Ordonnancements de threads dirigé par la mémoire sur architecture NUMA.

Sylvain Jeuland

Ordonnancement de threads dirigé par la mémoire sur architecture NUMA

Sylvain JEULAND
15 septembre 2007

Encadrants : Raymond Namyst et Pierre-André Wacrenier
Tuteur ENSEIRB : Julien Allali

Table des matières

1 Programmation des machines multiprocesseurs contemporaines **7**
1.1 Des architectures parallèles fortement hiérarchisées 7
1.1.1 Architectures multiprocesseurs 7
1.1.2 Importance du facteur NUMA 9
1.2 Ordonnancement et placement mémoire sur machines hiérarchisées 10
1.2.1 Ordonnancement de threads 10
1.2.2 Gestion de la mémoire 14
1.3 Discussion 17

2 MemAware : Ordonnancement des bulles lié à l'affinité mémoire **19**
2.1 Gestion des tas mémoire sur architecture NUMA 19
2.2 Un ordonnanceur pour la répartition des bulles 20
2.2.1 Articulation entre mSteal et mSpread 20
2.2.2 L'algorithme mSpread 21
2.2.3 L'algorithme mSteal 22

3 Mise en œuvre et évaluation **25**
3.1 Interface Heap Allocator 25
3.1.1 La structure pinfo 25
3.1.2 Création de tas et allocations mémoire 26
3.1.3 Entités, migration mémoire et fusion de tas 26
3.2 Implémentation des fonctionnalités de gestion de la mémoire 26
3.2.1 Migration mémoire 27
3.2.2 Bassins d'attraction principal 27
3.2.3 Epaisseur de bulle 28
3.3 Implémentation de l'ordonnanceur MemAware 29
3.3.1 Notion d'équilibre dans MemAware et choix de l'ordonnanceur 29
3.3.2 Détails d'implémentation du vol de travail par mSteal 30
3.3.3 Détails d'implémentation de la répartition par mSpread 31
3.4 Evaluation 33
3.4.1 Démonstrastion de MemAware sur une application synthétique 33
Références 40

Introduction

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 multicoeurs envahissent le marché et leur puissance de calcul se compte désormais en nombre de coeurs 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 coeurs. 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.

Par exemple, les architectures SMP (Symmetric MultiProcessing) des machines multiprocesseurs les plus répandues, dont les processeurs accèdent à une mémoire commune par un bus mémoire, peuvent maintenant être composées de processeurs multi-cœurs eux mêmes hiérarchisés, comme par le partage de certains niveaux de cache. La généralisation de la parallélisation interne conduit toutefois à la nécessité de distribuer la mémoire lorsque le nombre de processeurs devient élevé, pour éviter le goulet d'étranglement engendré par les multiples accès concurrents aux anciens bus centralisés. Des machines de types NUMA (Non Uniform Memory Access) distribuent ainsi leur mémoire au prix d'iniquités des temps d'accès aux différentes zones mémoire. Il est désormais connu que le temps requis pour accéder aux données varie graduellement avec la distance physique à la mémoire et que ces contraintes doivent être prises en compte par les applications. Aussi, pour faciliter l'exploitation des machines NUMA *SGI Origin2000*, un ensemble de travaux autour du système *Cellular Irix* ont été menés tant au niveau noyau qu'au niveau utilisateur [10, 3, 6]. Cependant, il ne faut pas qu'au cours du processus d'optimisation le programmeur spécialise trop son code par rapport à une machine donnée, sacrifiant, souvent à son insu, la portabilité de l'application aux performances.

Ce mémoire présente la démarche et les expériences que nous avons entreprises pour

jeter les bases d'un environnement d'exécution parallèle afin d'améliorer de façon portable l'exécution sur machine NUMA des programmes accèdant intensivement aux ressources mémoire. Après avoir rappelé les clefs de l'architecture et de la programmation parallèle et les différentes techniques de gestion mémoire existantes, nous détaillons notre démarche pour coordonner, lors de l'ordonnancement, le placement des tâches et de leurs données afin d'obtenir les performances optimales. Nous présentons une implémentation de notre support exécutif basée sur la bibliothèque de threads *MARCEL* et évaluons le comportement et les performances obtenus.

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

Depuis que les constructeurs de processeurs ne peuvent plus compter sur l'augmentation régulière de la fréquence pour améliorer les performances de leurs produits, de nouvelles architectures ont intégré du parallélisme depuis l'agencement même de la machine, pour gagner en puissance de calcul.

1.1.1 Architectures multiprocesseurs

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 [7] sur le sujet.

Architecture 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 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, plusieurs processeurs identiques se connectent 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.

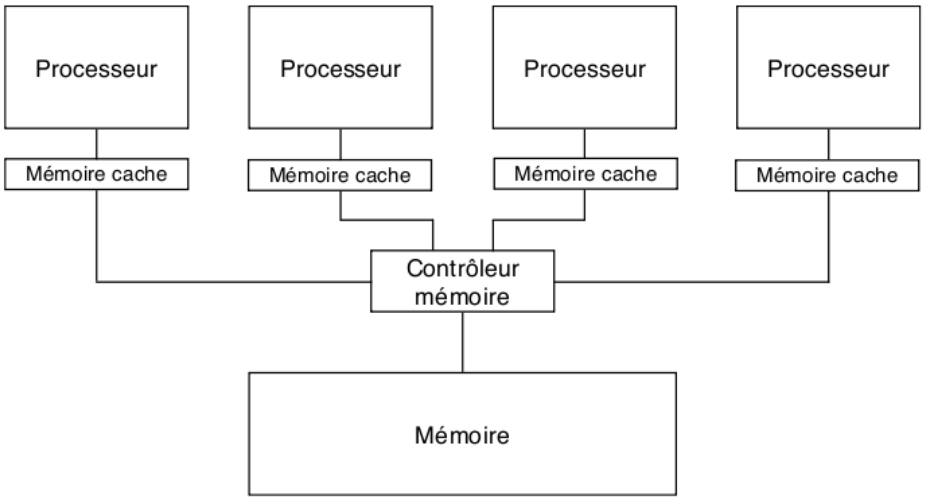


FIG. 1.1 – Exemple d'architecture SMP.

Peu onéreuse, cette architecture perd en efficacité lorsque le nombre des processeurs augmente. 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 a 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.

Architecture 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 machines parallèles NUMA (Non Uniform Memory Access) sont des machines multiprocesseurs à mémoire partagée, dans lesquelles la mémoire est morcelée en plusieurs bancs. Chaque processeur accède à une zone mémoire locale rapidement, et à des zones mémoire distantes avec un temps d'accès plus long selon la proximité géographique des processeurs. Concrètement, une machine NUMA est composée d'une interconnexion de blocs basés sur l'architecture SMP, reliés entre eux par un réseau rapide ou un commutateur, comme illustré sur la figure 1.2. Chaque bloc est constitué d'un nombre limité de processeurs et de leurs mémoires cache, connectés à un banc mémoire local. Le programmeur perçoit ces différents bancs de mémoire sous la forme d'une unique mémoire globale.

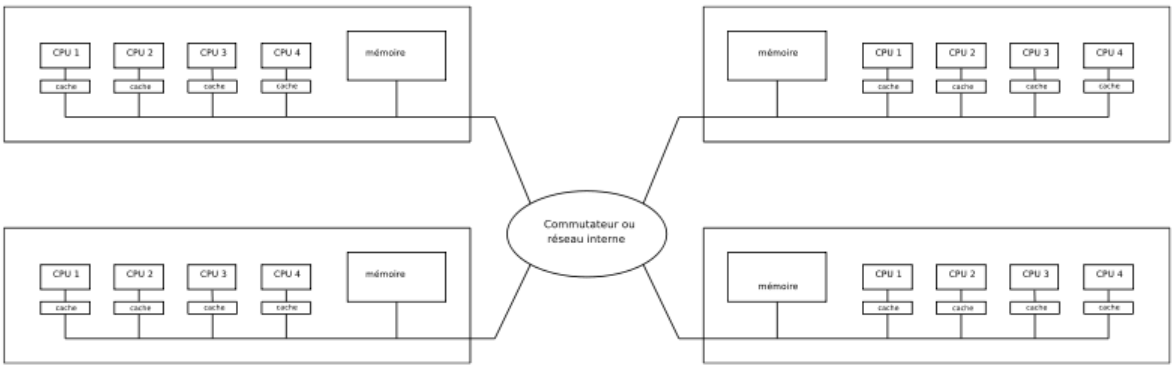


FIG. 1.2 – Machine NUMA composée de quatre nœuds comportant chacun quatre processeurs et un banc mémoire

Comme les performances du réseau interconnectant les blocs sont limitées, les variations pour l'accès aux différents bancs mémoire conduisent à la notion de facteur NUMA. Un facteur NUMA correspond au temps d'un accès à la mémoire locale, divisé par le temps d'accès à la mémoire distante. Ce facteur traduit la non uniformité d'accès à la mémoire, et contribue à l'évaluation de la machine en terme de performances. Plus celui-ci se rapproche de un, plus le coût supplémentaire, engendré par un accès distant par rapport à un accès local, est faible. On s'approche ici du modèle d'une machine accédant uniformément à une mémoire globale. Pour des machines NUMA dont la topologie est complexe, ou hiérarchique à plusieurs niveaux, plusieurs facteurs NUMA caractérisent la machine, compris aujourd'hui entre 110 et 300% selon les architectures.

1.1.2 Importance du facteur NUMA

Le facteur NUMA joue un rôle important dans les programmes qui utilisent la mémoire de manière intensive. En effet, lorsque les accès mémoire sont nombreux et répétés, il est préférable que les données soient accédées localement de manière à minimiser le temps d'accès. C'est ici qu'interviennent les notions de placement et de localité mémoire. En effet, optimiser le placement mémoire de manière à ce que celle-ci soit locale au processus léger (thread) qui l'utilise éviterait beaucoup d'accès distants. De plus, il se pose ici le problème de la mémoire partagée. Si plusieurs threads partagent les même données mémoire et que ceux-ci sont placés sur différents nœuds de la machine, certains devront recourir à des accès distants qui ralentiront le programme. Pour remédier à ce problème, il faudrait par exemple regrouper les threads partageant les mêmes données sur un même nœud comme sur la figure 1.3 à droite.

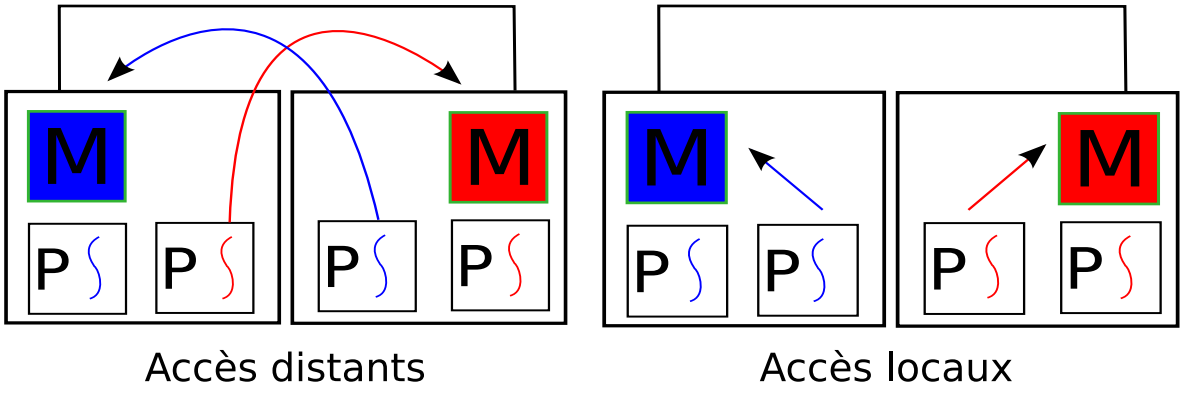


FIG. 1.3 – Répartition des threads et de leur mémoire sur la machine

Comme illustré sur la figure 1.3, il est préférable d'exécuter les threads où est allouée la mémoire. Néanmoins il peut se poser un problème de répartition. En effet, considérons une architecture avec deux nœuds A et B qui contiennent chacun deux processeurs. Soient trois threads t1, t2 et t3 qui accèdent à de la mémoire partagée sur le nœud A et un quatrième thread t4 qui accède à de la mémoire placée sur le nœud B. Imaginons que l'on place t1 et t2 sur les deux processeurs du nœud A et t4 sur un processeur du nœud B, il reste un processeur de libre sur le nœud B. Nous avons trois possibilités :

- placer t3 sur le processeur libre et à distance de sa mémoire,
- placer t3 sur le nœud A même s'il n'y a pas de processeur de libre,
- déplacer la mémoire de t3 du nœud A vers le nœud B.

Parfois, il arrive qu'il n'y ait plus de place sur un nœud A pour allouer de la mémoire et qu'on soit obligé d'allouer à distance sur un nœud B. On peut soit se contenter de cette situation, soit migrer le thread afin qu'il soit sur le nœud A. C'est ce genre de choix auquel bon nombre d'ordonnanceurs et d'allocateurs mémoire sont confrontés. Il est donc nécessaire de trouver un ordonnanceur qui travaille de concert avec un allocateur afin de gérer finement la répartition du travail et le placement mémoire de manière à optimiser l'exécution des programmes.

1.2 Ordonnancement et placement mémoire sur machines hiérarchisées

Plusieurs techniques sont mises en œuvre afin de favoriser la localité mémoire pour les programmes parallèles. Ces techniques reposent en général sur des appels systèmes afin de contrôler le placement des threads sur les nœuds (via l'ordonnanceur) ou définir des réservoirs de mémoire physique sur les nœuds (bibliothèques d'allocation spécialisées).

1.2.1 Ordonnancement de threads

La généralisation des architectures multicore a dopé la recherche en matière d'ordonnancement. Parmi les nombreuses études menées ces dernières années on peut citer l'algorithme

original d'ordonnancement, intitulé *cache fair algorithm* [1], qui s'assure du partage équitable des caches par différents threads et applications. Il a pour premier objectif de réduire les effets d'un partage de cache inégal entre threads. Cela évite que les priorités entre threads ne s'inversent. Il est difficile d'estimer le comportement global des différentes applications car il existe une grande variabilité des performances en fonction de leur cohabitation dans le cache (co-runners). L'algorithme utilise des fonctionalités matérielles afin de déterminer comment les performances d'un thread sont affectées par un partage de cache non équitable. Une fois les mesures réalisées, des modèles analytiques et statistiques permettent de calculer en temps réel combien de temps on doit donner aux threads affectés, tout en enlevant des quantums de temps aux autres threads.

Présentation de l'ordonnanceur de threads MARCEL et de la plateforme de développement BUBBLESCHED

MARCEL est une bibliothèque de threads utilisateurs faisant partie de la suite logicielle *PM2*, 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. A l'origine, cette bibliothèque a été pensée pour faciliter la migration de threads sur une machine, ou a travers un réseau, permettant ainsi la mise en place de solution d'équilibrage de charge. Aujourd'hui, cette bibliothèque fournit bien plus que cette migration. Elle introduit le concept de *bulle*, 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 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. Cette bibliothèque met aussi 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.

Pour accélérer le déroulement d'une application sur une machine hiérarchique, la bibliothèque *MARCEL* [8], outre son interface *POSIX*, dispose d'un ensemble de fonctionnalités supplémentaires, dénommée *BUBBLESCHED* [8], permettant de modéliser la machine, et de structurer l'ensemble des threads de l'application. L'ordonnanceur Affinity utilise la bibliothèque de threads Marcel.

Modélisation des machines de calcul

Comme 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. 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 réequilibre la charge entre les différentes listes. En revanche, rien n'assure que ce réequilibrage 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

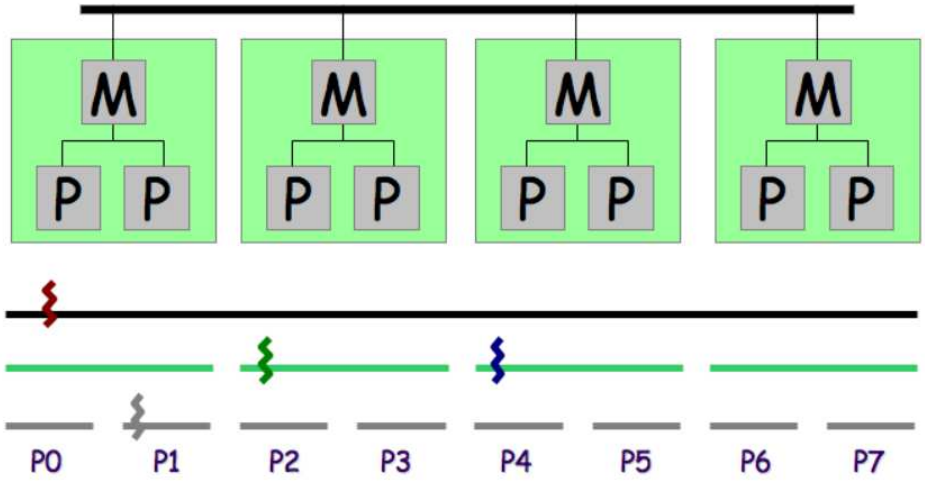


FIG. 1.4 – Modélisation de l'architecture d'une machine hiérarchique à l'aide d'un arbre de listes.

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 apparaitre 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* [2], 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 1.4.

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.

Encapsulation de threads dans les bulles

L'utilisateur donne des informations sur le parallélisme de son programme en regroupant des processus qui partagent de la mémoire dans des structures de bulle. Ces structures sont récursives, c'est-à-dire qu'une bulle peut contenir d'autres bulles, comme le montre la figure 1.5.

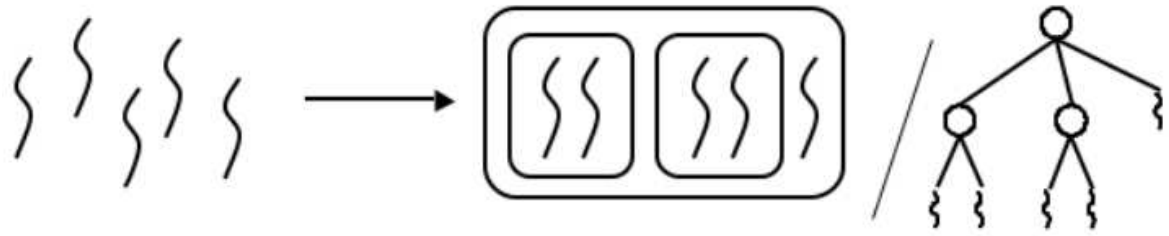


FIG. 1.5 – Encapsulation de processus légers dans des bulles

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 estimations de l'utilisation processeur ou mémoire.

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

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.

A 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* permettant de répartir facilement bulles et threads sur les listes, afin d'obtenir des distributions telles que présentées en 1.2.1. Un ordonnanceur 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 est inactif. Par exemple, nous pouvons citer comme exemple d'ordonnancement, l'algorithme Affinity présenté ci-dessous.

Un algorithme d'ordonnancement : Affinity

L'ordonnanceur *Affinity* [5], présenté par François Broquedis, utilise l'ordonnanceur MARCEL et la plateforme *BUBBLESCHED*. Il permet d'optimiser le déroulement des programmes OpenMP. Son principe est simple, il réunit les équipes de threads partageant une section parallèle dans une *bulle*. Ensuite, il s'agit de répartir les bulles sur une machine NUMA en respectant les relations d'affinités exprimées entre les tâches, par exemple le partage de ressources ou des synchronisations régulières.

Afin de conserver l'affinité, il ne faut séparer les threads travaillant sur les même données qu'en cas de force majeure. L'algorithme doit retarder au maximum l'éclatement des bulles. Le fonctionnement général de l'algorithme Affinity consiste à répartir un ensemble de bulles sur l'ensemble de la machine en en éclatant le moins possible. L'algorithme Affinity peut se décrire en plusieurs étapes. Au début, on compare le nombre d'entités (nombre de bulles et de threads) par rapport au nombre de niveaux topologiques sur lesquels on répartit. S'il y a assez d'entités pour alimenter tous les processeurs, on les répand sur la machine sans éclater de bulles. Dans le cas contraire, on est contraint d'éclater une bulle (la plus grosse de préférence) en séparant les threads se partageant les données. Lorsqu'un thread se termine et qu'un processeur devient inactif, on va chercher du travail à voler le plus localement possible. Une fois qu'on a trouvé assez d'entités afin de redonner du travail à notre processeur inactif, on relance l'algorithme *Affinity* sur une partie de la machine qui va de nouveau répartir le travail.

Tout au long de l'exécution du programme, l'algorithme *Affinity* sera appelé tant qu'il y a plus de threads que de processeurs de manière à toujours pouvoir bien répartir les entités. Lorsqu'une bulle s'exécute sur un processeur, les threads contenus à l'intérieur se partagent de manière optimisée le cache car il y a des chances qu'ils travaillent sur les mêmes données puisqu'ils viennent de la même section parallèle d'OpenMP.

1.2.2 Gestion de la mémoire

Dans l'existant, plusieurs bibliothèques et allocateurs ont été conçus afin de pouvoir gérer finement la mémoire dans des contextes multitâches, afin d'optimiser les temps d'accès aux données.

Bibliothèque standard d'allocation

Tout d'abord, rappelons que la bibliothèque *C* standard permet d'allouer de la mémoire en contexte multithreadé. Il est possible de l'utiliser mais elle n'obtient pas de bonnes performances. En effet, deux problèmes apparaissent.

- Le faux-partage se manifeste lorsque deux variables x et y sont malencontreusement allouées sur la même ligne de cache et que celles-ci sont utilisées par deux threads t1 et t2. Lorsque que le thread t1 fait la mise à jour de la variable x dans son cache, cela invalidera la ligne de cache de t2 qui n'utilise que y et la rendra inutilisable. t2 devra recharger la ligne à partir de la mémoire, d'où une perte de temps. Le phénomène s'amplifie si les modifications de x ou y sont fréquentes.
- Si deux threads accèdent à la même mémoire partagée, il faut des mécanismes de synchronisation efficaces afin qu'ils n'accèdent pas en même temps aux variables et ceci sans trop ralentir le programme.

Afin de résoudre ces problèmes, des bibliothèques spécifiques on été implémentées. Nous présentons ci-dessous les caractéristiques des bibliothèques les plus communes ainsi que les appels systèmes utiles au placement physique des données.

Allocateur Streamflow

Nombreux sont les allocateurs qui permettent de gérer la mémoire. On peut citer par exemple *Hoard* [4] qui alloue un tas par thread afin d'éviter le faux-partage et un tas global pour le partage mémoire. L'allocateur *Vam* [12] quant à lui supprime pour les petits objets, les en-têtes de blocs de pages allouées de manière à ne pas gaspiller trop de mémoire et ne pas polluer les lignes de cache. L'allocateur appelé *Streamflow* [9], qui est un allocateur mémoire haute-performance reprend notamment les caractéristiques explicitées précédemment. Il apporte une synchronisation allégée, une localité temporelle et spatiale des données aux niveaux cache et page, et réduit les conflits de cache ainsi que le faux-partage. Tout d'abord, comme *Hoard*, *Streamflow* alloue un tas par thread (afin d'éviter la contention mémoire et le faux-partage) et supprime les en-têtes. Ceci augmente la localité des données au niveau des caches et des pages. Un autre point intéressant de Streamflow est l'utilisation des désallocations distantes. Un thread peut libérer de la mémoire allouée par un autre thread. Pour ce faire, il devra faire appel à une primitive de synchronisation. On peut aussi noter que lorsqu'un thread meurt, celui-ci ne libère pas sa mémoire. Elle est déclarée comme orpheline et utilisable par les autres threads.

Un avantage certain de *Streamflow* est qu'il utilise peu de primitives de synchronisation, d'où un gain en temps considérable. Seuls quelques situations nécessitent une synchronisation comme par exemple la déclaration qu'un bloc de pages est orphelin ou son adoption par un autre thread. Enfin, on peut introduire la notion de superpages dans lesquelles on peut allouer des blocs de pages contiguës en mémoire physique et optimiser les placements des pages afin d'éviter les conflits de cache. Toutes ces optimisations font de Streamflow un très bon allocateur mémoire aussi scalable que les allocateurs séquentiels. Néanmoins, l'allocateur *Streamflow* ne s'adapte pas aux architectures NUMA, ce qui pose un problème lorsque l'on veut allouer sur un nœud A ou un nœud B.

Libnuma

La bibliothèque *NUMA Policy* est un ensemble d'appels système qui allouent de la mémoire pour les processus et qui gèrent le placement de la mémoire physique sur les différents nœuds NUMA. La *libnuma* quant à elle, offre une interface simple de programmation pour NUMA Policy supportée par le noyau Linux. Il existe plusieurs politiques d'ordonnancement comme l'allocation circulaire sur plusieurs nœuds, l'allocation sur un nœud préféré, sur le nœud local (où se trouve le thread) ou encore sur un nœud forcé. De plus, il est possible de lier des threads à des nœuds spécifiques ou de collecter des statistiques sur le taux de défaut de pages sur chaque nœud avec l'appel de la fonction *numastat*. Néanmoins, il est difficile de réordonnancer les différents threads en fonction du taux de défaut de pages car il n'existe pas d'ordonnanceur qui prend en compte les statistiques de retour.

Migration automatique de la mémoire

Dans les subsubsectiones précédents, les notions d'allocateur mémoire (Streamflow et Libnuma) et d'ordonnancement de threads avec gestion de cache (*cache fair algorith*, *MARCEL*, *BUBBLESCHED* et *Affinity*) ont été abordées. Néanmoins, afin d'améliorer les performances, il est nécessaire de combiner l'ordonnancement des threads avec la gestion mémoire sur architecture NUMA. L'idéal serait que chaque thread puisse disposer de la mémoire à laquelle il accède dans le même nœud que celui-ci afin de favoriser la localité mémoire.

Cette localité peut être favorisée dès le début du programme en faisant en sorte que la mémoire allouée soit placée auprès des threads dès le début du programme. En effet, lors de l'initialisation, le *first touch*, premier accès à la mémoire a tendance à placer toutes les pages sur un nœud particulier et non pas sur les nœuds où seront placés les threads du parallélisme. Une première technique pour éviter ce problème est présentée dans le papier de Jaydeep Marathe et Frank Mueller[6]. Le principe est d'exécuter une première fois une petite partie du programme afin de mesurer puis de connaître les affinités mémoire des threads. On les enregistre puis on relance le programme en plaçant les pages sur les bons nœuds en anticipant le first touch. Cette technique fonctionne avec la mémoire statique et la mémoire dynamique et ne requiert pas l'appui du système d'exploitation. Une seconde technique consiste à migrer les pages lors de la première itération de boucle avec l'appel d'une fonction *next touch*. On suppose que le premier accès mémoire d'un thread est représentatif du reste de l'exécution en espérant qu'il n'y a pas de grand changement dans l'exécution auquel cas il faudra migrer la mémoire de nouveau et en temps réel. Pour ce faire, certains ordonnanceurs font en sorte que les pages mémoires soient migrées en même temps que les threads. Nous pouvons citer *Irix* [10] qui dispose de mécanismes de migration et de placement de pages se basant sur des mesures du noyau. *Irix* ne tient pas compte de l'application afin d'obtenir des informations sur le déroulement du programme, ce qui peut conduire à des erreurs de migration de pages.

Dimitros Nikolopoulos présente un algorithme de migration de pages, le *predictive algorithm* [3], qui tient compte à la fois de l'ordonnanceur du système d'exploitation avec des informations données par l'application parallèle. Celle-ci prévient la migration d'un thread ou sa préemption, ce qui déclenche le processus de migration de pages. Cet algorithme est utilisé dans les programmes parallèles itératifs, ce qui permet au système d'exploitation de comparer le taux de référence mémoire d'un passage de boucle à l'autre et de savoir si celui-ci est plus important qu'avant. Soit une page P initialement allouée sur le nœud H (home).

Si le taux de référence de ce nœud H pour la page P diminue entre une itération 1 et une itération 2, alors que le taux de référence d'un autre nœud N à cette page P augmente entre l'itération 1 et l'itération 2, on peut supposer que le thread qui accède à cette page a été migré. Cette page est donc notée orpheline, et on peut la migrer si les bénéfices sont plus importants que le coût de migration.

1.3 Discussion

Bien qu'il soit techniquement possible de maitriser le placement des threads et celui des pages mémoire, les ordonnanceurs actuels ne sont pas capable d'assurer un placement efficace, faute d'un réel dialogue avec les applications. En effet, les programmes imposent aux systèmes d'allouer de la mémoire sur des nœuds sans tenir compte de leur état et de leur taux d'occupation. Cela peut amener un déséquilibre au niveau de l'encombrement mémoire de la machine. D'un autre côté, le système peut déplacer de la mémoire ou des threads d'un nœud A vers un un nœud B à l'insu de l'application qui avait alloué la mémoire localement aux threads. La mémoire ayant été migrée à distance, les performances sont donc réduites. Il manque donc une gestion mémoire finement couplée à l'ordonnancement et le guidant. L'idéal serait d'implémenter une gestion de la mémoire et un ordonnanceur de threads qui travaillent ensemble afin de gérer finement les aspects comme la charge de travail, le poids mémoire, la localité des données, etc. Il est nécessaire de définir une interface entre ces deux outils afin que l'un puisse utiliser les informations de l'autre, de sorte qu'ils travaillent de manière cohérente.

Chapitre 2

MemAware : Ordonnancement des bulles lié à l'affinité mémoire

L'objectif de ce mémoire est de concevoir et d'implémenter un ordonnanceur qui travaille de manière collaborative avec un allocateur mémoire adapté aux architectures NUMA. Nous cherchons à construire un ordonnanceur qui permet de répartir équitablement la charge de travail sur la machine en tenant compte des affinités mémoires, et ce tout en essayant d'obtenir une situation stable et durable. Notons que la stabilité est un paramètre important puisque tout réordonnancement à un coût, non seulement en terme de migration mémoire mais aussi en terme d'invalidation de cache.

Dans cette perspective, nous avons tout d'abord déterminé les paramètres utiles à un placement fin des tâches et de leur mémoire. Ces paramètres, donnés par le programme utilisateur, expriment la durée de vie d'un thread (afin de ne pas migrer des threads en fin de vie), la quantité mémoire allouée par une tâche et la fréquence d'accès à toute ou partie de celle-ci. De plus, il s'agit aussi de tenir compte du fait que certains threads partagent plus ou moins de mémoire avec d'autres threads. Pour exprimer cette relation de partage, nous avons exploité la notion de bulle intégrée à la bibliothèque *MARCEL* : dans notre cadre, une bulle regroupera des entités qui partagent de la mémoire. De plus, à toute entité est attaché un tas, partagé dans le cas d'une bulle par ses sous-entités et privé dans le cas d'un thread. Disposant de la fréquence d'accès et de la quantité de mémoire allouée, il est alors possible de qualifier la relation de partage attachée à une bulle et d'estimer le coût de sa migration ou de sa répartition sur plusieurs nœuds.

Pour réaliser cet environnement d'exécution, nous avons défini une bibliothèque d'allocation mémoire permettant au programmeur d'indiquer à un ordonnanceur à bulles les paramètres utiles, améliorant ainsi le dialogue entre l'application et l'ordonnanceur. Nous présentons dans cette section cette bibliothèque d'allocation et détaillons la stratégie d'ordonnancement que nous avons mis au point.

2.1 Gestion des tas mémoire sur architecture NUMA

Afin de pouvoir implémenter un ordonnanceur qui prenne en compte l'aspect mémoire sur architecture NUMA, il est nécessaire d'utiliser un allocateur qui d'une part, prenne aussi en compte les architectures NUMA, et qui d'autre part, travaille avec l'ordonnanceur. Cela passe par la synthèse d'informations liées à la mémoire que pourra utiliser notre ordon-

nanceur. On utilisera donc comme interface d'allocation et de communication, l'interface *Heap Allocator* qui permet de gérer finement les aspects mémoire de l'ordonnnancement. Cet allocateur, inspiré de *Streamflow* et doté de fonctionnalités de placement NUMA a été implémenté en collaboration avec Maxime Martinasso de l'équipe INRIA MESCAL de Grenoble. Afin de pouvoir utiliser cet allocateur, une interface spécifique à *MARCEL* a été définie pour exploiter des informations liées à la mémoire afin de guider l'ordonnancement. Cette interface nous offre des primitives d'allocations, de migration mémoire et des statistiques concernant la mémoire allouée. Nous détaillerons cette interface dans la section suivante.

2.2 Un ordonnanceur pour la répartition des bulles

A l'aide de la plateforme *BUBBLESCHED* et de l'allocateur *Heap Allocator*, il est maintenant possible d'écrire un ordonnanceur dirigé par la mémoire sur architecture NUMA. Cette ordonnanceur baptisé *MemAware* agira selon deux algorithmes *mSteal* et *mSpread* en fonction de l'équilibrage en temps réel du travail sur la machine. En effet, l'algorithme *mSteal* sera utilisé pour voler du travail afin de corriger un léger déséquilibre sans pour autant pertuber le reste de la machine. L'algorithme *mSpread* sera utilisé pour reconsidérer dans sa globalité la répartition du travail sur la machine.

2.2.1 Articulation entre mSteal et mSpread

Principe de l'ordonnanceur

L'ordonnanceur *MemAware* utilise les deux algorithmes *mSteal* et *mSpread* afin de réordonnancer les entités. Le choix de l'algorithme dépend de l'équilibrage du travail du niveau topologique par rapport à ses voisins. Un ensemble de niveaux est dit équilibré lorsque la différence de charge de travail entre les différents niveaux n'est pas trop grande. Nous verrons plus en détail cette notion d'équilibrage en section 3.

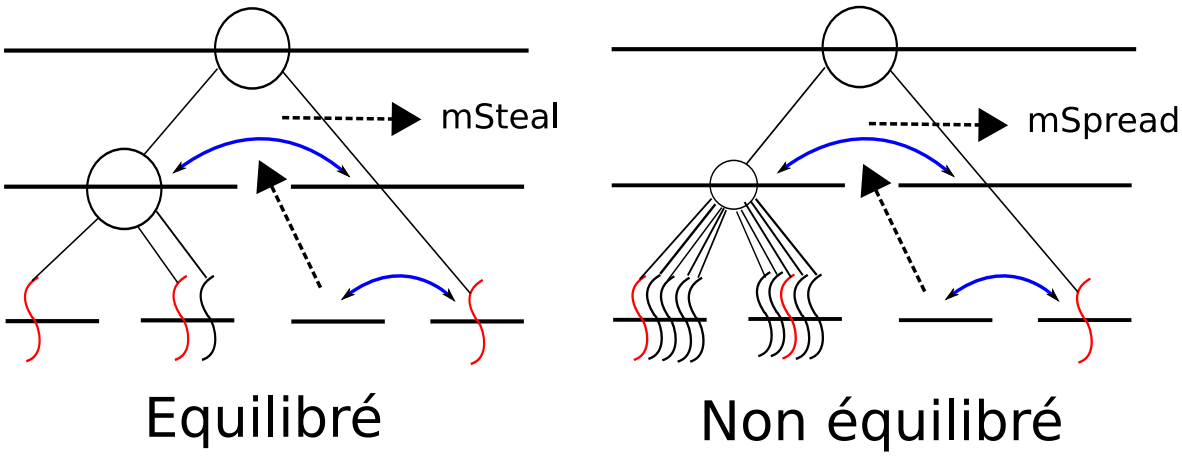


FIG. 2.1 – Equilibre et déséquilibre

Sur la figure 2.1, deux tests d'équilibrage de *MemAware* sont présentés. Les tests locaux d'équilibre entre deux niveaux sont représentés par les double-flèches bleues. Lorsque l'on constate un déséquilibre sur un ensemble de niveaux topologiques, *MemAware* appelle *mSpread* sur leur père afin de rééquilibrer sur l'ensemble de ces niveaux. C'est le cas sur l'exemple de droite où l'on appelle *mSpread* sur toute la machine car les niveaux 1 et 2 sont fortement déséquilibrés. Lorsque deux niveaux sont considérés comme équilibrés, on regarde plus haut dans la machine (flèches en pointillés) et une fois que *MemAware* a atteint le haut de la machine et constaté aucun déséquilibre, c'est le vol de travail *mSteal* qui est lancé, comme sur l'exemple de gauche.

2.2.2 L'algorithme mSpread

L'algorithme *mSpread* est un algorithme glouton qui prend en compte une liste d'entités triées en fonction du poids mémoire. Les entités lourdes en mémoire seront placées là où elles auront auparavant alloué de la mémoire afin que celles-ci ne soient pas obligées de faire des accès distants. Les autres, dont l'aspect mémoire est moins important, servent à rééquilibrer la charge de travail sur l'ensemble des processeurs. Lorsque l'on est amené à diriger une grosse bulle de charge sur plusieurs niveaux topologiques, on peut être amené à l'éclater afin de répartir des sous-entités sur les différents niveaux.

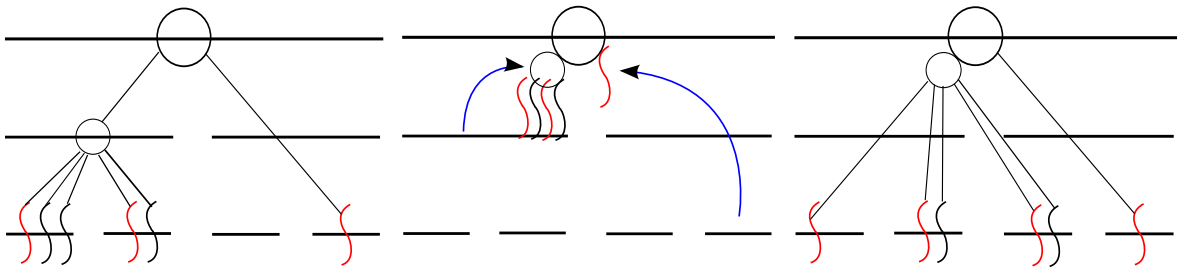


FIG. 2.2 – Algorithme *mSpread*

Cet algorithme est utilisé lorsque l'on constate un fort déséquilibre de répartition du travail sur la machine. En effet, une fonction permet de calculer ce déséquilibre et retourne le niveau topologique à partir duquel on doit rééquilibrer les processeurs.

Limitations de mSpread

Nous devons faire face à plusieurs dilemmes lors de l'exécution de *mSpread*. En effet, il n'est pas toujours possible de placer une entité là où elle utilise le plus la mémoire car il en résulterait un déséquilibre au niveau de la charge de travail (par exemple si toutes les entités ont un bassin d'attraction situé sur le même nœud). Déterminer le placement idéal de l'ensemble des entités sur l'ensemble des listes d'ordonnancement est un problème NP difficile (du type *bin packing*). *mSpread* utilise différentes heuristiques afin de déterminer un bon placement, par exemple placer la moitié des entités sur leur nœud préféré et équilibrer l'ensemble des listes d'ordonnancement afin d'équilibrer la charge.

2.2.3 L'algorithme mSteal

L'algorithme *mSteal* permet à un processeur inactif (qui n'a plus de thread à exécuter sur sa liste locale) de voler du travail à un autre processeur sans le rendre inactif pour autant. Ce processeur recherche dans son voisinage une entité à voler qui sera déterminé selon un score qui est une combinaison linéaire de la charge d'exécution restante, du volume mémoire et de l'épaisseur de la bulle qui la contient. On va privilégier le vol d'une entité dont la charge est importante et dont, lorsque le vol est inter-nœud, le volume mémoire et l'épaisseur de bulle sont faibles afin d'éviter le coût d'une importante migration mémoire ou des accès distants réellement pénalisants. Néanmoins, le déplacement d'une entité volée peut la rapprocher d'une zone mémoire partagée qui était allouée à distance, ce qui peut apporter un bonus au score. Remarquons que dans le cas d'un vol intra-nœud, il n'y a bien entendu pas de migration mémoire mais juste une pénalité de perte de cache.

Pour réaliser le vol, on parcourt les listes d'exécution couvrant directement le processeur afin de trouver une entité à voler. Lorsque plusieurs entités sont disponibles, on choisit celle qui a le meilleur score. Lorsqu'il y a une seule entité, on recherche la meilleure sous-entité à voler en regardant récursivement au sein de celle-ci. En cas de nouvel échec, on cherchera à voler une entité dans les listes des processeurs frères. A la fin du parcours des frères, on obtient pour chaque processeur sa charge totale et l'entité ayant le meilleur score. S'il n'est pas possible de voler une entité à partir des frères, on élargit la recherche sur la partie de la machine couverte par le niveau hiérarchique supérieur jusqu'à considérer l'ensemble de la machine. Il est donc possible que l'ordonnanceur soit contraint de voler une entité située dans un autre nœud, on pourra dans ce cas éventuellement migrer aussi sa mémoire.

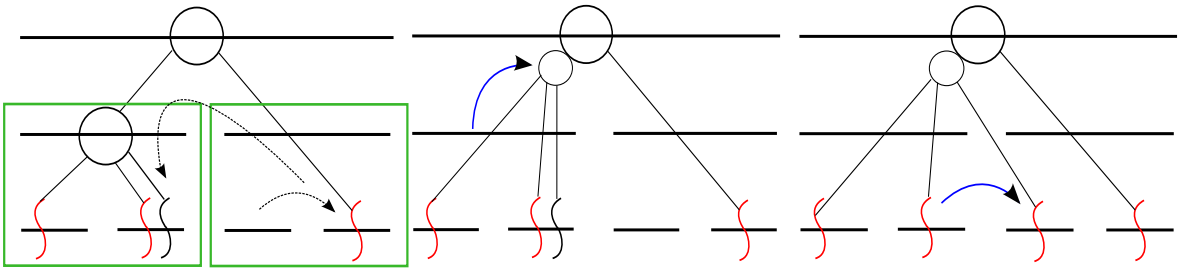


FIG. 2.3 – Algorithme *mSteal*

Sur la figure 2.3, le processeur 3 est inactif. Il va tout d'abord chercher du travail sur le processeur 4 qui ne peut lui en donner car il n'a pas de thread disponible. La recherche s'étend au niveau hiérarchique supérieur, sort du nœud mémoire B (en vert) et cherche du travail dans le nœud A. En descendant, il trouve un thread non exécuté (en noir) sur le processeur 2. Le thread noir est migré vers le processeur 3. Il faut faire attention car la bulle B1 est étalée sur plusieurs nœuds mémoire. Il faut donc la remonter sur le niveau commun recouvrant le processeur source et le processeur destination. Le thread est ensuite volé par le processeur 3.

Dans l'exemple précédent, l'entité volée change de nœud mémoire. L'algorithme *mSteal* prend bien en compte le fait que cette entité doit s'éloigner de sa mémoire en migrant, ce

qui peut conduire à des accès distants. *mSteal* migre donc la mémoire en même temps. De plus, il faut aussi considérer que la bulle remontée en haut de la machine soit maintenant éclatée sur plusieurs nœuds. Comme les threads se partagent les données de la bulle, il y aura nécessairement des accès distants. *mSteal* prend donc cet aspect en compte avant de choisir la meilleure entité à voler.

Limitations de mSteal

Idéalement pour choisir la meilleure entité à voler, il faudrait aussi résoudre est aussi un problème NP difficile. Le choix de l'entité basé sur le modèle du score est une heuristique qui permet de choisir une bonne entité dans le voisinage proche du processeur inactif. Expérimentalement, comme *mSteal* est utilisé de façon complémentaire à *mSpread*, l'entité volée est de bonne qualité mais elle peut ne pas être l'entité cible idéale : la recherche se faisant initialement de manière locale avant de prendre en compte une plus grande partie de la machine. En volant une bonne entité proche du processeur inactif, on peut rater l'entité idéale qui se situerait à un niveau supérieur.

Chapitre 3

Mise en œuvre et évaluation

Ce chapitre décrit quelques points techniques concernant l'interface *Heap Allocator*, la gestion de la mémoire et la conception de l'ordonnanceur *MemAware*, puis présente les performances de cette solution sur quelques programmes de test.

3.1 Interface Heap Allocator

3.1.1 La structure pinfo

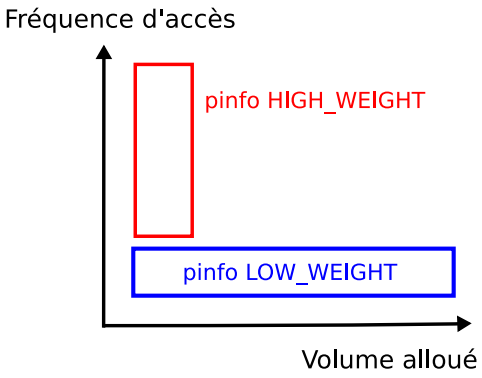


FIG. 3.1 – Exemples de *pinfos*

Cette *structure pinfo* est utilisée dans le but de collecter des informations liées à la mémoire. Cette structure décrit un ensemble de pages allouées sur un même masque de nœuds NUMA et dont la fréquence d'accès (entrée par l'utilisateur lors de l'allocation) est la même. Il y a trois types d'accès différents : LOW_WEIGHT, MEDIUM_WEIGHT et HIGH_WEIGHT. De plus, elle permet d'obtenir des informations importantes comme la répartition d'un tas sur différents nœuds. Dans cette structure, sont enregistrés :

- la politique d'allocation,
- la fréquence d'accès des pages référencées par ce *pinfo*,
- le masque de nœuds sur lesquel sont situées ces pages,
- un tableau dans lequel on pourra comptabiliser le nombre de pages sur chaque nœud à l'aide de la primitive *ma_hupdate_memory_nodes*.

Lors de l'appel à cette primitive, on parcourt les pages décrites par le *pinfo* donné en paramètre, et l'appel système *move_pages* nous renseigne sur leur emplacement. Une fois l'ensemble des pages parcourues, le tableau *nb_touched* de la structure *pinfo* renseigne sur la répartition des pages.

Enfin, un itérateur qui permet de connaître tous les *pinfos* d'un tas est présent dans l'interface. De cette manière, on a accès à l'ensemble des informations du tas. Par exemple, si l'on veut connaître la répartition globale d'un tas sur les différents nœuds d'une machine, on itère tous les *pinfos* du tas avec la fonction *ma_hnext_pinfo* et en actualisant le tableau nb_touched de chaque *pinfo* avec *ma_hupdate_memory_nodes*, on peut alors sommer le nombre de pages sur chaque nœud.

- `int ma_hnext_pinfo(ma_pinfo_t **ppinfo, ma_heap_t *heap);`
- `void ma_hupdate_memory_nodes(ma_pinfo_t *ppinfo, ma_heap_t *heap);`

3.1.2 Création de tas et allocations mémoire

L'interface permet tout d'abord d'allouer de la mémoire dans des tas, chacun pouvant répartir leurs données sur les différents nœuds de la machine. Ces allocations peuvent être dynamiques (malloc/free) ou statiques. Afin de pouvoir prendre en compte les allocations statiques, nous utilisons une primitive d'attachement *ma_hattach_memory* afin que la mémoire statique soit attachée à un tas.

- `void *ma_hmalloc(size_t size, int mempolicy, int weight, unsigned long *nodemask, unsigned long maxnode, ma_heap_t *heap);`
- `void ma_hfree(void *ptr);`
- `void ma_hattach_memory(void *ptr, size_t size, int mempolicy, int weight, unsigned long *nodemask, unsigned long maxnode, ma_heap_t *heap);`
- `void ma_hdetach_memory(void *ptr, ma_heap_t *heap);`

3.1.3 Entités, migration mémoire et fusion de tas

L'allocateur Heap Allocator permet de migrer tout le tas là où l'on le désire avec la primitive *ma_hmove_memory*. Cette primitive prend en entrée un pointeur vers une structure *pinfo* et migre les données. En itérant sur l'ensemble des *pinfos* d'un tas, il est donc possible de migrer tout le tas là où on le désire. De plus, un autre outil est disponible dans notre interface, il s'agit de la fusion de tas. Lorsqu'un thread meurt, il est possible de transférer sa mémoire à un autre thread si on considère par exemple que chaque thread à son propre tas.

- `int ma_hmove_memory(ma_pinfo_t *ppinfo, int mempolicy, int weight, unsigned long *nodemask, unsigned long maxnode, ma_heap_t *heap);`
- `void ma_hmerge_heap(ma_heap_t *hacc, ma_heap_t *h);`

3.2 Implémentation des fonctionnalités de gestion de la mémoire

Pour prendre en compte les relations mémoire / entité / nœud, il est nécessaire d'être capable de gérer finement le placement des pages mémoires sur les nœuds, c'est à dire de savoir décider s'il faut migrer ou non la mémoire d'une entité et savoir techniquement quelles pages migrer. Pour ce faire, nous avons déterminé un ensemble de fonctionnalités pour réaliser les migrations ainsi que deux paramètres : le *bassin d'attraction* d'une entité désignant

le nœud accueillant sa mémoire et l'*épaisseur* d'une bulle caractérisant l'affinité entre ses sous-entités. Pour calculer ces paramètres, nous avons défini l'*attraction* d'une entité pour un nœud comme combinaison linéaire du nombre de pages mémoire réellement accédées sur ce nœud et de leur fréquence d'accès (LOW, MEDIUM, HIGH). Soulignons ici qu'il s'agit de prendre en compte non pas les pages allouées par une entité sur un nœud mais bien celles qui ont été *touchées* c'est à dire effectivement utilisées au moins une fois. De même par *fréquence d'accès*, nous entendons la fréquence d'accès effective à la mémoire physique du nœud, les accès aux données situées dans les caches n'étant pas à prendre en compte. Voyons comment migrer la mémoire d'une entité, mesurer le bassin d'attraction d'une entité et l'épaisseur d'une bulle.

3.2.1 Migration mémoire

La migration mémoire est une opération essentielle pour optimiser la localité des données. Ainsi, lorsqu'une entité est déplacée d'un nœud à un autre, il peut être intéressant que sa mémoire l'accompagne afin de minimiser le nombre des accès distants. Pour effectuer une migration mémoire, on appelle (récursivement dans le cas d'une bulle) la fonction *ma_move_entity_alldata()* qui applique la fonction de migration *ma_hmove_memory* à tous les *pinfos* du tas de l'entité parcourus à l'aide de l'itérateur *ma_hnext_pinfo.*

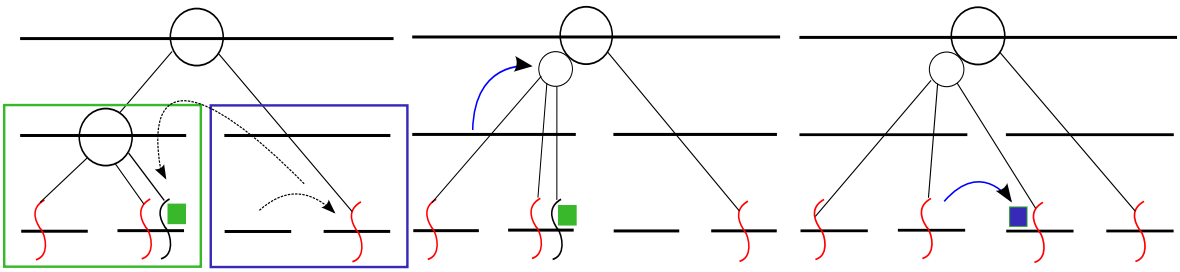


FIG. 3.2 – Migration d'une zone mémoire d'un nœud à l'autre

On voit sur la figure 3.2 la mémoire allouée par le thread volé. Au début, cette zone est sur le nœud de gauche coloriée en vert. Lors du vol du thread vers le nœud de droite, on migre aussi la mémoire locale du thread, recoloriée en bleu sur la figure. La mémoire contenue dans la bulle mère du thread ne sera pas migrée donc ce dernier effectuera des accès distants à la mémoire de la bulle.

3.2.2 Bassins d'attraction principal

Est appelé *bassin d'attraction principal* d'une entité (thread ou bulle), le nœud sur lequel se situent les pages auxquelles elle accède le plus. Placer une entité dans son bassin d'attraction permet de réduire son temps d'accès moyen à ses données et de minimiser la contention du réseau inter-nœud.

Le calcul du bassin d'attraction se fait à l'aide de l'appel de la fonction *ma_most_allocated_node()* qui prend en entrée l'entité considérée. On calcule l'attraction de l'entité et, de façon récursive, de ses sous-entités pour chaque nœud. Pour ce faire, il suffit de parcourir l'ensemble des *pinfos* de son tas à l'aide de l'itérateur *ma_hnext_pinfo()* de l'interface Heap Allocator. Pour chaque *pinfo*, on stocke dans le tableau *nb_touched[]* le nombre de pages concrétement accédées sur chaque nœud à l'aide de la fonction d'actualisation *ma_upgrade_memory_node*(). En fonction de la fréquence d'accès du *pinfo* (LOW, MEDIUM, HIGH), on applique un cœfficient d'utilisation de la mémoire aux valeurs du tableau *nb_touched*. Enfin, en sommant les tableaux *nb_touched* des *pinfos*, on obtient l'attraction de l'entité sur chaque nœud. Si l'entité considérée est une bulle, on répète récursivement cette opération pour ses sous-entités.

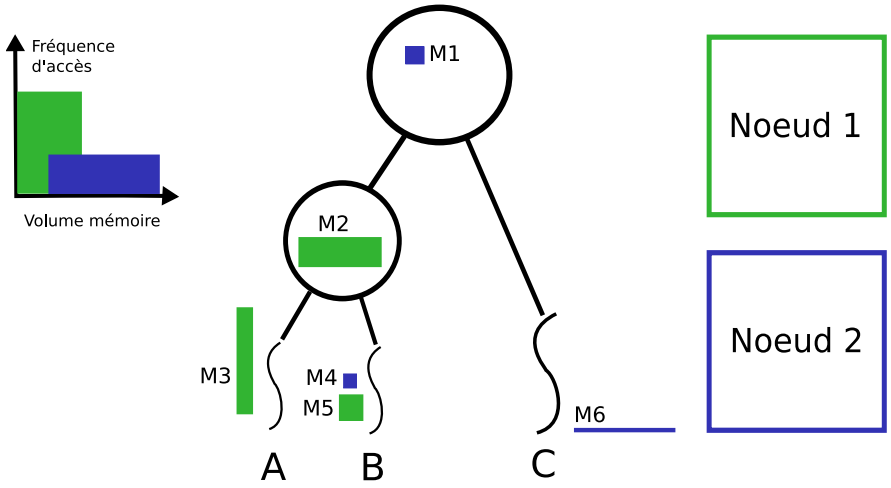


FIG. 3.3 – Calcul des bassins d'attraction

Sur la figure 3.3 la surface d'un rectangle représente l'attraction de l'entité pour un nœud et sa couleur le nœud. Globalement, comme la somme des surfaces vertes (correspondant au nœud 1) est plus importante que celle des surfaces bleues, le bassin d'attraction de la bulle principale est donc le nœud 1. En particulier, on constate que le thread C a alloué beaucoup de mémoire M2 sur le nœud 2, mais comme la fréquence d'accès est très faible, l'attraction en est donc amoindrie. A l'opposé, le thread A a alloué une petite quantité de mémoire M1 sur le nœud 1 très fortement accédée, son attraction est donc importante.

3.2.3 Epaisseur de bulle

Le tas d'une bulle contient la mémoire partagée par ses sous-entités et répartir une bulle sur différents nœuds peut engendrer plus ou moins d'accès distants à la mémoire de son tas. L'épaisseur d'une bulle est la somme des fréquences d'accès vers l'ensemble des nœuds. Cette mesure permet de savoir s'il est intéressant d'éclater une bulle sans être pénalisé par des futurs accès distants. Eclater une bulle améliore la répartition équitable de la charge de travail sur les processeurs mais peut être néfaste pour la répartiton mémoire. Au contraire, garder une bulle intacte permet de conserver la localité mémoire mais diminue la finesse de

la répartition. Il est donc nécessaire de trouver un compromis entre épartition de charge et localité mémoire.

On appelle la fonction *ma_bubble_memory_affinity()* pour calculer l'épaisseur d'une bulle. Cette fonction somme la fréquence d'accès de la bulle sur chaque nœud en parcourant le tas de celle-ci et, contrairement au calcul du bassin d'attraction, il n'y a pas d'appel récursif.

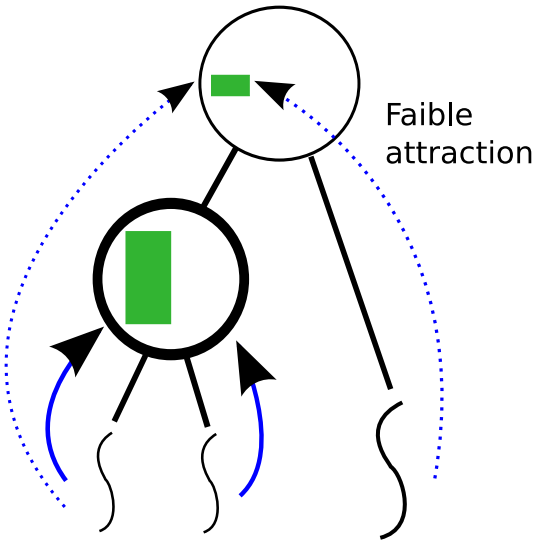


FIG. 3.4 – Epaisseur de bulles

3.3 Implémentation de l'ordonnanceur MemAware

Cette section développe quelques points techniques relatifs à la conception de l'ordonnanceur *MemAware*. On y dévoile des détails d'implémentations des deux algorithmes d'ordonnancement *mSteal* pour voler une entité et *mSpread* pour définir une nouvelle répartion. Pour sélectionner l'algorithme adéquat, on définit la notion d'*équilibre.*

3.3.1 Notion d'équilibre dans MemAware et choix de l'ordonnanceur

Pourquoi existe-t-il deux algorithmes d'ordonnancement *mSteal* et *mSpread* ? L'algorithme *mSpread*, cherchant à optimiser la répartition de la charge suivant l'attraction de bulles, perturbe significativement le travail de l'ensemble des processeurs situés dans sa zone cible. Aussi nous avons développé *mSteal* un algorithme relativement conservateur utilisé pour alimenter un processeur affamé sans pour autant modifier en profondeur la répartition existante : en effet, voler une entité à l'un des processeur ne bouscule pas les autres entités qui peuvent s'exécuter tranquillement. Cependant dans le cas d'un déséquilibre important, il est possible que beaucoup de vols consécutifs s'opèrent et finalement ralentissent la progression. Il s'agit donc de déterminer quel algorithme utiliser en fonction de l'équilibre de toute ou partie de la machine.

L'occupation d'une liste d'ordonnancement et des listes qu'elle couvre est calculée en fonction de la somme des charges des entités et du nombre de threads contenus. Deux sous-ensembles de listes d'ordonnancement sont dites en équilibre s'ils ont une moyenne des occupations des listes comparable. Pour vérifier l'équilibre de la machine, un processeur inactif compare l'occupation de sa liste d'ordonnancement (a priori vide) avec la moyenne des occupations de l'ensemble des listes d'ordonnancement des processeurs frères. Lorsque l'écart des occupations est important, cette partie de la machine est déséquilibrée et l'algorithme *mSpread* sera utilisé sur l'ensemble de ces listes d'ordonnancement. Autrement, la zone est dite équilibrée et le processeur cherche alors un éventuel déséquilibre à un niveau supérieur. Enfin, lorsqu'un processeur ne constate aucun déséquilibre au niveau de la machine, il utilise l'algorithme de vol de travail *mSteal*.

3.3.2 Détails d'implémentation du vol de travail par mSteal

L'algorithme *mSteal* est l'algorithme de vol de travail qui s'active lorsqu'un processeur devient inactif. Le processeur recherche la meilleure entité à voler parmi l'ensemble de la machine. Voyons comment cela se déroule.

Parcours d'un niveau topologique

Avant tout, la liste d'ordonnancement parcourue est verrouillée puis le nombre d'entités ordonnancées sur la liste est compté. Lorsque plusieurs entités sont présentes, on sélectionne l'entité ayant le meilleur score quitte à parcourir récursivement les bulles.

Parcours ascendant de la machine

Afin de déterminer une entité à voler, l'algorithme effectue un parcourt partant de la liste d'ordonnancement du processeur inactif vers celle de la machine entière : on élargit ainsi de proche en proche la recherche d'une entité cible tout en respectant la hiérarchie de la machine considérée. Pour ce faire on utilise la fonction *ma_see* pour rechercher les entités sur une liste donnée. Les fonctions *ma_see_up* et *ma_see_down* sont utilisées pour parcourir les listes de la machine : *ma_see_up* est appelée afin de regarder s'il existe des entités cibles en dessous du père du niveau hiérarchique considéré tandis que *ma_see_down* explore toutes les listes topologiquement situés sous le niveau considéré. Pour rechercher une entité cible on applique donc alternativement *ma_see_down* et *ma_see_up*.

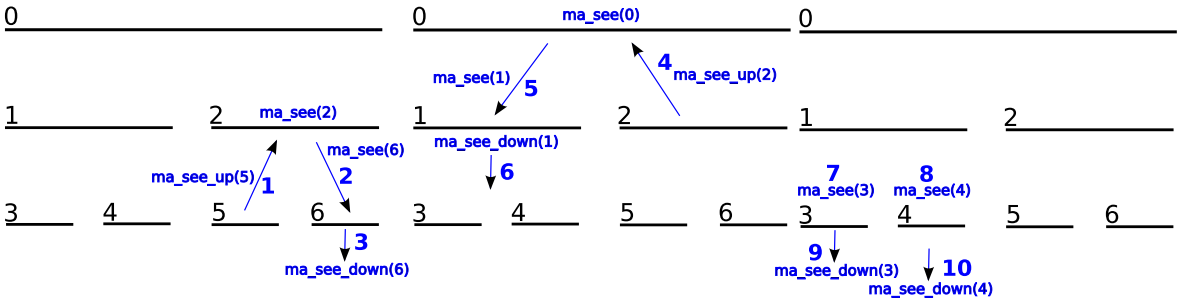


FIG. 3.5 – Parcours des niveaux topologiques de la machine à partir d'un processeur

Remonter les ancêtres sur un niveau commun

L'algorithme *mSteal* a dès à présent choisi l'entité à voler. Pour éviter des problèmes de deadlocks internes, *BUBBLESCHED* imposent par convention que toutes les sous-entités d'une bulle soient placées sur des listes d'ordonnancement couverte par celle de la bulle. Il faut donc s'assurer que la bulle contenant l'entité volée sur le niveau topologique soit commun au niveau source et au niveau destination. Pour cela, on remonte les bulles ancestrales jusqu'au niveau commun sans pour autant remonter leurs sous-entités non concernées par le vol. A la fin, la bulle éclatée se situe sur un niveau couvrant toutes les listes sur lesquelles sont situées les entités.

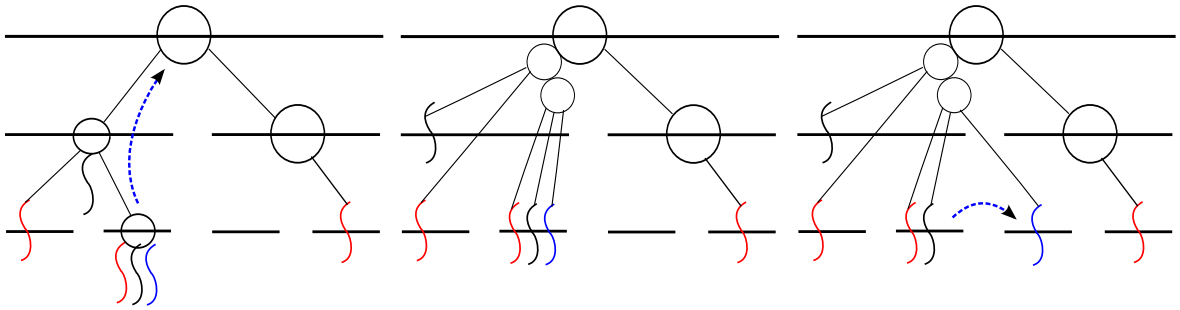


FIG. 3.6 – Exemple de remontée de bulle sur un niveau commun

Migration mémoire

La dernière action de l'algorithme *mSteal* est de migrer l'entité volée d'un processeur à l'autre. Avec la fonction *ma_move_entity_alldata*, l'algorithme migre aussi la mémoire du thread afin de conserver la localité des données.

3.3.3 Détails d'implémentation de la répartition par mSpread

L'algorithme *mSpread* a pour objet de rééquilibrer le travail en-dessous d'un niveau considéré. Ce processus se déroule en plusieurs étapes.

Remonter les entités sur le niveau commun

Afin de répartir toutes les entités situées sous un niveau commun, il faut tout d'abord les référencer. C'est la fonction *ma_lifton_entities()* qui parcourt l'ensemble des niveaux topologiques, référence chaque entités et rassemble les bulles dont les entités sont dispersées sur la machine. En particulier, il s'agit de soigneusement verrouiller les différents niveaux topologiques et les différentes entités de manière à ce que les changement d'états (de structure, de lieu d'ordonnancement) soient effectués avant qu'une autre entité ait accès à ces données. Les entités à répandre sont enregistrées dans un tableau utile à l'équilibrage.

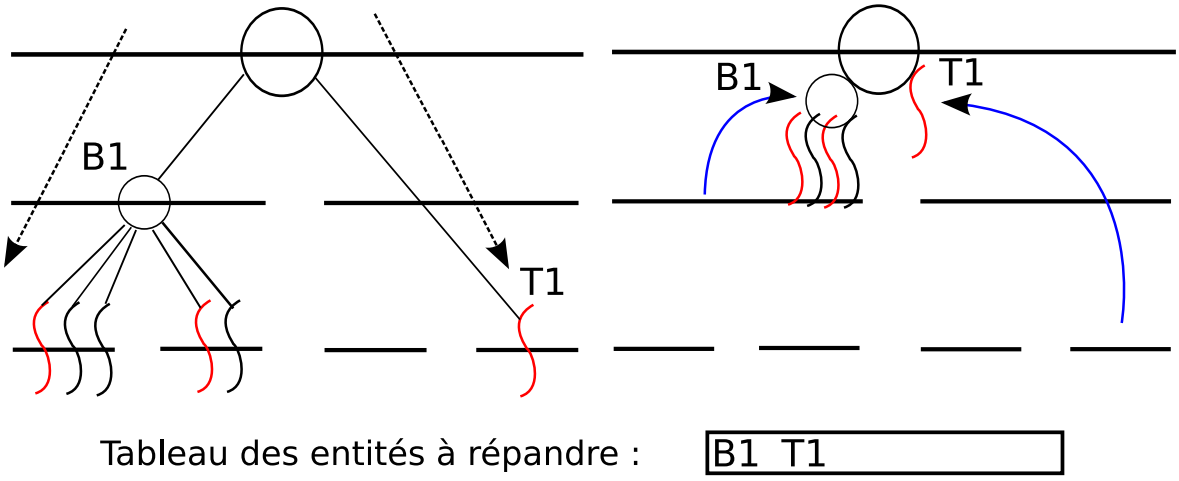


FIG. 3.7 – Remonter les entités sur le niveau à partir duquel répartir

Répartition des bulles

Une fois toutes les entités référencées et les bulles rassemblées, la fonction *marcel_spread_all_entities()* est appelée pour répartir l'ensemble des entités sur les listes couvertes par le niveau considéré. En gros il s'agit de placer les threads sur les listes des processeurs en privilégiant le plus possible les affinités en évitant d'éclater des bulles, en particulier les plus épaisses, autant que possible. Pour cela, *mSpread* compare le nombre de listes d'ordonnancement directement couvertes par le niveau considéré au nombre d'entités à répartir : lorsque le nombre d'entités est inférieur au nombre de listes, *mSpread* éclate, autant que nécessaire, les bulles les plus fines et parmi celles-ci les plus chargées possible afin de pouvoir alimenter l'ensemble des listes.

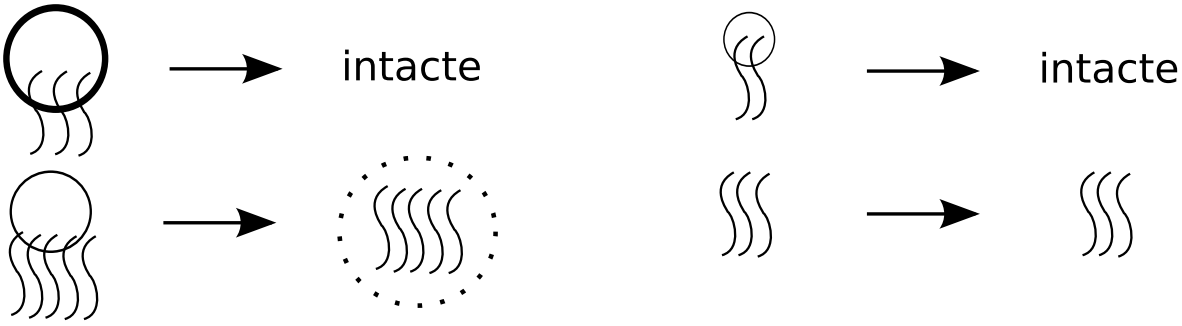


FIG. 3.8 – Eclatement des grosses bulles fragiles

La figure 3.8 montre que seules les bulles dont l'épaisseur est faible sont éclatées. Une grosse bulle de charge avec une forte épaisseur sera assez solide pour rester intacte. Une petite bulle ne sera pas inquiétée non plus car elle ne gène pas la répartition étant donnée qu'elle n'a pas beaucoup de charge par rapport aux autres entités.

Nœuds et caches

Une fois suffisant le nombre d'entités à répartir, l'algorithme *mSpread* doit orienter ces entités (sans les éclater) vers les nœuds et les processeurs. Il s'agit de favoriser à la fois la répartition de la charge et de minimiser la migration mémoire voire de faciliter la réutilisation de cache. Pour ce faire *mSpread* distingue les étages situés au-dessus des niveaux nœuds de ceux situés en-dessous. En effet, lorsque l'on veut placer une entité alors qu'elle se trouve au dessus des niveaux nœuds, il faut faire en sorte que celle-ci retombe sur son bassin d'attraction si son attraction est importante. On compare une combinaison linéaire de l'attraction et de la charge des entités afin d'exécuter sur leur bassin d'attraction les entités à la fois les plus chargées et les plus attirées ; les autres entités seront réparties afin d'équilibrer au mieux la charge sur les différents nœuds. En ce qui concerne les niveaux internes aux nœuds *mSpread* cherche à placer si possible les entités vers le processeur qui l'a exécuté en dernier afin de favoriser la réutilisation du cache et de limiter la contention mémoire.

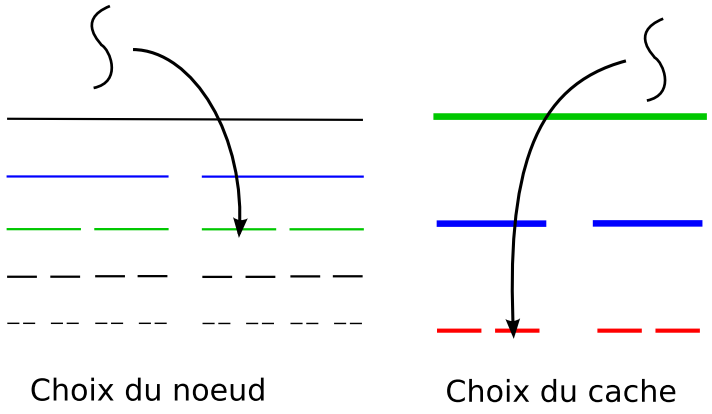


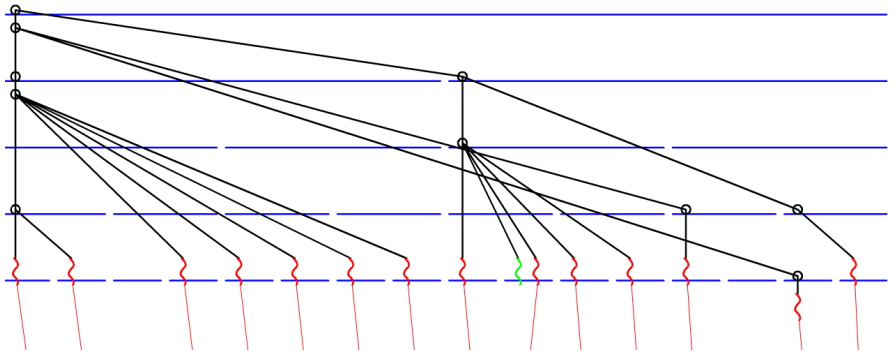
FIG. 3.9 – Choix du nœud puis du cache

3.4 Evaluation

3.4.1 Démonstrastion de MemAware sur une application synthétique

Afin d'illustrer le comportement de l'ordonnanceur *MemAware*, nous avons écrit une application synthétique qui met en jeu une hiérarchie de bulles s'exécutant avec l'ordonnanceur. Tout d'abord *MemAware* répartit la structure d'entités sur l'ensemble de la machine en utilisant l'algorithme *mSpread*. Pour ce faire, certaines bulles sont éclatées et d'autres non afin de pouvoir équilibrer au mieux les listes d'ordonnancement (comme nous pouvons le voir ci-dessous).

Une fois que les entités sont bien réparties, l'exécution du programme commence. Lorsqu'un processeur devient inactif, celui va chercher à voler une entité non exécutée sur un autre processeur comme nous pouvons le voir. Le processeur inactif va chercher à voler le thread vert.



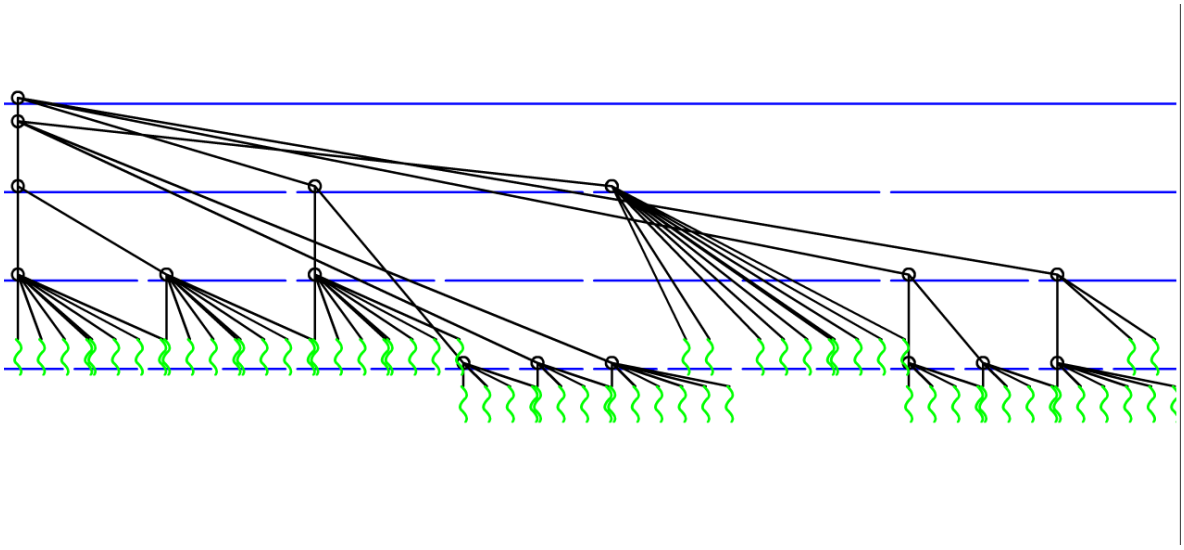
Nous voyons sur la figure du dessous que la bulle qui contient le thread qui sera volé remonte d'un niveau afin d'être commune aux niveaux où sont répartis ses sous-entités.

Une fois la bulle mise en place, il est possible de migrer le thread comme on peut le voir dans la figure suivante.

Tous les processeurs ont maintenant une entité à exécuter, excepté un. En effet, il n'y a plus d'entité à voler car elles sont toutes exécutés par les processeurs. Notons que ce programme ne fait pas appel à la fonction *mSpread* car la machine est équilibrée à tout niveau et un vol suffit à nourrir chaque processeur inactif.

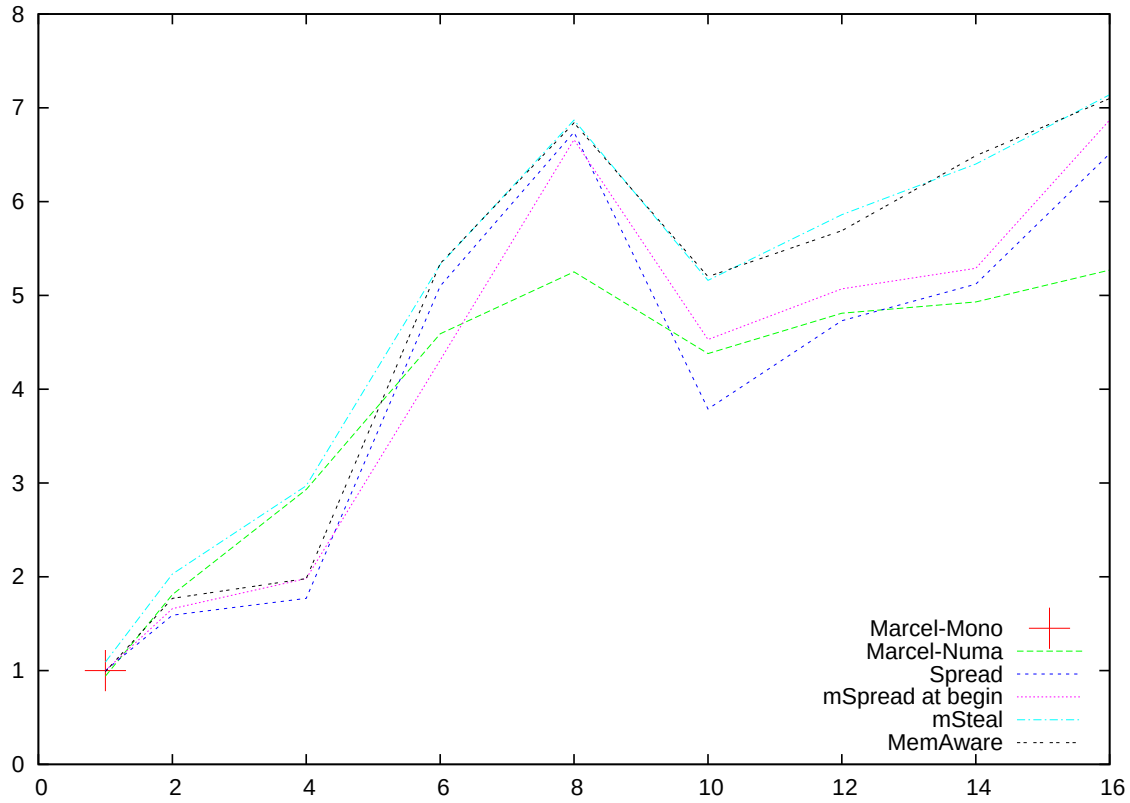
Présentation du test de performance

Afin de déterminer l'efficacité de l'ordonnanceur *MemAware*, un test de performance a été réalisé à l'aide d'une structure de bulle s'exécutant sur un programme qui réalise un nombre important d'accès mémoire. 64 threads sont répartis dans une quinzaine de bulles différentes. Ces threads accèdent à leur mémoire locale de manière intensive en évitant les effets de cache avec un parcours aléatoire de leur mémoire. Chaque équipe de threads réunis dans une bulle accède de manière intensive à de la mémoire partagée. Les tests sont lancés avec un nombre variable de processeurs.



Résultats du test

En fonction du nombre de processeurs utilisés, en tenant compte du temps d'exécution de l'application, on calcule l'accélération qui est le temps d'exécution parallèle sur le temps d'exécution séquentiel (calculé avec un unique processeur).



Nous avons exécuté l'application sur différents ordonnancements de *MARCEL* :

- *Marcel-Mono* : test de référence pour calculer le accéleration.
- *Marcel-Numa* : les entités ne sont pas réparties sur des nœuds bien spécifiques.
- *Spread* : On équilibre une bonne fois pour toute la charge sur les différents processeurs.
- *mSpread* au début du programme : La charge est équilibrée dès le début en tenant compte de l'affinité mémoire des entités dans une bulle.
- *mSteal* : Le vol de travail au cours du programme.
- *MemAware* : notre ordonnanceur.

Tout d'abord, nous remarquons une hausse générale de l'accélération jusqu'à 8 processeurs, puis une nette dégradation des performances dès que l'on dépasse 10 processeurs. Par contre, lorsqu'on remonte vers 16 processeurs, on retrouve une accélération similaire à celui pour 8 processeurs. La contention du bus mémoire limite les accès à la mémoire, donc diminue fortement les performances lorsque le nombre de processeurs devient élevé (au dessus de 8).

Les performances de l'ordonnanceur *Marcel-Numa* semblent stagner à partir de 6 processeurs. En effet, les entités ne sont pas réparties localement sur la machines et il n'y a donc pas de placement explicite de la mémoire. On peut donc avoir beaucoup d'accès distants ou alors, une concentration de la mémoire sur un nœud précis, ce qui engendre de la contention sur ce nœud lors des accès.

Les ordonnanceurs *Spread* et *mSpread* lancé au début du programme ont des performance similaires et meilleures que *Marcel-NUMA*. En effet, tous les deux répartissent les entités sur la machine, ce qui donne une meilleure localité des données car les threads sont répartis sur les processeurs.

mSteal obtient les meilleurs performances. Après une unique itération de l'algorithme *mSpread* afin de placer les entités, l'exécution est lancé et quand un processeur devient inactif, celui-ci vole du travail en migrant la mémoire de celui-ci afin de favoriser la localité des données. Malgré la contention du bus principal, *mSteal* obtient de meilleurs performances avec 16 processeurs qu'avec 8. *MemAware* a un comportement analogue à *mSteal* car il favorise l'algorithme *mSteal* puisque la machine ne présente pas de déséquilibre particulier, d'où des performances assez proches.

Conclusion

La généralisation des architectures SMP munies de processeurs multi-cœurs conduit à la nécessité de distribuer la mémoire pour éviter le goulet d'étranglement des bus centralisés, favorisant ainsi la démocratisation des architectures NUMA. Sur ces architectures, le temps requis pour accéder aux données varie graduellement avec la distance physique à la mémoire et ces contraintes doivent être prises en compte lors de l'ordonnancement des tâches et du placement de leurs données afin d'obtenir des performances optimales.

Nous avons cherché dans ce mémoire à minimiser le temps requis afin d'accéder aux données mémoire en favorisant la localité des données, c'est à dire en coordonnant le placement des flux d'exécutions et des données auxquelles ils accèdent le plus. Pour cela, nous avons tenu compte des affinités mémoires qu'entretiennent différentes tâches afin de regrouper celles-ci au sein des mêmes nœuds tout en veillant à ne pas laisser des processeurs inactifs. Nous avons implémenté une fonction de distribution de charge pour alimenter les processeurs équitablement afin d'obtenir une répartition relativement stable et durable ainsi qu'une fonction de vol de travail qui permet d'alimenter un processeur inactif en ne corrigeant que localement la répartition. Pour cela, nous avons dégagé les critères assurant la qualité du vol : ces critères reposent sur la quantité de charge restante avant la fin de l'exécution de l'entité cible, la quantité de mémoire fortement accédée nécessaire à migrer ainsi que l'affinité que l'entité cible entretient avec ses sœurs au sein d'une bulle. Cet environnement d'exécution utilise une bibliothèque de gestion de tas optimisée pour la programmation parallèle et offrant des fonctionnalités de placement et de statistique dédiés aux architectures NUMA, bibliothèque dont j'ai contribué à la définition et à la mise au point.

Les premières expérimentations montrent une diminution du temps d'exécution de l'ordre de 5% à 10% pour un programme accèdant intensivement à la mémoire. Ces performances peuvent être encore améliorées en optimisant la récolte des statistiques sur la mémoire.

Les perspectives ouvertes par ce travail sont de natures techniques et algorithmiques. Au niveau technique, mentionnons l'extraction automatique de paramètres pour estimer ou observer les fréquences d'accès aux différentes zones mémoires peut être faite de façon à la compilation (analyse statique) et à l'exécution à l'aide des compteurs matériels disponibles sur les microprocesseurs et chipsets. Au niveau algorithmique, il s'agit de concevoir un algorithme répartie qui unifierait *mSpread* et *mSteal* tout en ayant une complexité amortie interessante. Une piste est de calculer de façon incrémentale une répartition modèle sur laquelle les processeurs s'appuiraient pour effectuer leur vol de manière à tendre vers la répartition modèle et de ne corriger que le modèle en cas d'inaction d'un processeur. Ces travaux forment, tout comme ceux effectués dans l'équipe sur les placements des threads par rapport aux controleurs entrée-sortie [11], une brique de base pour la construction d'un algorithme de répartition généraliste sur machine hiérarchique.

Bibliographie

[1] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Cache-fair thread scheduling for multicore processors.

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

[3] Dimitrios S. Nikolopoulos, Thedore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. User-level dynamic page page migration for multiprogrammed shared-memory multiprocessors.

[4] Emery David Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard : A scalable memory allocator for multithreaded applications.

[5] François Broquedis. De l'exécution structurée de programmes OPENMP sur architectures hiérarchiques. June 2007.

[6] Jaydeep Marathe and Frank Mueller. Hardware profile-guided automatic page placement for ccNUMA Systems.

[7] John L. Hennessy and David A. Patterson. Computer architecture, a quantitative approach. *Morgan Kaufmann editions, 1990. 4ème édition.*, 1990.

[8] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. BubbleSched : une plate-forme pour la conception d'ordonnanceurs de threads portables sur machines multiprocesseurs hiérarchiques.

[9] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation.

[10] Steve Whitney, John McCalpin, Nawaf Bitar, John L. Richarson, and Luis Stevens. The SGI Origin Software Environment and Application Performance.

[11] Stéphanie Moreaud. Impact des architectures multiprocesseurs sur les communications dans les grappes de calcul : de l'exploration des effets numa au placement automatique archiques. June 2007.

[12] Yi Feng and Emery David Berger. A locality-improving dynamic memory allocator.