ALMOS : un système d'exploitation pour manycores en mémoire partagée cohérente.

Ghassan Almaless

UPMC - LIP6
4 place Jussieu,
75005 Paris - France
ghassan.almaless@lip6.fr

Résumé

Cet article propose de vérifier expérimentalement que les systèmes d'exploitation monolithiques à mémoire partagée peuvent passer à l'échelle sur les machines cc-NUMA many-cores (plus de 100 cœurs), à condition de distribuer les structures et répartir les traitements en respectant la localité des accès mémoire. Cette étude a été menée sur TSAR, un simulateur de processeur many-cores précis au cycle et au bit, configuré jusqu'à 256 processeurs, et sur ALMOS, un système d'exploitation que nous avons conçu spécifiquement pour TSAR et qui repose sur le même paradigme de programmation en mémoire partagée que d'autres systèmes monolithiques tels que Linux ou BSD. Outre le design du noyau d'ALMOS qui renforce la localité des accès mémoire des tâches, nos contributions portent sur (i) une nouvelle technique d'affinité mémoire que nous nommons Auto-Next-Touch permettant au noyau de migrer des pages physiques d'une tâche d'une manière transparente vis-à-vis de l'application utilisateur ; et (ii) Un ordonnanceur distribué à primitives de synchronisation passant à l'échelle. Nos premières mesures sont prometteuses. Sur deux applications de calcul parallèles, FFT de SPLASH-2 et EPFilter pour le traitement d'images médicales, l'accélération en terme de temps d'exécution augmente linéairement avec le nombre de cœurs jusqu'à 192 cœurs. Notre protocole de barrière de synchronisation entre processus, l'un des mécanismes fondamentaux de synchronisation dans les noyaux, passe à l'échelle jusqu'à 256 processeurs.

Mots-clés : Architecture des systèmes d'exploitation, noyau monolithique distribué, cc-NUMA, many-cores, scalabilité, passage à l'échelle.

1. Introduction

Depuis une dizaine d'années, l'industrie micro-électronique a atteint les limites de la physique concernant la dissipation thermique et l'augmentation de la fréquence d'horloge [1]. Les fabricants se tournent aujourd'hui vers les architectures multi-cœurs intégrées sur puce. Des processeurs de cinquante à cent cœurs sont déjà d'actualité [2][3][4][5] et des many-cœurs allant jusqu'à 1000 processeurs sont à prévoir prochainement [6]. Ces processeurs visent aujourd'hui au moins trois catégories d'applications : les serveurs haute performance type serveur web, les centres de calcul de l'informatique dans les nuages et les applications embarquées de type reconnaissance/traitement d'image.

Pour exécuter ces applications se pose la question des systèmes d'exploitation et de leur passage à l'échelle. Deux courants de pensée s'affrontent aujourd'hui. Le premier courant affirme qu'il faut entièrement repenser le paradigme de programmation, en mémoire partagée, des systèmes d'exploitation et des applications en passant à un paradigme de communication par envois de message [7] [8]. L'autre courant de pensée [9] affirme quant à lui que les systèmes et applications d'aujourd'hui communiquant par mémoire partagée sont d'ores et déjà capables de passer à l'échelle jusqu'à une cinquantaine de cœurs et que les systèmes devraient être capables de passer à des échelles plus larges.

Cet article propose d'examiner la seconde hypothèse et de vérifier expérimentalement s'il est possible

que les systèmes d'exploitation et les applications actuelles, reposant sur la notion de threads et de mémoire partagée, puissent passer à l'échelle. Notre expérience repose sur l'architecture TSAR permettant de simuler, au cycle prêt, un processeur many-cores cc-NUMA. Bien que le point de départ aurait pu être un noyau monolithique préexistant de type Linux, nous avons choisi dans cette expérience de concevoir un nouveau système, ALMOS, pour deux raisons. La première, pour des raisons de faisabilité concernant le temps d'expérimentation : le simulateur de TSAR est précis au cycle et ne permet que de simuler 1000 cycles par seconde rendant ainsi Linux impossible à tester. La seconde pour des raisons de commodité : de façon à passer à de telles échelles, il a été nécessaire de repenser d'une part, l'architecture interne du système et d'autre part, de nombreux algorithmes pour éviter un maximum de synchronisation entre les cœurs. Ainsi il est plus facile de repartir d'un nouveau système que d'adapter le code de Linux qui fait 8 millions de lignes [27]. Notre système d'exploitation est multi-processus, multi-threads. Il utilise la mémoire partagée et les verrous comme moyens de synchronisation. Il offre un sous-ensemble de l'API POSIX avec un suport natif des threads POSIX. Les applications testées n'ont pas été modifiées et n'utilisent aucune primitive système qui n'existerait pas aujourd'hui. Cet article présente les contributions suivantes :

- Une proposition d'une approche de conception de noyaux monolithiques qui délègue la gestion des ressources à des gestionnaires dédiés par cluster, permettant ainsi de distribuer les structures de données noyau et de répartir certains traitements. Cette approche vise à renforcer la localité des accès mémoire sur une architecture cc-NUMA.
- Une architecture d'un ordonnanceur distribué client-serveur passant à l'échelle.
- Une technique noyau d'affinité mémoire automatique et transparente vis-à-vis des applications utilisateur que nous nommons "Auto-Next-Touch".

Les enseignements de cette expérience préliminaire sont déjà nombreux :

- Le résultat majeur de cette expérience est que le banc d'essai SPLASH-2/FFT ainsi que l'application EPFilter issues du monde du calcul haute performance passent à l'échelle en nombre de cœurs. L'accélération, en terme de temps d'exécution, avec 64 cœurs est de 39 pour SPLASH-2/FFT et celle d'EPfilter est de 80 avec 192 cœurs. La courbe d'accélération en fonction du nombre de cœurs est quasi-linéaire.
- Le second résultat de cette expérience est que nous n'avons pas observé de phénomène d'écroulement lors d'un rendez-vous avec 256 cœurs : notre ordonnanceur et ses primitives passant à l'échelle permettent d'avoir des mécanismes de synchronisation noyau passant à l'échelle.
- Le troisième résultat de cette expérience est que notre technique d'affinité mémoire "Auto-Next-Touch" est effective et permet de porter des applications parallèles sans modifications préalable.

La suite de l'article est organisée comme suit. Nous commençons dans la section 2 par présenter l'architecture matérielle cc-NUMA de nos expérimentations, nommée TSAR. Nous décrivons, dans la section 3, le noyau monolithique distribué d'ALMOS et deux de ces mécanismes internes, à savoir, l'ordonnanceur distribué client-serveur et l'affinité mémoire "Auto-Next-Touch". Dans la section 4, nous présentons les résultats de nos expérimentations effectuées sur une configuration matérielle de TSAR allant jusqu'à 256 coeurs sur (i) le passage à l'échelle de nos applications de test sur ALMOS ; (ii) le passage l'échelle de notre ordonnanceur et le mécanisme de synchronisation par barrière ; et (iii) la validation de la technique d'affinité mémoire "Auto-Next-Touch". La section 5 présente quelques travaux apparentés. Enfin, nous concluons dans la section 6.

2. Tera-Scale Architecture (TSAR)

L'architecture TSAR [10], définie par le LIP6 en coopération avec BULL, est constituée de plusieurs clusters interconnectés par un réseau sur puce (NoC) appelé DSPIN [11] (Distributed, Scalable, Predictable, Integrated Network) dont la topologie est une grille 2D. Un cluster est une agglomération de ressources mémoire, processeurs et quelques périphériques internes. La figure 1 illustre cette architecture.

L'architecture en multi-clusters est extensible jusqu'à 1024 clusters. Avec 4 processeurs par cluster, cela donne une configuration maximale de 4096 processeurs. L'approche multi-clusters permet potentiellement de cadencer chaque cluster avec une horloge différente et donc d'adapter la fréquence de chaque cluster en fonction de sa charge.

Le choix des processeurs est basé sur des cœurs RISC 32 bits simples, c'est-à-dire, sans prédicteurs de

branchement, ni exécution spéculative ou dans le désordre. Ce type de cœur offre le meilleur rapport (MIPS/microWatt) [28]. Chaque cœur possède deux caches L1 séparés pour les instructions et les données. Ces L1 sont des caches d'adresses physiques. Chaque cœur dispose également d'une MMU à deux TLB séparées pour les instructions et les données. Les composants L1 et MMU sont indépendants du type du cœur de processeur 32 bits choisi (MIPS, ARM, etc.). L'espace d'adressage physique est sur 40

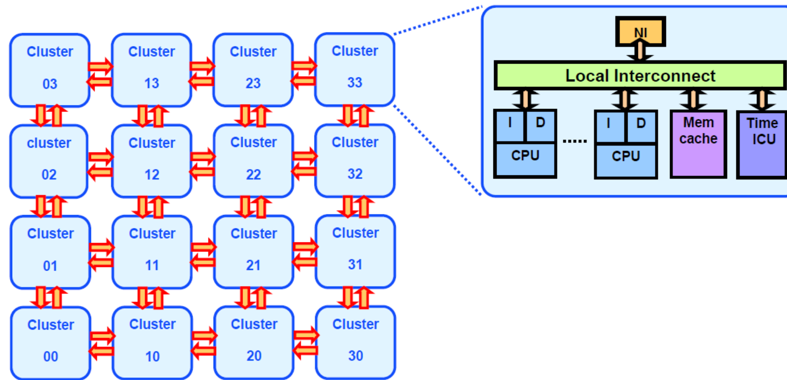

FIGURE 1 – TSAR est une grille-2D de clusters : chaque cluster contient 4 cœurs, un contrôleur DMA (Direct Memory Access), un concentrateur d'interruptions, un cache-mémoire, et une interface réseau pour le NoC (Network Interface). Ces différents composants sont liés par un interconnect local.

bits ce qui permet d'adresser 1 Tera octets de mémoire physique. Cet espace d'adressage physique est strictement partitionné, en *segments* de mémoire physique contigus, entre les clusters de l'architecture. Cela signifie que, connaissant une adresse physique on peut en déduire l'identité du cluster qui gère le segment mémoire physique contenant cette adresse. Dans une architecture possédant 1024 clusters, chaque cluster gère un segment mémoire allant jusqu'à 1 Gigaoctets. La gestion de l'accès vers un segment mémoire physique d'un cluster est faite par un contrôleur mémoire faisant également office d'un cache à stratégie "Write-Back" [29]. Il est appelé cache-mémoire.

Dans TSAR, la cohérence mémoire est assurée par le protocole DHCCP (Distributed Hybrid Cache Coherence Protocol) qui se base sur l'approche de répertoire distribué assurée par les cache-mémoires. Dans ce protocole, lors d'un miss au niveau L1, le cache L1 adresse sa requête directement au cache-mémoire gestionnaire du segment concerné. Si le cache L1 est dans le même cluster cela revient à faire un accès local, sinon c'est un accès distant. Lors d'une écriture faite par un processeur, son L1 transmet cette écriture au cache-mémoire gestionnaire de la ligne de cache concernée. Le cache-mémoire invalide ou met à jour toutes les copies de cette ligne de cache de tous les L1.

Les caches L1 utilisent la stratégie "Write-Through" [29] permettant aux répertoires des caches-mémoire d'être toujours à jour, simplifiant ainsi le protocole de cohérence DHCCP de TSAR. L'inconvénient est que les caches L1 génèrent systématiquement beaucoup de trafics puisque chaque écriture se traduit par une transaction sur le réseau local s'il s'agit d'une écriture locale ou via le NoC s'il s'agit d'une écriture distante. De point de vue système d'exploitation, le noyau a un rôle primordial afin de renforcer la localité des accès mémoire des processeurs en limitant les transactions distantes sur le NoC. Ceci est une condition nécessaire pour exploiter efficacement cette architecture many-cores.

3. Advanced Locality Management Operating System (ALMOS)

Le but de cette section est de présenter le système d'exploitation ALMOS. Nous allons tout d'abord présenter l'objectif de ce système. Ensuite nous présentons l'architecture de son noyau. Enfin, nous décrivons deux de ces mécanismes noyau concernant l'ordonnancement distribué client-serveur et l'affinité mémoire "Auto-Next-Touch".

3.1. Motivations et approche de conception

ALMOS est un nouveau système d'exploitation dédié à la recherche sur le passage à l'échelle des couches applicatives sur les architectures many-cores cc-NUMA. La localité des accès mémoire est primordiale pour le passage à l'échelle des applications multi-threads sur ces architectures, et particulièrement sur TSAR. ALMOS vise à renforcer la localité des accès mémoire faits par les tâches, en mettant en œuvre des structures de données, mécanismes et politiques permettant une gestion adaptée des ressources globales mémoire et processeurs. ALMOS vise à rendre transparent la topologie physique de l'architecture, son caractère distribué et la gestion de ses ressources, pour les applications utilisateur.

Dans le but de renforcer la localité des accès mémoire, la conception du noyau d'ALMOS (i) distribue la gestion des ressources mémoire et processeurs à travers la mise en place des *gestionnaires de ressources par cluster* (ii) rend explicite l'interaction inter-clusters.

S'appuyer sur cette organisation distribuée du noyau permet de renforcer la localité des accès mémoire des tâches puisque les demandes en ressources mémoires peuvent être satisfaites localement par le gestionnaire de ressources du cluster dans lequel le noyau a placé cette tâche. Ces demandes peuvent être des allocations de pages physiques en réponse à des défauts de pages ou des allocations mémoires des objets noyau lors du traitement des services systèmes. Enfin, comme les structures du noyau sont distribuées, le noyau n'a pas la notion d'un état global des ressources. Cela l'oblige à mettre en place un mécanisme décentralisé de prise de décision concernant le placement des tâches et de leurs objets mémoires, ainsi que l'équilibrage de charge des processeurs. Un tel mécanisme est absent dans la version actuelle d'ALMOS mais il a fait l'objet d'une spécification. La figure 2 donne un aperçu de ce système.

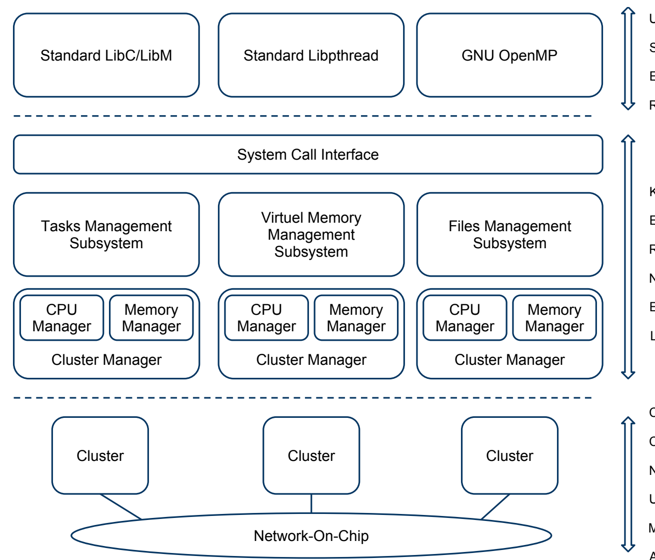

FIGURE 2 – Un aperçu du système d'exploitation ALMOS

Actuellement ALMOS dispose d'un certain nombre de bibliothèques et exécutifs système tels que, une bibliothèque C assez complète, une bibliothèque mathématique, une bibliothèque Pthread assez complète et l'exécutif GNU-OpenMP. Le noyau d'ALMOS dispose d'un ensemble de sous-systèmes primordiaux permettant de gérer les tâches et leurs espaces virtuels ainsi qu'un système de gestion de fichiers. Ces sous-systèmes sont articulés autour d'un certain nombre de gestionnaires de ressources de cluster nommés "Cluster Manager". Un "Cluster Manager" est un objet physiquement placé dans la zone mémoire physique de son cluster. Un gestionnaire de cluster contient, entre autres, un gestionnaire mémoire et un gestionnaire processeur pour chaque processeur physique appartenant à ce cluster.

Le gestionnaire mémoire est responsable de la gestion des pages physiques de son cluster. Il dispose d'un cache d'objets noyau et il est capable de répondre aux requêtes d'allocation et de libération mémoire de n'importe quel type. Ainsi, lors de l'exécution d'un service système, si une tâche a besoin de créer un objet, elle s'adresse au gestionnaire mémoire du cluster où elle s'exécute.

Le gestionnaire processeur est responsable de la gestion des temps et alarmes relatifs à son processeur. Il contient un gestionnaire d'événements locaux et distants. Les évènements locaux permettent à une tâche, en mode noyau, de différer un traitement résultant dans un chemin critique (p.ex. mise à jour du temps suite à une interruption horloge) tandis que les événements distants permettent de déporter l'exécution d'une fonction sur un autre processeur (p.ex. en cas d'équilibrage de charge interruption) ou d'invoquer un service distant.

Chaque gestionnaire de processeur dispose de son propre ordonnanceur. Cet ordonnanceur, local par processeur, applique différentes politiques d'ordonnancement réclamées par les tâches qui lui sont affectées (placées).

3.2. Ordonnanceur distribué client-serveur

Nous commençons d'abord par identifier le problème de passage à l'échelle des primitvies d'ordonnancement telles qu'elles existent aujourd'hui dans d'autres noyaux monolithiques comme Linux ou BSD. Ensuite, nous présentons l'architecture de notre ordonnanceur qui permet de remédier à ce problème.

L'ordonnanceur est un composant essentiel d'un noyau qui gère, entre autres, l'ensemble des tâches à l'état prêt. Selon les politiques d'ordonnancement fournies par l'ordonnanceur, et pour des raisons de performance, cet ensemble de tâches est réalisé par liste chaînée ou un arbre de priorité. Afin de garantir la cohérence de cet ensemble, lors de la réalisation des primitives d'ordonnancement, un mécanisme de verrou est utilisé.

La présence de verrou est lié au fait que dans cette approche classique de la réalisation de l'ordonnanceur, certaines primitives peuvent être exécutées par des tâches qui s'exécutent sur d'autres processeurs. L'état d'un ordonnanceur est donc partagé. Les opérations distantes telle que l'opération "réveiller une tâche" sérialisent d'autres opérations comme "mise en attente passive" ou encore "céder le processeur courant".

Prenons le cas de la primitive "réveiller une tâche". Cette primitive est appelée par une tâche A qui cherche à réveiller une tâche B. Cette primitive prend le verrou de l'ordonnanceur qui gère la tâche B afin de chaîner cette dernière dans la bonne liste d'attente à l'état prêt après avoir recalculé sa priorité. Cette opération est coûteuse. Elle est payée par la tâche A qui peut-être sur un autre processeur que le processeur cible, celui de la tâche B. Par conséquent, elle pénalise également toute opération d'ordonnancement sur le processeur de la tâche B puisqu'elle nécessite une prise de verrou de cet ordonnanceur.

Dans une architecture NUMA, l'accès distant lors de la réalisation de l'opération "réveiller une tâche" ajoute un surcoût supplémentaire. Ce surcoût peut-être toléré si la tâche A réveille un nombre faible de threads. Il devient intolérable pour des applications fortement mulit-threads où le nombre de threads à réveiller est important lors de la notification de la fin d'un rendez-vous par barrière ou lors de la notification de l'occurrence de l'évènement attendu sur une variable de condition. Afin de passer à l'échelle ce type de mécanismes de synchronisation, il est indispensable de réduire le surcoût de l'opération "réveiller une tâche".

Nous proposons dans le noyau d'ALMOS un ordonnanceur suivant un modèle de traitement client-serveur. L'ordonnanceur, par processeur, joue le rôle de serveur. Il attribue à chaque tâche, lors de son admission, un identifiant local. Les primitives d'ordonnancement sont exécutées uniquement par une tâche appartenant à cet ordonnanceur aux différents points d'ordonnancement (lorsqu'une tâche cède son processeur ou en cas d'une interruption horloge, etc.). Les clients sont toute tâche A cherchant à réveiller une autre tâche B.

L'opération "réveiller une tâche" est divisée en deux temps : (i) le client, la tâche qui cherche à réveiller une autre, signale cet évènement au serveur cible en lui présentant l'identifiant local de la tâche et un identifiant désignant l'opération demandée ; (ii) le serveur cible exécute l'opération.

Les conséquences directes de cette organisation sont : (i) l'ordonnanceur n'a pas à protéger ces ressources

par des verrous (il suffit de désactiver les interruptions ou lever le niveau d'exécution pour exécuter une section critique) ; (ii) les opérations "réveil" / "mise en attente" peuvent se faire en parallèle sans aucune contention.

Dans la section 4.1, nous comparons expérimentalement les deux approches : classique (à verrou) et distribuée (client-serveur).

3.3. Auto-Next-Touch : une affinité mémoire automatique

Nous commençons d'abord par identifier trois problèmes liés à la phase d'initialisation dans un schéma de traitement typique fait par une application parallèle mulit-threads. Ensuite, nous présentons une solution actuelle "Migration au prochain accès" ou "Next-Touch" permettant de résoudre deux de ces problèmes. Toutefois, nous montrons que cette solution ne résoud pas le troisième problème et pose elle-même de nouveaux problèmes. Enfin, nous présentons notre solution, "Auto-Next-Touch" qui permet de remédier à ces limitations et de répondre à l'ensemble de ces problèmes.

Le schéma de traitement typique d'une application parallèle multi-threads, illustré dans la figure 3, commence par une phase d'initialisation séquentielle suivie d'une phase de traitement parallèle. Les données initiales sont préparées dans la phase d'initialisation faite par un seul thread. Or la stratégie consistant à contrôler le placement d'une page physique lors du défaut de page engendré par le premier accès à une adresse virtuelle, "First-Touch", permet d'allouer la page physique demandée depuis la mémoire locale au processeur sur lequel le thread s'exécute. La conséquence de cette stratégie "First-Touch" est que lors de la phase parallèle, les threads dédiés au traitement d'un sous-ensemble de données initiales y accèdent à distance, ce qui impacte sérieusement les performances dans un contexte NUMA. D'autre part, lors de la phase d'initialisation, les allocations mémoire faites par l'unique thread peuvent potentiellement générer une pénurie de mémoire physique locale. Ce qui impactera les allocations mémoires faites, dans la phase parallèle, par les threads localisés dans le même cluster que le thread d'initialisation tant que les threads distants n'ont pas effectué/terminé la migration de leurs pages. Dans ce cas de figure, les threads du cluster en question vont allouer à distance leurs pages physiques réclamées à ce moment dans la phase parallèle. Ce qui impactera également les performances.

L'idée de la stratégie "Next-Touch" est de laisser au programmeur le soin d'indiquer au noyau, à la fin de la phase d'initialisation, les zones mémoires (virtuelles) qui font l'objet de traitement parallèle. Le noyau parcours la table des pages du processus et marque chaque page physique concernée pour une migration au prochain accès. Ainsi, lors de la phase parallèle, les pages physiques contenant les données à traiter par un thread vont être migrées vers la mémoire locale du thread. Néanmoins, cette technique présente deux limitations : (i) une complexité d'usage puisque le programmeur doit paramétrer son application, étudier le profil d'accès mémoire et informer le noyau sur les adresses des zones mémoire à migrer lors du prochain accès fait par une autre tâche ; (ii) une réécriture des applications existantes. D'autre part, elle ne répond pas au problème de dispersion potentielle des allocations mémoires lors de la phase parallèle.

Nous proposons dans le noyau d'ALMOS une solution que nous nommons "Auto-Next-Touch" qui permet de (i) décharger le programmeur d'indiquer les zones mémoires à migrer au début de la phase parallèles ; (ii) porter des applications existantes multi-threads sans modification préalable ; et (iii) imposer une politique d'allocation mémoire afin de garantir une disponibilité en mémoire physique locale pour la phase parallèle de traitement.

Notre solution se repose sur deux techniques : la première, consiste à détecter l'instant où un processus mono-thread passe en multi-thread. A ce moment le noyau parcourt, automatiquement et d'une manière transparente, l'ensemble des régions virtuelles appartenant au tas du processus en marquant les entrées des tables de pages physiques existantes comme "Migration au prochain accès". La deuxième, consiste à limiter la quantité de mémoire physique locale que le thread peut allouer lorsqu'il est l'unique thread de son processus et débloquer le reste quand le processus passe en mulit-thread. Cette limitation de quantités mémoire a l'inconvénient de pénaliser les performances d'une application mono-thread, puisqu'elle introduit une dispersion des allocations mémoire et par conséquent dégrade fortement la localité des accès mémoire. Nous avons fait le choix de payer ce prix, car nous pensons que la majorité écrasante des applications destinée à s'exécuter sur les architectures many-cores sont mulit-threads.

Dans la section 4.2, nous comparons expérimentalement les performances, en terme de temps d'exécu-

tion d'une application avec et sans "Auto-Next-Touch".

4. Résultats

Le but de cette section est de décrire les expérimentations que nous avons menées et leurs résultats afin, d'une part, de valider le passage à l'échelle de notre approche de conception du noyau d'ALMOS et, d'autre part, de savoir si le modèle de programmation par mémoire partagée peut être utilisé sans recours à des primitives spécifiques et sans réécriture des programmes. Nous commençons par décrire les applications employées dans cette expérimentation ainsi que la configuration matérielle utilisée. Ensuite, nous présentons les résultats et notre conclusion.

Les applications utilisées sont : FFT de SPLASH-2 [13] et une application de traitement d'images médicales nommé EPfilter. Le choix de ces deux applications, de classe HPC, est motivé par la nécessité d'avoir des applications parallèles passant à l'échelle ce qui nous permet d'écarter le problème de la qualité des applications en cas d'anomalie dans les résultats. Ces deux applications sont écrites en C et utilisent l'API POSIX Threads pour exprimer le traitement parallèle. Elles suivent le même schéma de traitement illustré par la figure 3. Comme cette figure le montre, l'application est divisée en plusieurs

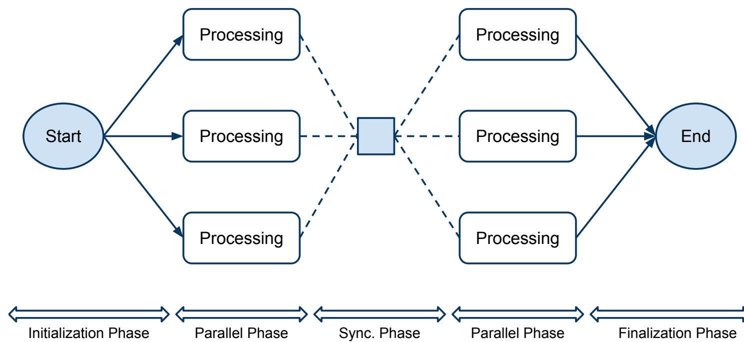

FIGURE 3 – Schéma général de traitement pour les applications parallèles SPLASH-2/FFT et EPfilter

phases. La phase d'initialisation permet de remplir la matrice de calcul pour la FFT, ou de lire l'image depuis le disque pour EPfilter et de lancer ensuite autant de tâches "worker" qu'il y a de processeurs dans la plate-forme matérielle. Les tâches "worker" alternent deux phases : une phase de traitement parallèle et une phase de synchronisation par barrière. La FFT utilise six phases de synchronisation par barrière tandis que EPfilter en utilise trois.

La taille du problème pour l'application FFT est celle recommandée par les auteurs de l'application, pour un nombre de processeurs allant de 1 jusqu'à 64, à savoir, 65536 (M=16) points complexes. L'algorithme est décrit dans [14].

L'application EPfilter traite une image de 960x960 pixels de 2 octets. Le traitement consiste à appliquer un filtre de convolution de 201x35 pixels. Le filtre de convolution est constitué d'un filtre vertical (35 pixels) et d'un filtre horizontal (201 pixels). Chaque tâche "worker" traite 960/N lignes de l'image, N étant le nombre de tâches. La configuration matérielle du simulateur CABA (Cycle Acurate Bit Acurate) de TSAR est la suivante :
– Le nombre de clusters varie de 1 à 16 pour la FFT et de 1 à 64 pour EPfilter.
– Le nombre de processeurs par cluster est de 4 pour la FFT et de 3 pour EPfilter.
– Le type de processeurs est MIPS32.
– La taille d'un cache mémoire est : 256 Ko (16 ways, 256 sets, 64 octets par ligne).
– La taille d'un cache L1 est 16 Ko (2 ways, 64 sets, 64 octets par ligne).

– La dimension d'une TLB est 64 (4 ways, 16 sets).

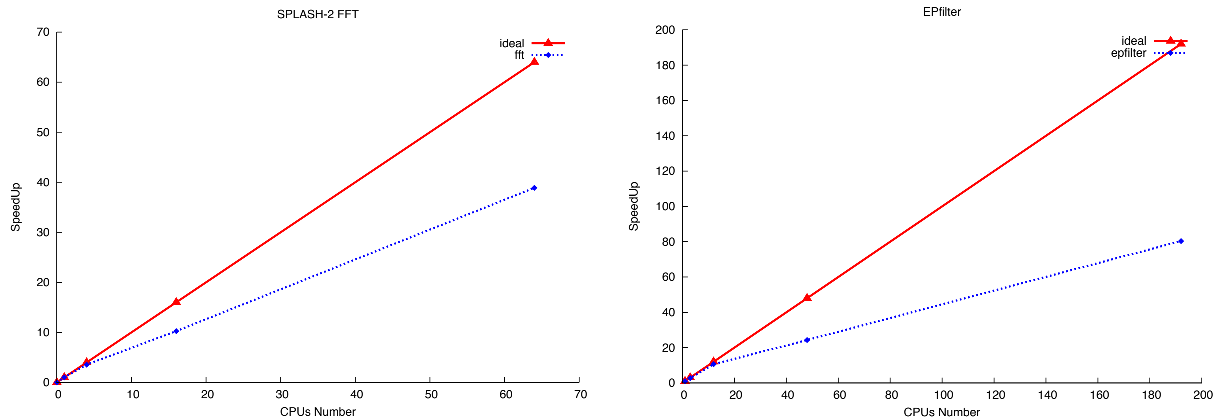

FIGURE 4 – Passage à l'échelle du noyau ALMOS

Les courbes, fft et epfilter, de la figure 4 montrent une accéleration linéaire jusqu'à 64 processeurs pour la FFT et 192 processeurs pour EPfilter. Ceci est un résultat encourageant concernant la scalabilité du noyau et son approche de conception renforçant la localité des accès mémoire faites par les threads de ces deux applications. L'acceleration est moins élevée dans le cas de l'application EPfilter car le résultat de traitement fait par chaque tâche sur son sous-ensemble de pixels ne regroupe pas les écritures, au contraire, chaque pixel traité est tout de suite transposé (écrit) avant de traiter le prochain pixel. Tandis que dans le cas de l'application FFT, chaque tâche ramène dans un buffer local une partie de ses points complexes à traiter, elle effectue le traitement, puis elle transpose ces points. Cette stratégie de traitement est plus efficace car le Write-Buffer des caches L1 est mieux utilisé dans ce cas. Par conséquent, les processeurs sont moins bloqués par les écritures et offrent donc une meilleurs performance.

4.1. Service de synchronisation par rendez-vous

Le service de synchronisation par rendez-vous offert par le noyau implique la mise en attente (sleep) de chaque tâche arrivant sauf la dernière qui doit réveiller (wakeup) les N-1 tâches en attente de cet évènement (N étant le nombre de tâches). Comme présenté dans la section 3.2, nous proposons de déléguer le réveil d'une tâche à l'ordonnanceur gérant cette tâche.

Afin de comparer les deux approches, classique et client-serveur, nous avons repris l'application de SPLASH-2/FFT [1] et augmenté le nombre de processeurs dans TSAR à 256 (64 clusters). Nous avons fixé la politique d'ordonnancement à Round-Robin avec une priorité fixe dans le but de simplifier au maximum le traitement à faire. Par conséquent, lors d'un sleep/wakeup dans l'approche classique, il n'y a pas de calcul de priorité et seulement une prise de verrou ainsi qu'une suppression/insertion dans une fille d'attente. La figure 5 présente les résultats obtenus. Elle montre le coût, en nombre de cycles, nécessaires pour réveiller les tâches lors de deux barrières (b0 et b1) de synchronisation pour un nombre de tâches allant de 4 à 256. Deux versions existent : v0 correspond à une version du noyau utilisant l'approche classique et v1 une version du noyau utilisant l'approche client-serveur.

Les résultats montrent que l'approche client-serveur est bien plus performante. Elle présente une meilleure scalabilité. Ce qui indique clairement la pertinence de cette nouvelle solution.

1. la taille du problème reste celle recommandée jusqu'à 64 processeurs, mais le but, ici, n'est pas d'évaluer l'accélération sur 256 processeurs.

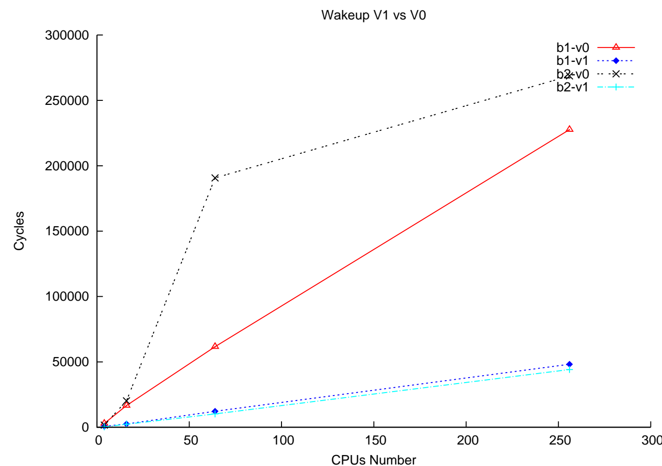

FIGURE 5 – Evalutation du coût, en terme de nombre de cycles, du service noyau de barrière de synchronisation. La version v0 correspond à l'ordonnanceur réparti à synchronisation par verrou. La version v1 correspond à l'ordonnanceur réparti selon un modèle client-serveur

4.2. Affinité des pages physiques

Comme nous l'avons vu, dans la figure 3 décrivant le flot de traitement des deux applications SPLASH-2/FFT et EPfilter, le traitement commence par une phase séquentielle d'initialisation faite par la tâche initiatrice. Lors de cette phase, un ensemble de zones mémoire sont allouées et initialisées. Ce qui implique une allocation des pages physiques correspondantes depuis la mémoire locale au processeur sur lequel la tâche initiatrice s'exécute.

La stratégie "Auto-Next-Touch" présenté dans la section 3.3 permet de migrer automatiquement les pages d'un thread en les relocalisant sur son cluster lorsqu'elle sont accèdées pour la première fois arpès sa création.

La figure 6 montre l'accélération de l'application SPLASH-2/FFT en utilisant notre nouvelle stratégie "Auto-Next-Touch" par rapport à un comportement par défaut sans aucune directive de migration. Cette expérimentation nous montre la différence avec et sans cette stratégie de migration au prochain accès. Ceci montre que (i) l'accélération en suivant une stratégie automatique de migration au second accès est sans surcoût pour la phase de traitement parallèle et (ii) le noyau fait un choix d'allocation correcte concernant le placement des pages physiques lors des défauts de page.

5. Travaux apparentés

La localité des accès mémoire est un facteur important pour les performances des machines NUMA. Ce facteur dépend des stratégies d'allocation des ressources processeurs et mémoire et est connu [15] depuis les premières machines NUMA, CM* [16].

Des solutions articulées autour de la migration et réplication des pages de mémoire physique au niveau noyau existent [17] [18] [19]. L'approche nécessite des informations fournies par le matériel comme les miss de cache et les miss TLB. Nous pouvons les inscrire dans le cadre d'une découverte dynamique de la localité des accès mémoire. Ces études soulignent [16] l'importance de la prise en compte des effets de cache des processeurs lors de l'ordonnancement des tâches ainsi que la notion d'affinité vis-à-vis des clusters. D'autre part, pour mieux exploiter les architectures NUMA, les programmes doivent être exprimés en parallélisme gros-grain et éviter le faux partage induit quand une page contient des structures

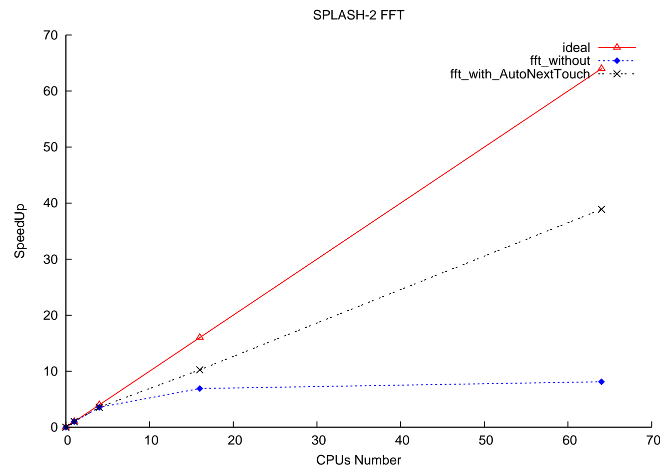

FIGURE 6 – Évaluation de la stratégie de migration automatique des pages physique au second accès proposée par le noyau

de données indépendantes accédées par plusieurs tâches s'exécutant sur plusieurs processeurs [17].

S'appuyer sur le système d'exploitation et son système de mémoire virtuelle [18] [19] permet, lors d'un défaut de page, de choisir d'allouer une page depuis une zone mémoire locale, ou si la page physique est distante et en lecture seule, de la répliquer ou encore de la migrer. Après plusieurs expérimentations, les auteurs sont arrivés à conclure que les performances des programmes peuvent êtres améliorées en utilisant des stratégies de réplication et de migration, mais pas autant que des programmes pré-optimisés. Ils n'ont pas trouvé qu'une seule politique puisse être considérée comme la meilleure pour les applications testées.

Utiliser une méthode d'allocation de page physique à la demande ou "first-touch" au niveau noyau et en faire usage depuis le mode utilisateur tout en se basant sur des informations de trace (fournies par le matériel, mais accessible également en mode utilisateur) a été exploré par [20]. Dans cette méthode, une application est pré-exécutée afin d'analyser les références mémoire et d'établir les affinités des pages de mémoire. L'application est exécutée par la suite, le noyau est alors guidé en provoquant des défauts de pages afin d'associer les pages virtuelles à des pages physiques adéquates.

Au lieu de chercher à établir la localité d'une manière dynamique, d'autres études montrent l'importance de la prise en compte de la topologie de l'architecture afin de déduire la localité des accès mémoire et de l'exploiter lors de l'allocation des ressources matérielles mémoire ou processeur [21]. D'après cette étude, ces considérations prennent d'autant plus d'importance que les configurations NUMA sont grandes.

Les recherches dans le domaine des systèmes d'exploitation n'ont pas cessé d'évoluer depuis les premières machines NUMA. Ainsi plusieurs projets de recherches récents existent tels que : le système K42 [22] qui est un système à base de micro-noyau, Barrelfish [7] qui introduit la notion de multi-kernel ou encore Corey [23] qui permet à l'utilisateur de contrôler le placement mémoire de certaines structures du noyau. D. Wentzlaff et A. Agrawal ont suivis une approche de conception par micro-noyau et avancent le concept Factored Operating Systems (Fos) [24] qui se fonde sur le partitionnement spatial des ressources matérielles processeurs entre le système et les applications utilisateur plutôt que les partager temporellement (timeslicing). Ces projets de recherches apportent différents points de vue mais expriment la nécessité d'apporter des réponses nouvelles pour les actuelles et futures architectures many-cores.

Ces différentes approches de conception ont un message commun, la conception traditionnelle, monolithique, des noyaux n'est pas scalable. Cependant, la question reste ouverte puisque des travaux plus récents menés par S. Boyd-Wickizer, M Frans Kaashoek et al. [9], ceux-là mêmes qui ont soutenu la thèse de l'exo-noyau dans Corey [23] contre le concept du noyau monolithique, et ce dès 16 cœurs, ont démontré la scalabilité du noyau Linux sur une machine PC à 48 cœurs, en éliminant certains goulots d'étranglement du noyau.

L'approche de conception monolithique distribuée, mise en œuvre dans le noyau d'ALMOS, est différente de celle d'un mulit-kernel, micro-noyau ou exo-noyau. En effet ALMOS est un noyau monolithique en mémoire partagée. Nous partageons le même constat que K42 [22] portant sur l'importance de la localité pour le passage à l'échelle et l'organisation en objets distribués. L'approche de conception du noyau d'ALMOS est une évolution possible pour les noyaux monolithiques traditionnels tels que Linux et BSD.

Concernant la stratégie "migration au second accès", nous proposons comme dans le système Solaris (MADV_ACCESS_LWP), un drapeau supplémentaire pour informer le noyau de cette stratégie par l'appel madvise. H. Lof et al. [25] ont étudié l'efficacité de cette stratégie. B. Goglin et N. Furmento [26] ont proposé son intégration dans Linux. Nous avons proposé notre solution, au niveau noyau, "Auto-Next-Touch" permettant de remédier aux limitations de cette stratégie toute en la rendant automatique et transparente pour les applications utilisateur.

6. Conclusion

Cet article présente le système d'exploitation ALMOS et montre comment répartir les structures de données internes du noyau pour passer à l'échelle en nombre de cœurs. Plus généralement, il cherche à montrer que le modèle des threads s'exécutant en mémoire partagée avec un protocole de cohérence cc-NUMA semble capable de passer à l'échelle. Nos premiers résultats sont très encourageants puisque nous obtenons une accélération linéaire jusqu'à 256 processeurs pour la barrière de synchronisation du noyau, jusqu'à 192 pour l'application EPFilter, et une accélération linéaire jusqu'à 64 cœurs pour l'application SPLASH2/FFT. De nouvelles mesures sur des configurations à 768 cœurs sont planifiées. Enfin, nous avons souligné le besoin d'un mécanisme décentralisé permettant au noyau une gestion adaptée des ressources matérielles lors du placement des tâches, de l'allocation mémoire ou de l'équilibrage de charge des processeurs. En s'appuyant sur une solution, que nous avons déjà spécifiée, nous allons doter le noyau d'ALMOS d'un tel mécanisme puis le valider expérimentalement.

Bibliographie

1. V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. – Clock rate versus IPC : The end of the road for conventional microarchitectures. – In Proceedings of the International Symposium on Computer Architecture, p. 248-259, June 2000
2. Tera-scale Computing Research Program – http ://techresearch.intel.com/articles/Tera-Scale/1421.htm
3. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K.Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, T. Mattson. – A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS, – Proceedings of the International Solid-State Circuits Conference, Feb 2010
4. Timothy G. Mattson , Rob Van der Wijngaart , Michael Frumkin. – Programming the Intel 80-core network-on-a-chip terascale processor, – Proceedings of the 2008 ACM/IEEE conference on Supercomputing, Austin, Texas, November 15-21, 2008
5. TILE-Gx™processors family – http ://www.tilera.com/products/TILE-Gx.php
6. Shekhar Borkar, – Thousand core chips : a technology perspective. – Proceedings of the 44th annual conference on Design automation, San Diego, California, June 04-08, 2007
7. Andrew Baumann , Paul Barham , Pierre-Evariste Dagand , Tim Harris , Rebecca Isaacs , Simon Peter , Timothy Roscoe , Adrian Schüpbach , Akhilesh Singhania. – The multikernel : a new OS architecture for scalable multicore systems, – Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, October 11-14, 2009

8. Edmund B. Nightingale , Orion Hodson , Ross McIlroy , Chris Hawblitzel , Galen Hunt. – Helios : heterogeneous multiprocessing with satellite kernels, – Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana, USA, October 11-14, 2009
9. Boyd-Wickizer, S., Clements, A., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R., and Zeldovich, N. – An Analysis of Linux Scalability to Many Cores, – in USENIX OSDI 2010
10. Tera-Scale ARchitecture – https ://www-asim.lip6.fr/trac/tsar/wiki
11. I. Miro-Panades, A. Greiner, and A. Sheibanyrad. – A low cost Network-on-Chip with guaranteed service well suited to the GALS approach, – in IEEE 1st Int. Conf. on Nano-Networks, 2006
12. Stanford parallel applications for shared memory (SPLASH-2). – http ://www-flash.stanford.edu/apps/SPLASH/.
13. Woo, S. C., Singh, J. P., and Hennessy, J. L. – The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. – Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), October 1994
14. Karsten Schwan and Anita K. Jones. – Specifying resource allocation for the cm* multiprocessor, – IEEE Software, 3(3) :60-70, May 1984
15. R. J. Swan , S. H. Fuller , D. P. Siewiorek, Cm*. – a modular, multi-microprocessor, – Proceedings of the June 13-16, 1977, national computer conference, Dallas, Texas, June 13-16, 1977
16. Rohit Chandra , Scott Devine , Ben Verghese , Anoop Gupta , Mendel Rosenblum. – Scheduling and page migration for multiprocessor compute servers, – Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, San Jose, California, United States, p. 12-24, October 05-07, 1994
17. Vijayaraghavan Soundararajan , Mark Heinrich , Ben Verghese , Kourosh Gharachorloo , Anoop Gupta , John Hennessy. – Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors, – Proceedings of the 25th annual international symposium on Computer architecture, Barcelona, Spain, p. 342-355, June 27-July 02, 1998
18. William J. Bolosky , Michael L. Scott , Robert P. Fitzgerald , Robert J. Fowler , Alan L. Cox, – NUMA policies and their relation to memory architecture. – Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, Santa Clara, California, United States, p. 212-221, April 08-11, 1991
19. Jaydeep Marathe , Frank Mueller. – Hardware profile-guided automatic page placement for ccNUMA systems, – Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, New York, USA, March 29-31, 2006
20. Timothy Brecht – On the importance of parallel application placement in NUMA multiprocessors, – USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems, San Diego, California, p. 1-1, September 22-23, 1993
21. J. Appavoo, D. Da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. – Experience distributing objects in an SMMP OS. – ACM Transactions on Computer Systems, 25(3), 2007
22. S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. – Corey : An operating system for many cores. – In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, p. 43-57, 2008
23. D. Wentzla and A. Agarwal. – Factored operating systems (fos) : The case for a scalable operating system for multicores. – Operating Systems Review, 43(2), Apr. 2009
24. Henrik Löf , Sverker Holmgren. – Affinity-on-next-touch : increasing the performance of an industrial PDE solver on a cc-NUMA system, – Proceedings of the 19th annual international conference on Supercomputing, Cambridge, Massachusetts, June 20-22, 2005.
25. Brice Goglin , Nathalie Furmento. – Enabling high-performance memory migration for multithreaded applications on LINUX, – Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing, p. 1-9, May 23-29, 2009.
26. N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. – Faults in Linux : Ten Years Later. – In 16th International Conference on Architectural Support for Programming Languag and Operating Systems (ASPLOS), p. 305-318, Newport Beach, CA, USA, Mar. 2011.
27. E. Grochowski and M. Annavaram. – Energy per instruction trends in Intel microprocessors. – Technology@Intel Magazine, vol. 4, (no. 3), pp.1-8, 2006
28. John L. Henesy and David A. Patterson – Computer Architecture : A Quantitative Approach – Morgan Kaufmann Publishers, 2007. – ISBN 1-55860-329-8.