naturelle de création de threads : puisque les threads qui travaillent sur une même tâche sont créés par un même thread, la hiérarchie de bulles résultante correspond bien aux

tâches. Il ne restait alors plus qu'à démarrer les *gang-schedulers* avec les paramètres appropriés.

6. Travaux apparentés

Bossa (Barreto *et al.*, 2002) fournit des abstractions d'ordonnancement de haut niveau et un langage pour développer et prouver des ordonnanceurs. L'implémentation actuelle est basée sur le noyau LINUX. Cependant, l'objectif ici est surtout de pouvoir *prouver* la correction d'un ordonnanceur. Par conséquent, le langage proposé, bien que suffisamment puissant pour implémenter l'ordonnanceur monoprocesseur de LINUX 2.4, est assez restrictif, et limite ainsi beaucoup les programmeurs.

Le projet ELITE (Steckermeier *et al.*, 1995) fournit une très bonne implémentation d'une plate-forme d'ordonnancement de niveau utilisateur qui prend en compte les affinités entre threads, cache et données. Des modèles mathématiques sont même utilisés pour calculer les probabilités des fautes de cache. Cependant, l'interaction avec l'application est très faible : les affinités sont détectées plutôt que fournies par les programmeurs.

Les Process Aggregates (sgi, n.d.) de SGI sont des conteneurs à processus Unix. Les sessions et groupes de processus peuvent être implémentés en tant que Process Aggregates par exemple. Cependant leurs utilisations actuelles sont uniquement pour la sécurité et la gestion des quotas.

Plusieurs systèmes d'exploitation fournissent des outils pour distribuer des threads sur la machine en les rassemblant dans des ensembles : liblgroup sur Solaris (Sun, n.d.) et NSG sur Tru64 (Com, n.d.). De tels ensembles ressemblent beaucoup à des bulles sans notion de récursion. Par nature même, aucun de ces outils n'est vraiment portable, mais surtout, aucun d'entre eux ne fournit le degré de contrôle que nous fournissons : avec la plate-forme BubbleSched, l'application peut fournir des fonctions qui s'insèrent au cœur même de l'ordonnanceur, pour pouvoir réagir à des évènements tels que le *réveil* de thread ou l'*inactivité* des processeurs. Un tel degré d'interaction avec l'ordonnanceur noyau est impossible.

7. Conclusion

Dans ce papier, nous avons proprosé un modèle de programmation qui permet à l'application de structurer son parallélisme afin d'influer sur l'ordonnancement de ses threads. Ce modèle repose sur les concepts de bulle (ensembles imbriqués de threads permettant de qualifier le parallélisme de l'application), sur des listes de tâches pour modéliser l'architecture de la machine visée et sur des primitives portables d'ordonnancement de haut niveau. Au final, des programmeurs spécialistes de l'ordonnancement peuvent « piloter » finement l'ordonnanceur en écrivant quelques fonctions s'insérant au sein de celui-ci, et les programmeurs d'applications peuvent ensuite facilement essayer et paramétrer les ordonnanceurs produits.

Une implémentation au sein de l'ordonnanceur MARCEL, nous a ensuite permis de valider cette proposition sur de nombreux architectures et systèmes. Des exemples d'implémentation de stratégies d'ordonnancement ont montré combien cela était à la fois facile et puissant. Bien sûr, écrire de telles fonctions reste difficile du point de vue algorithmique, mais avec l'aide du déboggueur, les programmeurs peuvent vraiment laisser les détails techniques de côté pour se concentrer sur ces problèmes algorithmiques. Nous pensons aussi que notre plate-forme est un bon outils pour valoriser le savoir-faire d'une communauté : une gamme d'ordonnanceurs adaptés à un domaine spécifique peut être réalisée par un groupe de spécialistes et être ainsi utilisée par un plus grand nombre de programmeurs.

Ces travaux ouvrent de nombreuses perspectives. A court terme, un mécanisme générique pour les attributs et statistiques sur les bulles sera développé comme aide à la décision : les programmeurs d'application pourraient même fournir leurs propres outils de mesure, que la plate-forme BubbleSched synthétiserait selon la hiérarchie de bulles. Plusieurs approches algorithmiques pourront alors être implémentées, testées et affinées pour ordonnancer de véritables applications.

Comme nous l'avons déjà évoqué et évalué en section 5.2, une perspective naturelle est l'automatisation de la production de bulles. Nous pensons que notre approche devrait bien s'adapter aux programmes OPENMP et améliorer l'exécution des sections parallèles imbriquées.

A plus long terme, un ordonnanceur générique pourrait être développé ; celui-ci prendrait en compte autant d'informations qu'il est possible de collecter auprès du matériel, du compilateur et du programmeur. L'intégration au sein du noyau LINUX pourrait même être envisagée, puisqu'une hiérarchie naturelle de bulles existe déjà au travers des notions de threads, processus, jobs, sessions et utilisateurs.

8. Logiciel

Marcel est disponible en téléchargement au sein de la suite PM2 sur
`http://runtime.futurs.inria.fr/Runtime/logiciels.html`
selon les termes de la licence GPL.

Remerciements

Ces travaux ont été financés par l'Agence nationale pour la recherche (ANR) au travers des projets PARA (ANR-05-CIGC-001) et NUMASIS (ANR-05-CIGC-002).

9. Bibliographie

Barreto L. P., Muller G., « Bossa : une approche langage à la conception d'ordonnanceurs de processus », *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 14)*, Hammamet, Tunisie, April, 2002.

Carlson W., Draper J., Culler D., Yelick K., Brooks E., Warren K., Introduction to UPC and Language Specification, Technical Report n° CCS-TR-99-157, George Mason University, May, 1999.

Com, *NUMA Scheduling Groups (NSG)*. n.d., `http://modman.unixdev.net/?sektion=4&page=numa_scheduling_groups&manpath=OSF1-V5.1-alpha`.

Dandamudi S., Cheng S.*Systems Architecture*, , « Performance impact of run queue organization and synchronization on large-scale NUMA multiprocessor systems », 1997, vol. 43, p. 491-511.

Danjean V., Contribution à l'élaboration d'ordonnanceurs de processus légers performants et portables pour architectures multiprocesseurs, PhD thesis, Ecole Normale Supérieure de Lyon, December, 2004.

Danjean V., Namyst R., Russell R., « Integrating Kernel Activations in a Multithreaded Runtime System on Linux », *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, vol. 1800 of *Lect. Notes in Comp. Science*, En conjonction avec IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag, Cancun, Mexico, May, 2000, p. 1160-1167. `http://www.ens-lyon.fr/~bouge/Biblio/Danjean/DanNamRus00RTSPP.ps.gz`.

Demmel J. W., Gilbert J. R., Li X. S.*SIAM J. Matrix Analysis and Applications*, , « An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination », 1999, vol. 20, n° 4, p. 915-952.

Edler J., Practical Structures for Parallel Operating Systems, PhD thesis, New York University, May, 1995.

Feitelson D. G., Rudolph L.*Parallel and Distributed Computing*, , « Evaluation of Design Choices for Gang Scheduling Using Distributed Hierarchical Control », 1996, vol. 35, p. 18-34.

Fukuda A., Fukiji R., Kai H., « Two-level Processor Scheduling for Multiprogrammed NUMA Multiprocessors », *Computer Software and Applications Conferences*, IEEE, 1993, p. 343-351.

Hénon P., Ramet P., Roman J., « PaStiX : A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions », *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, Springer-Verlag, January, 2000, p. 519-527.

Li H., Tandri S., Stumm M., Sevcik K. C., « Locality and Loop Scheduling on NUMA Multiprocessors », *International Conference on Parallel Processing*, vol. 2, August, 1993, p. 140-127.

Lin, *Linux Scalability Effort*. n.d., `http://lse.sourceforge.net/`.

Lovett T. D., Clapp R. M., Safranek R. J., NUMA-Q : An SCI-Based Enterprise Server, Technical Report, Sequent Computer Systems Inc., 1996.

Marathe J., Mueller F., « Hardware Profile-guided Automatic Page Placement for ccNUMA Systems », *Sixth Symposium on Principles and Practice of Parallel Programming*, March, 2006.

Markatos E., Leblanc T.*Parallel and Distributed Systems*, April, , « Using processor affinity in loop scheduling on shared-memory multiprocessors », 1994, vol. 5, n° 4, p. 379-400.

Mattson T., EigenMann R., « OpenMP : An API for Writing Portable SMP Application Software », *SuperComputing 99 Conference*, November, 1999.

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

Oguma H., Nakayama Y., « A Scheduling Mechanism for Lock-free Operation of a Lightweight Process Library for SMP Computers », *Conference on Parallel and Distributed Systems*, IEEE, July, 2001, p. 235-242.

Ousterhout J. K., « Scheduling techniques for concurrent systems », *Third International Conference on Distributed Computing Systems*, October, 1982, p. 22-30.

Pérache M., « Nouveaux mécanismes au sein des ordonnanceurs de threads pour une implantation efficace des opérations collectives sur machines multiprocesseurs », *Rencontres francophones en Parallélisme, Architecture, Système et Composant (RenPar 16)*, Le Croisic, France, March, 2005.

Polychronopoulos C., Kuck D.*Transactions on Computers*, December, , « Guided Self-Scheduling : A Practical Scheduling Scheme for Parallel Supercomputers », 1987, vol. 36, n° 12, p. 1425-1439.

Radović Z., Hagersten E., « Efficient Synchronization for Nonuniform Communication Architectures », *SC02*, IEEE, Baltimore, Maryland, USA, October, 2002.

Roberson J., ULE : A Modern Scheduler For FreeBSD, Technical Report, The FreeBSD Project, `jeff@FreeBSD.org`, 2003.

Russinovich M.*Windows IT Pro*, July, , « Inside the Windows NT Scheduler, Part 2 », 1997. `http://www.windowsitpro.com/Article/ArticleID/303/303.html`.

Schreiber R., « An Introduction to HPF », *The Data Parallel Programming Model : Foundations, HPF Realization, and Scientific Applications*, Springer-Verlag, 1996, p. 27-44.

sgi, *Process Aggregates*. n.d., `http://oss.sgi.com/projects/pagg/`.

Shen X., Gao Y., Ding C., Archambault R., « Lightweight Reference Affinity Analysis », *19th ACM International Conference on Supercomputing*, Cambridge, MA, USA, June, 2005, p. 131-140.

Solutions D. I., *The Dolphin SCI Interconnect*. February, 1996, `http://www.dolphinics.com/pdf/whitepaper/T-WhitePaper.pdf`.

Steckermeier M., Bellosa F., Using Locality Information in Userlevel Scheduling, Technical Report n° TR-95-14, University of Erlangen-Nürnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstraße 1, 91058 Erlangen, Germany, December, 1995. `http://www4.informatik.uni-erlangen.de/Projects/FORTWIHR/ELiTE/`.

Sun, *Solaris Memory Placement Optimization (MPO)*. n.d., `http://iforce.sun.com/protected/solaris10/adoptionkit/tech/mpo/mpo_man.html`.

Tang P., Yew P.-C., « Processor Self-Scheduling for Multiple Nested Parallel Loops », *Proceedings 1986 International Conference on Parallel Processing*, August, 1986, p. 528-535.

Tzen T., Ni L.*Parallel and Distributed Systems*, January, , « Trapezoid Self-Scheduling : A Practical Scheduling Scheme for Parallel Compilers », 1993, vol. 4, n° 1, p. 87-98.

Wang Y.-M., Wang H.-H., Chang R.-C.*Systems Software*, , « Clustered Affinity Scheduling on Large-Scale NUMA Multiprocessors », 1997, vol. 39, p. 61-70.

Wang Y.-M., Wang H.-H., Chang R.-C.*Systems and Software*, , « Hierarchical loop scheduling for clustered NUMA machines », 2000, vol. 55, p. 33-44.

Whitney S., McCalpin J., Bitar N., Richardson J. L., Stevens L., « The SGI Origin Software Environment and Application Performance », *COMPCON 97*, IEEE, San Jose, California, 1997, p. 165-170.

Article reçu le 23 janvier 2007
Accepté après révisions le 25 octobre 2007

Samuel Thibault *a terminé un doctorat en informatique à l'université de Bordeaux 1. Ses travaux portent sur l'ordonnancement de* threads *sur machines multiprocesseurs hiérarchiques, en particulier dans le cadre du calcul hautes performances. Il est actuellement en post-doctorat chez XenSource.*

Raymond Namyst *est Professeur en informatique à l'université de Bordeaux 1. Il dirige l'équipe projet INRIA Runtime, qui s'intéresse à l'étude et la conception de supports d'exécutions performants pour architectures parallèles.*

Pierre-André Wacrenier *est Maître de Conférences en informatique à l'université de Bordeaux 1. Ses travaux portent sur le développement des supports OpenMP pour un ordonnancement efficace sur machines multiprocesseurs hiérarchiques.*

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   Technique et Science Informatique RENPAR

2. AUTEURS :
   Samuel Thibault[*], *Raymond Namyst*[*], *Pierre-André Wacrenier*[*]

3. TITRE DE L'ARTICLE :
   BubbleSched, plate-forme de conception d'ordonnanceurs de threads sur machines hiérarchiques

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   BubbleSched

5. DATE DE CETTE VERSION :
   6 août 2008

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   [*] LaBRI - Université Bordeaux 1
   351 cours de la Libération — F-33405 Talence cedex
   {samuel.thibault,raymond.namyst,pierre-andre.wacrenier}@labri.fr
   – téléphone : 05 40 00 35 40
   – télécopie : 05 40 00 66 69
   – e-mail : samuel.thibault@labri.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes2.cls`,
   version 1.23 du 17/11/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél. : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com