THÈSE DE DOCTORAT DE L'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité Informatique

École Doctorale Informatique, Télécommunication et Électronique

Présenté par

Ghassan ALMALESS

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core

Soutenue le 27 février 2014

devant le jury composé de :

M. Guy Gogniat, Professeur, Lab-STICC, Lorient	Rapporteur
M. Frédéric Pétrot, Professeur, TIMA, Grenoble	Rapporteur
M. Marc Shapiro, Directeur de Recherche, INRIA-LIP6, Paris	Examinateur
M. Renaud Sirdey, Directeur de Recherche CEA, Saclay	Examinateur
M. Alain Greiner, Professeur, LIP6, Paris	Directeur de thèse
M. Franck Wajsburt, Maître de Conférence, LIP6, Paris	Co-Encadrant de thèse

Table des matières

À ma femme, ma fille, mes parents et mon cher oncle.

Résumé

De nos jours, des processeurs à mémoire partagée cohérente ayant jusqu'à 100 cores intégrés sur la même puce sont une réalité et des processeurs many-cores ayant plusieurs centaines, voire, un millier de cores sont à prévoir prochainement. Dans ces architectures, la question de la localité du trafic lié aux miss de caches L1 (data, instruction et TLB) est primordiale à la fois pour passer à l'échelle et pour réduire la consommation électrique (énergie consommée par bit transféré). Notre thèse est que : (i) la gestion de la localité des accès mémoire doit être prise en compte au niveau du noyau du système d'exploitation et elle doit être effectuée d'une manière transparente aux applications utilisateur; et (ii) les noyaux monolithiques actuels sont incapables de renforcer la localité des accès mémoire des threads d'une même application parallèle, car la notion de threads dans ces noyaux est intrinsèquement inadaptée pour les processeurs many-cores. Par conséquent, nous pensons que la démarche suivie jusqu'à présent pour faire évoluer les noyaux monolithiques n'est pas suffisante et qu'il est impératif de mettre la question de la localité des accès mémoire au centre de cette évolution.

Pour prouver notre thèse, nous avons conçu et réalisé ALMOS (Advanced Locality Management Operating System), un système d'exploitation expérimental à base de noyau monolithique distribué. ALMOS dispose d'un nouveau concept de thread, nommé *Processus Hybrides*. Il permet à son noyau de renforcer, d'une manière transparente, la localité des accès mémoire liés à l'exécution de chaque thread. La gestion des ressources (cores et mémoires physiques) dans le noyau d'ALMOS est distribuée renforçant la localité des accès mémoire lors de la réalisation des services systèmes. La prise de décision concernant l'allocation mémoire, le placement des tâches et l'équilibrage de charge dans le noyau d'ALMOS est décentralisée, multi-critères et sans prise de verrou. Elle repose sur une infrastructure distribuée coordonnant d'une manière scalable l'accès aux ressources.

En utilisant le prototype virtuel précis au cycle et au bit près du processeur many-core TSAR, nous avons expérimentalement démontré que : (i) les performances (scalabilité et temps d'exécution) du schéma d'ordonnancement distribué du noyau d'ALMOS sur 256 cores dépassent celles des noyaux monolithiques existants; (ii) la réalisation répartie de l'appel système *fork* permet de passer à l'échelle ce service système sur 512 cores; (iii) le coût de la mise à jour de l'infrastructure distribué de prise de décisions du noyau d'ALMOS ne nécessite que 0.05% de la puissance de calcul totale du processeur TSAR; (iv) les performances (scalabilité, temps d'exécution et trafic distant) de la stratégie d'affinité mémoire du noyau d'ALMOS, nommé *Auto-Next-Touch*, dépassent celles des deux stratégies *First-Touch* et *Interleave* sur 64 cores; (v) le modèle de processus hybrides d'ALMOS permet de passer à l'échelle deux applications hautement multi-threads existantes sur 256 cores et une troisième sur 1024 cores; et enfin (vi) le couple ALMOS/TSAR (64 cores) offre systématiquement une bien meilleure scalabilité que le couple Linux/AMD (Interlagos 64 cores) pour 8 applications de classe HPC et traitement d'images.

Abstract

Nowadays, single-chip cache-coherent many-core processors having up to 100 cores are a reality. Many-cores with hundreds or even a thousand of cores are planned in the near future. In theses architectures, the question of the locality of L1 cache-miss related traffic (data, instruction and TLB) is essential for both scalability and power consumption (energy by moved bit). Our thesis is that: (i) handling the locality of memory accesses should be done at kernel level of an operating system in a transparent manner to user applications; and (ii) the current monolithic kernels are not able to enforce the locality of memory accesses of multi-threaded applications, because the concept of thread in these kernels is inherently unsuitable for many-core processors. Therefore, we believe that the evolution approach of monolithic kernels undertaken until now is insufficient and it is imperative to put the question of the locality of memory accesses in the heart of this evolution.

To prove our thesis, we designed and implemented ALMOS (Advanced Locality Management Operating System), an experimental operating system based on a distributed monolithic kernel. ALMOS has a new concept of thread, called *Hybrid Process*. It allows its kernel to enforce the locality of memory accesses of each thread. The resources (cores and physical memory) management in ALMOS's kernel is distributed enforcing the locality of memory accesses when performing system services. Decision making regarding memory allocation, tasks placement and load balancing in ALMOS's kernel is decentralized, multi-criteria and without locking. It is based on a distributed infrastructure coordinating, in a scalable manner, the accesses to resources.

Using the cycle accurate and bit accurate virtual prototype of TSAR many-core processor, we experimentally demonstrated that: (i) performance (scalability and execution time) on 256 cores of the distributed scheduling scheme of ALMOS's kernel outperform those of the shared scheduling scheme found in existing monolithic kernels; (ii) distributed realization of the *fork* system call enables this system service to scale on 512 cores; (iii) updating the distrusted decision-making infrastructure of ALMOS's kernel costs just 0.05 % of the total computing power of TSAR processor; (iv) performance (scalability, execution time and remote traffic) of memory affinity strategy of ALMOS's kernel, called *Auto-Next-Touch*, outperform those of two existing strategies *First-Touch* and *Interleave* on 64 cores; (v) concept of *Hybrid Process* of ALMOS's kernel scales up two existing highly multi-threads applications on 256 cores and a third one on 1024 cores; and finally (vi) the couple ALMOS/TSAR (64 cores) gives systematically much better scalability than the couple Linux/AMD (Interlagos 64 cores) for 8 multi-threads applications belonging to HPC and image processing domains.

Remerciements

Je tiens à remercier tout d'abord mon directeur de thèse, Alain Greiner, de m'avoir embarqué à bord du projet TSAR, de m'avoir laissé la liberté de mener mes travaux de recherches, d'avoir adopté et soutenu le projet ALMOS, pour sa relecture méticuleuse de ce manuscrit, pour sa disponibilité quand j'avais besoin de son avis/aide et pour toutes les discussions que j'ai pu avoir avec lui et qui m'ont fait progressé tout au long de cette thèse.

J'exprime toute ma gratitude envers mon encadrant de thèse, Franck Wajsburt. Je le remercie pour m'avoir donné l'opportunité de travailler sur la thématique des systèmes d'exploitation dès mon Master (M1 et M2). Je le remercie pour sa confiance et son soutien tout au long de cette thèse, mais surtout dans les moments les plus difficiles où son écoute, ses conseils et ses encouragements étaient ci-précieux. J'ai tellement apprécié de travailler avec lui (enseignements, présentations, séminaires, conférences, projets, etc.). Je le remercie d'avoir soutenu, accompagné et adopté le projet ALMOS dès le début.

Je remercie également les membres du jury. Messieurs Guy Gogniat, professeur au laboratoire Lab-STICC, et Frédéric Pétrot, professeur au laboratoire TIMA, d'avoir accepté de rapporter sur mes travaux de thèse. Messieurs Marc Shapiro, directeur de recherche à l'INRIA-LIP6 et Renaud Sirdey directeur de recherche au CEA-LIST d'avoir accepté d’examiner ma soutenance de thèse.

Je souhaite remercier Pierre Sens et Isabelle Demeure d'avoir accepté d'être le jury de ma soutenance à mi-parcours et pour leur retour constructif. Les excellents cours en systèmes d'exploitation (licence et M1) de Pierre Sens m'ont passionné et m'ont fait aimer le domaine.

Je tiens à remercie les personnes suivantes : Marie-Minerve Louërat pour son soutien et son encouragement. Pirouz Bazargan-Sabet pour les différentes discussions très enrichissantes et pour notre travail de réflexion sur un mécanisme LL/SC scalable (j'espère que le dépôt de brevet en cours aboutira). Emmanuelle Encrenaz-Tiphene, Karine Heydemann, François Pêcheux, Habib Mehrez, Hassan Aboushady et Quentin Meunier pour leurs gentillesses, conseils et encouragements. Abdelmalek Si-Merabet, Jean-Paul Chaput et Manuel Bouyer pour les discussions enrichissantes. Patricia Renaud et Gérard Nowak pour m'avoir donné l'opportunité d’enseigner au niveau Licence. Olivier Marin, Gaël Thomas, Gilles Muller pour leurs encouragements. Nam Nguyen et Fabien Colas-Bigey pour l’intérêt qui ont porté pour mes travaux de recherches et pour leurs conseils précieux sur la préparation de l'après-thèse. Je remercie également tous les doctorants, ingénieurs et partenaires avec qui j'ai travaillé et échangé dans le cadre des projets ALMOS, TSAR, SHARP et Tsunamy.

À ce stade d’aboutissement de tout un projet vivement voulu depuis maintenant plus de 12 ans, j'ai une pensée bien particulière à mes parents dont l'amour, l'encouragement et le soutien ont été sans limites tout au long de mon parcours. Je tiens à remercier vivement mon oncle Pierre Joseph pour tout ce qu'il a fait pour moi. Sans lui, ce projet n'aurait fort probablement jamais vu le jour.

Enfin, merci infiniment à ma femme Zakia pour être à mes côtés et pour m'avoir supporté et soutenu pendant mes longues années d'études. Le gazouillement de notre petite fille, Mathilde, confirme bien qu'une nouvelle vie commence.

Chapitre 1

Introduction

Depuis une dizaine d'années, les fabricants des processeurs haute-performance ont atteint des limitations physiques concernant la dissipation thermique et l'augmentation de la fréquence d'horloge [1]. Ils se sont tournés vers les architectures multi-cores à mémoire partagée cohérente garantie par matériel. De nos jours, des processeurs à mémoire partagée cohérente ayant jusqu'à 100 cores intégrés sur la même puce sont une réalité [17,14] et des processeurs many-cores ayant plusieurs centaines, voire, un millier de cores sont à prévoir prochainement [40].

Dans les architectures many-cores à mémoire partagée cohérente, les cores sont typiquement regroupés dans des tuiles/clusters et le cache du dernier niveau est physiquement distribué sur ces derniers. Les tuiles/clusters sont reliés entre-eux par un NoC (Network-On-Chip) [104,13,8,106]. Par conséquent, la latence des accès mémoire liés aux miss des caches L1 est variable selon la distance séparant le core demandeur du tuile/cluster contenant le cache L2 responsable des lignes demandées. Dans ces architectures, la question de la localité du trafic généré par les miss de caches L1 (data, instruction et TLB) est primordiale à la fois pour passer à l'échelle et pour réduire la consommation électrique (énergie consommée par bit transféré). Notre thèse est que : (i) la gestion de la localité des accès mémoire doit être prise en compte au niveau du noyau de système d'exploitation et elle doit être effectuée d'une manière transparente aux applications utilisateur; et (ii) les noyaux monolithiques sont incapables de renforcer la localité des accès mémoire des threads d'une même application parallèle, car la notion de threads dans ces noyaux est intrinsèquement inadaptée pour les processeurs many-cores cc-NUMA intégrés sur puce. Par conséquent, nous pensons que la démarche suivie jusqu'à présent pour faire évoluer les noyaux monolithiques n'est pas suffisante et qu'il est impératif de mettre la question de la localité des accès mémoire au centre de cette évolution.

Pour prouver notre thèse, nous avons conçu et réalisé ALMOS (Advanced Locality Management Operating System), un système d'exploitation expérimental à base de noyau monolithique distribué en mémoire partagée [7]. ALMOS dispose d'un nouveau concept de thread, nommé *Processus Hybride*. Ce concept réunie la protection fournie par les processus et la facilité de programmation en mémoire partagée fournie par les threads. Tout en étant compatible avec le standard PThreads, il permet au noyau d'ALMOS de : (i) contrôler finement et d'une manière transparente aux applications utilisateur le placement des pages physiques par thread; (ii) renforcer la localité des accès mémoire (miss de caches data, instruction et TLB) liés à l'exécution d'un thread; et (iii) restaurer la localité des accès mémoire d'un thread après sa migration. La gestion des ressources (cores et mémoires physiques) dans le noyau d'ALMOS est distribuée renforçant la localité des accès mémoire lors de la réalisation des services systèmes. La prise de décision concernant l'allocation mémoire, le placement des tâches et l'équilibrage de charge dans le noyau d'ALMOS est décentralisée, multi-critères et sans prise de verrou. Cette prise de décision repose sur une infrastructure distribuée permettant de coordonner d'une manière scalable l'accès aux ressources.

En utilisant le prototype virtuel précis au cycle et au bit près du processeur many-core TSAR (Tera-Scale ARchitecture) [8,9], les évaluations expérimentales ont montré que : (i) le schéma

d'ordonnancement distribué client-serveur du noyau d'ALMOS dépasse, en termes de scalabilité et de temps d'exécution, le schéma d'ordonnancement partagé des noyaux monolithiques existants tels que Linux sur 256 cores; (ii) la réalisation répartie de l'appel système *fork* permet de passer à l'échelle ce service système sur 512 cores; (iii) le coût de la mise à jour de l'infrastructure distribuée de prise de décisions du noyau d'ALMOS ne nécessite que 0.05% de la puissance de calcul totale du processeur TSAR; (iv) la stratégie d'affinité mémoire automatique du noyau d'ALMOS, nommé *Auto-Next-Touch* dépasse en termes de scalabilité, de temps d'exécution et de quantité de trafic distant, les performances des deux stratégies *First-Touch* et *Interleave* sur 64 cores; (v) le modèle de processus hybrides d'ALMOS permet de passer à l'échelle deux applications hautement multi-threads existantes sur 256 cores et une troisième sur 1024 cores, tout en réduisant la quantité du trafic distant des trois applications; et (vi) le couple ALMOS/TSAR (64 cores) offre systématiquement une bien meilleure scalabilité que le couple Linux/AMD (Opteron Interlagos 64 cores) pour les 8 mêmes applications multi-threads non modifiées de classes HPC et traitement d'images.

1.1 Contributions

Les principales contributions de cette thèse sont les suivantes :

- La conception, la réalisation et l'évaluation expérimentale d'ALMOS (Advanced Locality Management Operating System), un système d'exploitation expérimental permettant de passer à l'échelle des applications hautement multi-threads existantes sur un processeur many-cores à mémoire partagée cohérente.

- Le concept de *Processus Hybride*, une redéfinition de la notion de thread telle qu'elle existe dans les noyaux monolithiques. Il permet de doter un noyau monolithique d'une abstraction nécessaire afin de contrôler finement les accès mémoire avec la granularité de thread.

- Le concept de *Réplicas Noyau* permettant à un noyau monolithique de répliquer son code et ses informations de traduction d'une manière portable et sans assistance matérielle particulière. Combiné avec le concept de *Processus Hybride*, il permet de renforcer la localité des accès mémoire des threads liés aux miss de caches instruction et TLB lors de l'exécution des services systèmes.

- La stratégie d'affinité mémoire automatique *Auto-Next-Touch*, une stratégie d'allocation et de migration transparente de pages physiques de données. Elle permet au noyau de remédier aux problèmes de la stratégie *First-Touch* appliquée par défaut par les noyaux monolithiques existants tels que Linux et le noyau de Solaris.

- Une organisation distribuée permettant de décentraliser la gestion des ressources (cores et mémoires physiques) et de renforcer la localité des accès mémoire lors de cette gestion. En particulier, elle comporte : (i) un schéma d'ordonnancement distribué client-serveur permettant de passer à l'échelle les primitives d'ordonnancement; et (ii) une infrastructure et une méthodologie permettant de répartir la réalisation des services systèmes afin de renforcer la localité de leurs accès mémoire.

- L'infrastructure distribuée *DQDT (Distributed Quaternary Decision Tree)* permettant une prise de décision décentralisée, multi-critères et sans verrou lors de l'allocation des ressources distantes. Cette prise de décision intervient lors de l'allocation mémoire, le placement d'une tâche ou encore lors de l'équilibrage de charge.

1.2 Plan du manuscrit

Dans le chapitre 2, nous introduisant les problématiques traitées dans cette thèse. Ce chapitre est composé de 5 sections. La section 2.1 présente l'émergence des architectures cc-NUMA many-cores intégrées sur puce. La section 2.2 présente notre première problématique concernant le renforcement de la localité des accès mémoire. La section 2.3 présente notre deuxième problématique concernant l'architecture scalable de l'ordonnanceur. La section 2.4 présente notre troisième problématique concernant la gestion coordonnée des ressources cores et mémoires. Enfin, la section 2.5 présente les 5 questions pour lesquelles nous apportons des réponses dans cette thèse.

Dans le chapitre 3, nous présentons notre synthèse de l'état de l'art concernant deux éléments : la scalabilité des systèmes d'exploitation et la localité des accès mémoire des tâches. Ce chapitre est composé de trois sections. La section 3.1 présente une analyse des différents travaux portant sur les approches traditionnelles et alternatives visant à concevoir un système d'exploitation scalable. La section 3.2 présente une analyse de différents travaux concernant la prise en compte de la localité des accès mémoire à trois niveaux différents : utilisateur, noyau du système d'exploitation et matériel. Enfin, les conclusions du chapitre sont présentées dans la section 3.3.

Dans le chapitre 4, nous présentons les principes généraux sur lesquels repose le système d'exploitation expérimental ALMOS. Ce chapitre est composé de 6 sections. La section 4.1 présente une vue globale d'ALMOS. La section 4.2 présente la gestion décentralisée des ressources dans le noyau d'ALMOS. La section 4.3 présente l'infrastructure distribuée DQDT. La section 4.4 présente la stratégie d'affinité mémoire automatique Auto-Next-Touch. La section 4.5 présente les réplicas noyau et les processus hybrides. Enfin, les conclusions du chapitre sont présentées dans la section 4.6.

Dans le chapitre 5, nous présentons les évaluations expérimentales d'ALMOS. Ce chapitre est composé de 4 sections. La section 5.1 présente les questions auxquelles les expérimentations menées dans ce chapitre doivent fournir des réponses. La section 5.2 présente le dispositif d'expérimentation composé de deux éléments : (i) le prototype virtuel précis au cycle et au bit près du processeur many-core TSAR, et (ii) les 8 applications hautement multi-threads qui relèvent du domaine du HPC et du domaines de traitement d'images. La section 5.3 présente les expérimentations menées et l'analyse de leurs résultats. Enfin, les conclusions du chapitre sont présentées dans la section 5.4.

Enfin, nous présentons nos conclusions et perspectives dans le chapitre 6. Ce chapitre est composé de deux sections. La première présente nos réponses aux questions posées dans le chapitre 2 portant sur nos trois problématiques : (i) le renforcement de la localité des accès mémoire; (ii) l'architecture scalable de l'ordonnanceur; et (iii) la gestion coordonnée des ressources cores et mémoires. La deuxième section présente les perspectives que nous voyons pour nos travaux de recherche menés dans le cadre de cette thèse.

Chapitre 2

Problématique

Dans ce chapitre, nous commençons par présenter les architectures cc-NUMA many-cores intégrées sur puce. Ensuite nous introduisons un certain nombre de problématiques que nous traitons dans cette thèse, qui porte sur la conception d'un noyau en mémoire partagée pour des architectures cc-NUMA many-cores. Nos questions de recherche sont posées dans la conclusion de chaque problématique. Nous terminons ce chapitre par une conclusion globale qui regroupe l'ensemble de ces questions pour lesquelles nous apportons des réponses dans cette thèse.

2.1 Émergence des architectures cc-NUMA many-cores

Augmenter le parallélisme de tâche nécessite, principalement, l'augmentation du nombre de processeurs. Dans une architecture multiprocesseurs classique dite SMP (Symetric Multi-Processors) les processeurs accèdent à la mémoire par un bus unique partagé. La figure 2.1 illustre cette architecture. Dans ce type d'architecture, le bus devient rapidement un goulot d'étranglement quand le nombre de processeurs augmente. Pour continuer à délivrer plus de performances, les fabricants de ces processeurs ont disposé principalement de deux leviers. Le premier est la possibilité d'augmenter la fréquence d'horloge et de cadencer ainsi les processeurs toujours plus vite. Le deuxième est un budget en nombre de transistors qui double chaque 18 mois (loi de Moore) permettant ainsi de proposer des micro-architectures toujours plus sophistiqués (p. ex : prédicteur de branchement, unité d'exécution supersclaire, exécution dans le désordre, etc) permettant d'exploiter le parallélisme d'instruction d'un unique fil d'exécution.

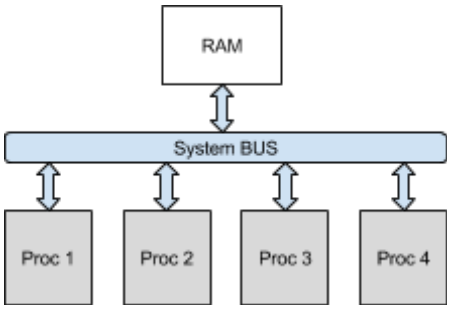


Figure 2.1 : Une architecture multiprocesseurs SMP à 4 processeurs.
L'accès à la mémoire se fait à travers d'un bus unique partagé.

La cohérence des caches dans ces architectures est assurée par le matériel en utilisant des techniques d'espionnage du bus partagé. Puisque la latence d'accès à la mémoire est uniforme pour tous les processeurs, la principale contrainte pour les programmeurs d'applications destinées à s'exécuter sur une architecture SMP est la parallélisation de leur code et la gestion de la synchronisation inter-tâches. Pour le système d'exploitation, les principales contraintes sont : (i) prendre en compte l'existence de plusieurs fils d'exécution dans un processus utilisateur afin de pouvoir les répartir sur les processeurs et obtenir ainsi du parallélisme réel; et (ii) préserver le parallélisme des tâches exprimé par les applications utilisateurs en permettant l'existence simultanée de plusieurs fils d'exécution dans le noyau. Afin de tirer le meilleur profit de la capacité de calcul offerte par ces architectures SMP, les applications, les compilateurs et les systèmes d'exploitation doivent prendre en compte un

certain nombre d'aspects tels que la hiérarchie des caches (taille, niveau de partage et alignement), le faux partage d'une ligne de cache, l'affinité d'une tâche logicielle à un processeur et à son cache lors de l'équilibrage de charge entre les processeurs. D'autres aspects concernant la micro-architecture du processeur sont également à prendre en compte, tels que la présence d'un prédicteur de branchement, le profondeur et le nombre de pipelines d'exécution, la présence des unités de calculs spécifiques (flottant, vectoriel, etc).

Depuis une dizaine d'années, les fabricants des processeurs haute-performance ont atteint des limitations physiques concernant la dissipation thermique et l'augmentation de la fréquence d'horloge [1]. Alors que la loi de Moore semble tenir sa promesse et devant ces limitations physiques, les architectes de ces processeurs haute-performance ont été contraint à augmenter le nombre de processeurs afin de continuer à accroître d'avantage les performances. Limitées par leur goulot d'étranglement, qui est le bus partagé, les architectures multiprocesseurs classiques SMP ont été abandonnées en faveur des architectures multi-cores où à l'intérieur même d'un processeur, plusieurs cœurs de calcul avec leurs caches, appelés cores, sont intégrés sur la même puce et peuvent partager certaines ressources matérielles. Un exemple de ces architectures multi-cores est Intel Core i7 [15] et AMD Opteron [16]. Dans l'architecture AMD Opteron, chaque core dispose de son propre cache L1 de données ayant une stratégie d'écriture Write-Through. Chaque deux cores partagent une FPU et un cache L1 d'instructions.

Dans une architecture multi-core cc-NUMA (cache-coherent Non-Uniform Memory Access), les cores sont regroupés dans des nœuds, chaque nœud dispose d'un accès direct à un module mémoire local. Les nœuds sont connectés entre-eux par un mécanisme d'interconnexion global permettant d'avoir un espace d'adressage physique partagé entre les cores (n'importe quel core de n'importe quel nœud peut accéder à tous les modules mémoire). La latence d'accès à ces modules mémoire est variable et dépend de la distance séparant un core du module mémoire cible, d'où le caractère NUMA de l'architecture. La cohérence des caches dans ces architectures est assurée par un protocole de cohérence matériel basé sur la notion de répertoire distribué d'où le terme "cc" (cache-coherent) dans l'acronyme cc-NUMA. La figure 2.2 illustre une architecture AMD Opteron à 48-cores cc-NUMA.

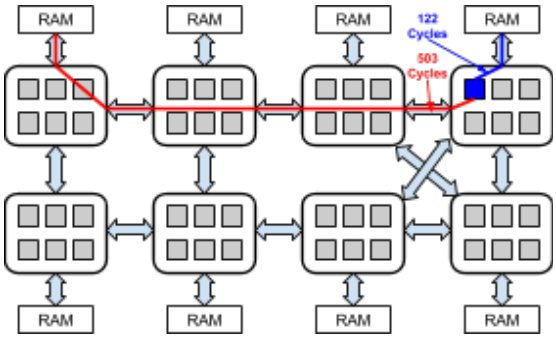


Figure 2.2 : AMD Opteron à 48-cores cc-NUMA : 8 nœuds (puces) à 6-cores chacun. Un core (e.g. en blue) accède à son module mémoire local en 122 cycles tandis que sa latence d'accès au module mémoire le plus éloigné est de 503 cycles.

L'augmentation constante de la densité d'intégration permet aujourd'hui d'avoir jusqu'à 100 cores sur la même puce d'où le terme processeur many-core. Un exemple industriel de ces processeurs many-cores à mémoire partagée cohérente, est Intel MIC (Many Integrated Cores) [17] à 50 cores et Tilera Tile-Gx à 100 cores [14]. Pour sa part, Tilera table sur un doublement de nombre de cores dans ses processeurs chaque les deux ans [19] alors que des many-cores intégrant jusqu'à 1000 cores sont à prévoir prochainement [40].

Étant donné le très grand nombre de cores dans une architecture many-core et pour optimiser le rendement énergétique (nombre d'instructions exécutées par Watt consommé) les cores peuvent être plus simples (simple scalaire, exécution dans l'ordre) avec des petits caches. Les interconnects reliant les nœuds entre eux sont des réseaux (Network-On-chip) à commutation de paquets intégré sur puce. Ce qui signifie que l'espace d'adressage physique est partagé entre tous les cores avec une latence variable d'accès à la mémoire selon le nombre de routeurs (hop) séparant un core du contrôleur mémoire cible. Comme dans les architectures SMP, la cohérence des caches continue à être assurée par des protocoles matériels basés sur la notion d'un répertoire distribué. Dans les architectures proposées par Tilera comme Tile64 [13] et Tile-Gx [18], le NoC est un mesh 2D avec une stratégie d'acheminement des paquets de type Wormehole et une stratégie de routage X-First. La stratégie d'écriture des caches L1 est Write-Through. La figure 2.3 illustre cette architecture.

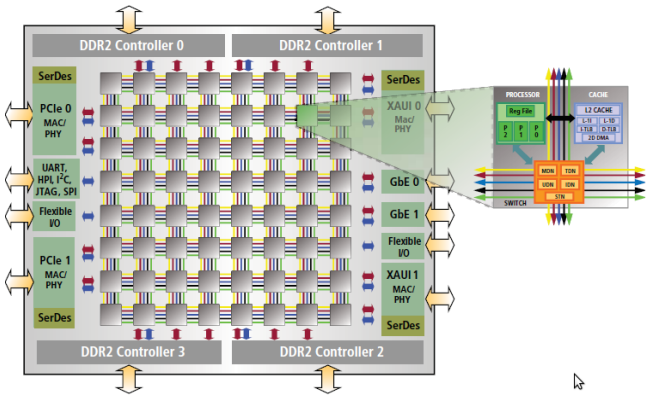


Figure 2.3 : Tile64 : 64 cores intégrés sur puce avec 64 caches L2 (LLC) distriubés, un Mesh 2D, 4 contrôleurs mémoire DDR2 et contrôleurs de périphériques. (Source : site web de Tilera).

2.1.1 Conclusion

Les architectures cc-NUMA à plusieurs dizaines, voire 100 cores, sont une réalité aujourd'hui. Le nombre de cores est appelé à doubler chaque deux ans. L'approche cc-NUMA permet de maintenir le modèle de programmation en mémoire partagée cohérente ce qui assure à l'industrie informatique une transition douce depuis les architectures dite UMA (Uniform Memory Access) comme le SMP à condition que les couches logicielles et en particulier le système d'exploitation, s'adaptent à la caractéristique fortement NUMA des architectures many-cores. Les contraintes liées à l'exploitation efficace des architectures SMP continueront à exister dans les architectures cc-NUMA, car un nœud cc-NUMA est un multiprocesseurs SMP. En revanche, la latence variable de l'accès à la mémoire physique distribuée fait de la problématique de la localité des accès mémoire des tâches logicielles, un problème primordial à prendre en compte sous peine de subir une dégradation sévère des performances et de la consommation énergétique. Le grand nombre de cores et de modules mémoire imposent une gestion scalable de ces deux ressources, telle que le placement des tâches, l'allocation mémoire et l'équilibrage de charge. Afin de tirer le meilleur profit d'une architecture cc-NUMA, d'autres problématiques sont aussi à prendre en compte telles que l'accès aux entrées sorties, l'accès aux systèmes de fichiers et la pile réseau. Dans cette thèse nous allons nous focaliser sur deux aspects : la localité des accès mémoire et la gestion scalable des cores et des mémoires physiques. Nous allons aborder ces deux problématiques de point de vue de la conception d'un système d'exploitation optimisé pour les architectures many-cores cc-NUMA.

2.2 Renforcement de la localité des accès mémoire

Dans une architecture cc-NUMA, l'unité de base impliquée dans un accès mémoire fait par un core est la ligne de cache. Le caractère NUMA signifie qu'une ligne de cache est associée à un seul et unique LLC (Last Level Cache). Ainsi, quand le cache de premier niveau (cache L1) d'un core fait un miss, une requête de lecture est envoyée au LLC gestionnaire de cette ligne. La latence de cet accès dépend du facteur NUMA de l'architecture (le coût d'accès distant divisé par le coût d'accès local). Plus la distance entre le core et le LLC cible est grande, plus la latence est importante. Ce qui implique une dégradation en performances d'une part, en terme de temps d'exécution d'une tâche et d'autre part, en terme de consommation énergétique (énergie consommée par bit déplacé) qui est un élément important dans une architecture many-core intégrée sur puce.

Le temps total d'exécution d'une tâche peut être décrit comme suit : $T_{exec} = T_{cpu} + T_{mem}$ où T_{cpu} est le temps utile passé en exécution effective par un core tandis que T_{mem} présente le temps perdu par ce core lorsqu'il est bloqué en attendant l'accomplissement de ses requêtes d'accès mémoire. T_{mem} peut être décrit comme suit : $T_{mem} = T_{Inst} + T_{Data} + T_{TLB}$. La première composante représente le temps perdu en attendant l'accomplissement des accès mémoire liés aux miss du cache L1 instruction. La deuxième composante représente le temps perdu en attendant l'accomplissement des accès mémoire liés aux miss, écritures et accès atomiques du cache L1 data. La dernière composante représente le temps perdu en attendant l'accomplissement des accès mémoire liés aux miss TLB (Translation Lookaside Buffer). Par conséquent, les latences des accès mémoires dépendent du placement du code, des données et les informations de traduction des adresses virtuelles vers des adresses physiques (les tables de pages). Cela est valable pour les tâches d'une application utilisateur comme pour celles du noyau du système d'exploitation.

Renforcer la localité des accès mémoire c'est faire en sorte que les accès mémoire effectués par une tâche qui s'exécute sur un core restent, en majeure partie, vers le contrôleur mémoire local. Malgré l'effort constant pour adapter les systèmes d'exploitation en mémoire partagée dits monolithiques[1] aux architectures cc-NUMA, tel que la prise en compte de la topologie matérielle afin de déterminer le voisinage d'un core et la mise en place d'un allocateur de pages physiques par nœud, le rapprochement des objets mémoire d'une tâche du core exécutant son code est rarement pris en charge par le noyau lui-même. En revanche, ces noyaux proposent des APIs (Application Programming Interface) afin de permettre aux programmeurs d'applications parallèles de contrôler, par eux-mêmes, l'affinité des tâches vis-à-vis des cores et des bancs mémoire physique. Ces APIs, dites NUMA, sont non standards (non portable) et dépendent fortement du noyau sous-jacent. Un exemple de ces APIs : la *libnuma* [20] dans le cas de Linux et la *Locality Group API* [21] dans le cas de Solaris.

Nous pensons que c'est le système d'exploitation, et plus précisément, son noyau, qui doit prendre en charge l'aspect NUMA de l'architecture matérielle d'une manière transparente aux applications utilisateurs, pour deux raisons : la première est d'assurer une compatibilité avec les applications parallèles préexistantes sans pour autant sacrifier les performances. Ceci est un élément indispensable afin de faciliter l'adoption des many-cores cc-NUMA en s'appuyant sur un modèle de programmation en mémoire partagée exprimé par des langages de programmation et des interfaces standards et répandus. La seconde raison est que, dans un environnement généraliste multi-applications et dynamique, les décisions prises par chaque application expriment un besoin individuel qui peut être en contradiction avec celles prises

1 Dans un noyau monolithique, les services systèmes sont réalisés au niveau de privilège noyau en opposition avec d'autres approches tels que les micro-noyaux où les services systèmes sont réalisés par des serveurs systèmes s'exécutant en mode utilisateur (communication par passage de messages).

par les autres applications. Le noyau étant chargé d'allouer les ressources aux applications et étant capable de les gérer d'une manière transparente, des décisions plus globales peuvent être prises à ce niveau en prenant en comptes les contraintes et les demandes dynamiques de l'ensemble des applications.

Dans les noyaux monolithiques en mémoire partagée tels que Linux, BSD et Solaris, le contrôle du placement des données d'une tâche peut se faire à l'aide du mécanisme de mémoire virtuelle. La politique proposée par défaut est le "First-Touch". Selon cette politique, quand une tâche accède pour la première fois à une page dans l'espace virtuel de son processus, un défaut de page (une exception matérielle) se déclenche. En réponse à ce défaut de page, le noyau alloue une page physique appartenant au nœud local au core sur lequel la tâche s'exécute. En cas de pénurie de pages physiques libres localement, le noyau réalise l'allocation depuis le premier nœud ayant une page physique libre se trouvant à proximité. La notion de voisinage d'un nœud est propre pour chaque noyau et elle est représentée d'une manière différente d'un noyau à un autre.

Bien que la stratégie "First-Touch" favorise la localité des accès mémoire d'une tâche, elle a l'inconvénient majeur de ne pas tenir compte de la phase séquentielle d'initialisation se trouvant dans la plupart des applications parallèles. En effet, une application parallèle commence très souvent par une phase séquentielle dans laquelle un certain nombre d'allocations mémoire sont effectuées. Les structures de données ainsi allouée sont alors initialisées par une seule tâche (e.g. génération des valeurs ou lecture des données depuis le disque). Par conséquent, le noyau alloue sur le même nœud toutes les pages physiques nécessaire pour ces données. Une fois la phase d'initialisation terminée, plusieurs tâches sont créées pour traiter parallèlement ces données. Pour les tâches qui seront placés sur les cores du même nœud que celui du core exécutant la phase d'initialisation, les accès aux données sont locales. En revanche, toutes les autres tâches accéderont d'une manière distante à ces données ce qui génère un trafic distant subissant le facteur NUMA de l'architecture. Plus il y a de tâches exécutées par des cores différents, plus les performances se dégradent car (i) la distance par rapport au nœud ayant les données initiales augmente; et (ii) il y a plus de contention : sur l'interconnect, sur le cache ayant une copie des lignes accédées et sur le contrôleur du banc mémoire contenant les lignes qui ne sont pas présentes dans ce cache. Par conséquent, cette politique ne permet pas le passage à l'échelle dans une architecture cc-NUMA.

D'autres politiques d'affinité mémoire sont proposées via les APIs NUMA permettant à un programmeur d'associer, à une région virtuelle, un certain choix concernant la provenance (les nœuds) des pages physiques (p. ex : entrelacement, round-roubin, etc) [20]. Même si ce contrôle fin, non transparent et non standard, est donné aux programmeurs des applications, cela reste insuffisant. En effet, les tâches subissent les décisions d'équilibrage de charge prises par le noyau afin d'améliorer le temps de réponse global du système en transférant l'exécution d'une tâche depuis un core chargé vers un core oisif ou moins chargé. Ce transfert peut impliquer un changement de nœud. Ces décisions d'équilibrage de charge impactent la localité des accès mémoire d'une tâche, car ce qui était local avant la migration ne l'est plus après.

Finalement, il faut noter qu'aucun noyau monolithique ne traite la localité des accès mémoire lors d'un miss TLB. Historiquement, ces noyaux ont favorisé le partage du code entre les applications afin d'économiser l'empreinte en mémoire physique et ils continuent de le faire. Ceci est en contradiction avec la disponibilité de ressources mémoire physiques abondantes (plusieurs dizaines voire centaines de gigaoctets) et la caractéristique NUMA des

architectures multi/many-cores d'aujourd'hui. La localité des accès mémoire au code et aux tables de pages doit être prise en compte.

2.2.1 Conclusion

Permettre au système d'exploitation de renforcer la localité des accès mémoire dans une architecture many-core cc-NUMA est une problématique primordiale à résoudre, car elle impacte à la fois les performances en terme de temps d'exécution, de scalabilité, et de consommation énergétique (énergie consommée par bit transféré). Lors de l'exécution d'une tâche, les accès mémoire effectués par le core sur lequel elle s'exécute concernent les données, le code et les tables de pages. Nos questions concernant le renforcement de la localité des accès mémoire sont les suivantes :

- Comment un noyau monolithique en mémoire partagée peut-il renforcer la localité des accès mémoire d'une manière transparente aux applications ?

- Comment un noyau monolithique en mémoire partagée peut-il restaurer la localité d'une tâche après une décision d'équilibrage de charges impliquant la migration d'une tâche d'un nœud vers un autre ?

2.3 Architecture scalable de l'ordonnanceur

L'ordonnancement des tâches est une fonction principale d'un noyau qui permet de décider, entres autres, quelle tâche parmi celles prêtes à être exécutées, doit être élue pour gagner un core et pour combien du temps. L'entité qui réalise cette fonction d'ordonnancement est appelé l'ordonnanceur. Selon les politiques d'ordonnancement fournies par l'ordonnanceur, et pour des raisons de performance, cet ensemble de tâches est réalisé en utilisant une structure de données sous forme de listes chaînées ou d'un arbre de priorité. Ces structures de données sont appelés d'une manière générale les files d'attente de l'ordonnanceur (run queue). Afin de garantir la consistance de ces structures, un mécanisme de synchronisation par verrou est utilisé dans les noyaux monolithiques existants tels que Linux, BSD et Solaris.

Les noyaux monolithiques d'aujourd'hui mettent généralement en place un ordonnanceur par core. Cette approche a les avantages suivants : (i) réduire la contention sur les files d'attentes; (ii) réduire la complexité de certain algorithmes de gestion de ces files d'attentes comme l'élection, le calcul de priorité ou le calcul de charges, car le nombre de tâches par core est généralement limité[2]; et (iii) assurer une meilleure localité des caches de premier niveau d'un core. Dans cette approche, la présence du verrou est nécessaire, car l'état d'un ordonnanceur quelconque peut être modifié potentiellement par tous les cores de l'architecture many-core. L'opération "réveiller une tâche" (wakeup) appelée lors qu'une tâche veut réveiller une autre, est en conflit avec l'opération "mise en attente passive de la tâche courante" (sleep) ou encore "céder le core courant" (yield), car elle est potentiellement exécutée par n'importe quel core. D'autres opérations sont également sérialisées par le verrou, car elles manipules également l'état de l'ordonnanceur telles que "attacher une nouvelle tâche", "détacher une tâche" et "élire une tâche".

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 rajouter cette dernière dans la bonne file d'attente après avoir recalculé sa priorité. Cette opération est coûteuse. Elle est payée par la tâche *A* qui peut-être sur un autre core que le core cible, celui de la tâche *B*. Par conséquent, elle pénalise également toute

2 Par ses politiques d'équilibrage de charges, le noyau veille à ce que le nombre de tâches affectées à un core reste limité.

opération d'ordonnancement sur le core de la tâche *B* puisqu'elle nécessite une prise de verrou de son ordonnanceur. Dans une architecture cc-NUMA, les accès distants pour mettre à jours les structures de données de l'ordonnanceur cible lors de la réalisation de l'opération "réveiller une tâche" subissent le facteur NUMA de l'architecture. Plus le core cible est loin plus les latences d'accès sont grandes, moins bonnes sont les performances. Plus le nombre de tâches à réveiller est grand, plus cette perte en performances l'est également. Un autre effet, est la pollution du cache du core qui exécute l'opération de réveil avec des lignes de caches appartenant aux nœuds distants. Ces lignes de caches inutiles, en terme de localité temporelle, seront invalidées par le protocole de cohérence aux prochaines écritures effectuées lors de l'exécution de la prochaine primitive d'ordonnancement par les cores cibles. Ce qui représente un temps perdu pour ces cores à cause de ce trafic de cohérence.

Enfin, La présence de verrou complique le code de certaines opérations impliquant un accès exclusif à plusieurs ordonnanceurs (e.g lors de l'équilibrage de charges), car elles nécessitent une gestion plus fine de l'ordre de prise des verrous pour éviter un inter-blockage (deadlock).

2.3.1 Conclusion

L'ordonnancement est une fonction fondamentale dans un noyau. Les primitives d'ordonnancement sont les briques de base utilisées par les sous-systèmes d'un noyau afin de réaliser un certain nombre de services de synchronisation inter-tâches (e.g. les barrières, les variables de conditions, etc) et pour gérer l'accès aux ressources systèmes. Par conséquent, les performances et la scalabilité de la fonction d'ordonnancement sont des questions importantes. La réalisation de la fonction d'ordonnancement dans les noyaux monolithique d'aujourd'hui nécessite l'utilisation de verrous et ne tient pas compte de l'aspect cc-NUMA de l'architecture.

Notre question concernant l'ordonnancement est la suivante :

- Comment doter un noyau monolithique d'une fonction d'ordonnancement scalable et adaptée aux architectures many-cores cc-NUMA ?

2.4 Gestion coordonnée des ressources cores et mémoire

La gestion des ressources cores et mémoire physique s'appuie sur deux fonctions principales d'un noyau monolithique : l'ordonnancement des tâches et l'allocation de mémoire physique. La gestion des ressources cores et mémoire physique incluent la création d'un processus, la création d'une tâche, le placement[3] d'une tâche, la migration d'une tâche et l'équilibrage de charges entre les cores. La fonction ordonnancement est réalisée par les ordonnanceurs tandis que la fonction d'allocation mémoire est réalisée par les allocateurs de mémoire physique.

La prise en compte de la topologie et de l'aspect NUMA, a été introduite dans les noyaux monolithiques mais elle présente un certain nombre de limitations quand le nombre de core passe de plusieurs dizaines à plusieurs centaines. En effet, cette prise en compte a été introduite dans l'ordonnanceur et dans les fonctions d'allocation mémoire d'une manière indépendante en utilisant des structures de données séparées. Cette séparation ne permet pas au noyau une prise de décision multi-critères en prenant en compte à la fois la localité, la disponibilité des cores et la disponibilité mémoire (en terme de pages physiques). D'autre part, les algorithmes d'équilibrage de charges ont besoin d'accéder à tous les ordonnanceurs afin de déterminer leurs charges exprimées en nombre de tâches prêtes à être exécutée. D'une

3 Généralement, la notion de placement des tâches et celle de l'équilibrage de charges sont attribuées à la fonction d'ordonnancement dans un noyau monolithique.

manière semblable, quand une requête d'allocation de pages physiques ne peut pas être satisfaite sur le nœud local du demandeur, l'allocateur de pages physiques de ce nœud, doit tenter l'allocation depuis les nœuds voisins[4] en visitant, successivement, tous les nœuds en cas d'échec. Dans les deux cas, l'accès aux structures de données distantes des nœuds (ordonnanceur par core et allocateur mémoire par nœud) implique des accès distants coûteux et un trafic de cohérence afin de copier/invalider les lignes accédées. Ces lignes distantes polluent le cache L1 (voire même le L2/L3 selon les architectures) du core exécutant l'accès à ces structures distantes. Accéder aux structures de données de chaque core, dans une architecture many-core cc-NUMA à plusieurs centaines de cores, n'est pas scalable.

Dans les noyaux monolithiques actuels, les services de gestion des tâches et processus tels que la création d'un processus, la création et la migration d'une tâche, restent encore inconscients au facteur NUMA de l'architecture. Prenons l'exemple de Linux où lors de la création d'une tâche, les allocations mémoire concernant les structures de données noyau (p. ex : descripteur de tâche) sont allouées depuis le nœud local *"A"* du core qui exécute le service. Or, la tâche nouvellement créée est potentiellement placée sur un autre core dans un autre nœud *"B"*. Tous les accès effectués par cette dernière à ses propres structures noyau (p. ex : lors de la réalisation d'un service demandé par un appel système) vont systématiquement référer au nœud distant *"A"*, là où ces structures ont été allouées (le nœud du créateur de la tâche). Si une tâche dans un nœud donné crée N nouvelles tâches placées sur d'autre cores que le sien, il y aura un trafic (N vers 1) d'accès distant vers le nœud de la tâche créatrice lorsque ces tâches demandent des services systèmes. Par conséquent, ce trafic favorise une contention sur le cache LLC du nœud *"A"*. Cette situation se retrouve dans la plupart des applications parallèles où l'application commence par une phase séquentielle d'initialisation qui se termine par la création de N tâches dites tâches de traitement avant de démarrer la phase du calcul parallèle.

2.4.1 Conclusion

La gestion des ressources cores et mémoire est au centre d'un noyau monolithique en mémoire partagée. L'allocation des pages physiques et objets mémoires du noyau, la création des processus, la création des tâches, le placement des tâches, et l'équilibrage de charge entre les cores constituent le cœur même du noyau. Au delà de la problématique de synchronisation lors de l'exécution de ces services, la localité des accès mémoire est un enjeu primordial pour la performance et pour la scalabilité sur une architecture many-core à plusieurs centaines de cores. Nos questions concernant la gestion des cores et mémoire sont les suivantes :

- Comment doter un noyau monolithique en mémoire partagée d'un mécanisme décentralisé passant à l'échelle permettant d'une part, d'unifier la représentation de la localité pour tous les services et sous-systèmes du noyau et d'autre part, d'effectuer une prise de décision multi-critères en prenant en compte à la fois la localité, la disponibilité des cores et la disponibilité des pages physiques ?

- Comment renforcer la localité des accès mémoire lors de la réalisation des services noyau tels que la création d'un processus, la création d'une tâche et la migration d'une tâche ?

4 Selon un certain ordre qui représente le voisinage d'un nœud sachant que cette notion de voisinage et son implémentation en terme de structures de données sont propres à chaque noyau.

2.5 Conclusions

Dans ce chapitre nous avons abordé un certain nombre de problématiques concernant la conception d'un noyau en mémoire partagé scalable pour une architecture many-core cc-NUMA et le renforcement de la localité des accès mémoire des tâches. L'ensemble des questions auxquelles nous apportons des réponses dans cette thèse sont les suivantes :

Renforcement de la localité des accès mémoire

- Comment un noyau monolithique en mémoire partagée peut-il renforcer la localité des accès mémoire d'une manière transparente aux applications ?

- Comment un noyau monolithique en mémoire partagée peut-il restaurer la localité d'une tâche après une décision d'équilibrage de charges impliquant un changement de nœud cc-NUMA ?

Architecture scalable d'un ordonnanceur

- Comment doter un noyau monolithique d'une fonction d'ordonnancement scalable et adaptée aux architectures many-cores cc-NUMA ?

Gestion coordonnée des ressources cores et mémoire

- Comment doter un noyau monolithique en mémoire partagée d'un mécanisme décentralisé passant à l'échelle permettant d'une part, d'unifier la représentation de la localité pour tous les services et sous-systèmes du noyau et d'autre part, d'effectuer une prise de décision multi-critères en prenant en compte à la fois la localité, la disponibilité des cores et la disponibilité des pages physiques ?

- Comment renforcer la localité des accès mémoire lors de la réalisation des services noyau tels que la création d'un processus, la création d'une tâche et la migration d'une tâche ?

Chapitre 3

État de l’art

Dans ce chapitre nous allons présenter notre synthèse de l’état de l’art concernant deux éléments : la scalabilité des systèmes d’exploitation et la localité des accès mémoire des tâches. Ce chapitre est composé de deux sections. La première, section 3.1, présente une analyse de différents travaux portant sur les approches traditionnelles et alternatives visant à concevoir un système d’exploitation scalable. La deuxième, section 3.2, présente une analyse de différents travaux concernant la prise en compte de la localité des accès mémoire à trois niveaux différents : utilisateur, noyau du système d’exploitation et matériel. Pour chaque section, une conclusion est présentée avant de terminer le chapitre par une conclusion globale.

3.1 Scalabilité des systèmes d’exploitations

Les recherches sur la scalabilité des systèmes d’exploitation n’ont pas cessé d’évoluer depuis les premières machines NUMA. Ainsi, parallèlement à l’évolution constantes des systèmes d’exploitation monolithiques, plusieurs projets de recherche ont proposé des approches alternatives permettant de mieux passer à l’échelle les systèmes d’exploitation. Dans cette section, nous présentons ces différentes approches alternatives, ainsi que l’évolution de la scalabilité dans le contexte de l’approche monolithique traditionnelle.

3.1.1 Hierarchical Clustering

Parmi les premiers travaux sur la scalabilité des systèmes d’exploitation se trouve la thèse d’Unrau [22] (1993) à l’université de Toronto. Le contexte de ses travaux a été l’apparition des machines NUMA telles que : NYU Ultra-Comuter [23], IBM RP3 [24], BBN Butterfly [25] et plus spécialement, Hector [26] (développée par la même université). Unrau a formulé trois grands principes pour une conception scalable d’un système d’exploitation qui sont : préserver le parallélisme, borner le surcoût des services et préserver la localité. Unrau et al. [27] (1995) ont proposé le système d’exploitation Hurricane à base de micro-noyau avec une approche de conception nommée *Hierarchical Clustering*.

Selon cette approche, le système d’exploitation est vu en tant qu’une collection de clusters. Un cluster est un système d’exploitation complet à base de micro-noyau prenant en charge un faible nombre de processeurs. Il existe donc autant d’instances du système d’exploitation que de clusters. Les serveurs systèmes (en mode utilisateur) de chaque cluster coopèrent entre-eux pour donner aux applications une vision globale d’un seul système d’exploitation. Étant donné que chaque cluster propose l’ensemble des services système via des serveurs dédiés, l’existence de plusieurs clusters constitue une réplication des mêmes services et augmente la disponibilité du système. La dimension (en nombre de processeurs) d’un cluster est un compromis entre le coût des accès intra-cluster et inter-clusters. La figure 3.1 illustre l’organisation de plusieurs clusters pour constituer un système distribué fortement couplé.

Dans Hurricane, chaque cluster dispose de son propre ordonnanceur qui assure localement l’équilibrage de charge et l'ordonnancement des processus, tandis que l’équilibrage de charge inter-clusters est assuré par un ordonnanceur global. Les communications dans Hurricane peuvent être réalisées par l’envoi de messages ou par l’utilisation de Protected Procedure Calls (PPC) [28]. Un envoi de message est réalisé par l’interruption du cluster cible pour : (i)

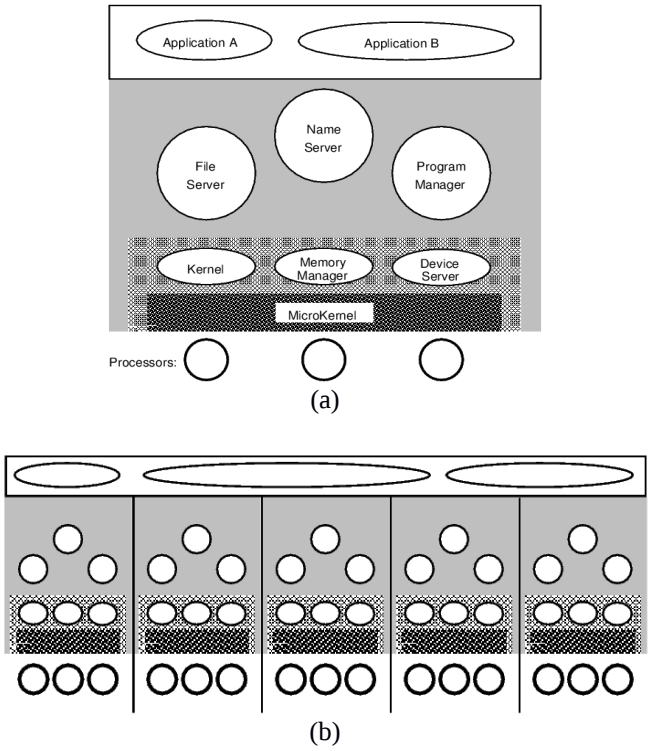


Figure 3.1 : (a) un système à un cluster, (b) un système à plusieurs clusters. Les services systèmes en mode utilisateur coopèrent pour fournir l'image d'un unique système aux applications. Source : Unrau et al. [27].

l'allocation d'un tampon mémoire pour réceptionner le message, (ii) copier le message et les informations de contrôle depuis le processus émetteur du cluster source, et (iii) attacher le tampon à la file d'attente du processus destinataire du message. PPC est un moyen de communication entre un client et un serveur qui s'exécutent sur le même processeur. L'idée est que le serveur se comporte comme un objet passif (un conteneur de ressource et un domaine de protection). Quant un client demande un service, il change son espace d'adresse pour celui du serveur et invoque un thread de traitement associé à sa requête avant de retrouver son espace d'adressage initial.

Nous partageons les trois principes d'Unrau concernant la conception d'un système d'exploitation scalable. En revanche, en préconisant la coordination d'un ensemble d'instances d'OS, l'approche Hierarchical Clustering est une approche gros-grain et ne résout pas les problèmes de scalabilité et de localité des accès mémoire au niveau de chaque OS. Les différents travaux présentant Hurricane ne décrivent pas les mécanismes de coordination des différentes instances d'OS permettant de donner l'image d'un seul système pour l'utilisateur. D'autre part, les moyens de communication (passage de message et PPC) entre les différentes entités (serveurs systèmes, applications utilisateur, micro-noyaux des différents instances) sont complexes et coûteux comparés aux moyens de communication dans le cas d'un unique système d'exploitation à base de noyau monolithique; ayant donc une seul instance. Dans l'approche monolithique, les communications entre les serveurs systèmes peuvent se faire directement en mémoire partagée puisqu'ils s'exécutent au même niveau de privilège (celui du noyau); tandis que les communications entre les clients (en mode utilisateur) et les serveurs systèmes passent tout simplement à travers de l'interface des appels systèmes du noyau ou encore à travers de la mémoire partagée (sans appel système) comme dans la proposition de Soares et Stumm [98] (2010). Enfin, les évaluations présentées par Unrau et al. [27] ont été menées sur une machine NUMA sans cohérence de caches par matériel ayant seulement 16 processeurs. Ces évaluations ont examiné la question du

dimensionnement des clusters (en terme de nombre de processeurs attribués à un cluster) dans le but d'augmenter le nombre de processus séquentiels exécutés par l'unité de temps. Par conséquent, il nous est impossible, d'extrapoler ces résultats pour ce qui concerne : (i) la scalabilité du système dans le contexte des applications multi-threads, et (ii) la scalabilité du système dans le contexte de processeurs many-core à mémoire partagée cohérente.

3.1.2 Clustered Objects

L'approche *Hierarchical Clustering*, en faisant coopérer plusieurs instances d'un système d'exploitation, pose le problème du dimensionnement des clusters, accroît la complexité du système et ne renforce pas la localité des accès mémoire à l'intérieur même d'un cluster [29]. Les travaux sur l'approche *Hierarchical Clustering* du système d'exploitation Hurricane ont cependant formé une base sur laquelle le système d'exploitation Tornado a été proposé par Gamsa et al. [30,29] (1999) de l'université de Toronto en coopération avec IBM. Tornado comme son successeur à IBM, le système d'exploitation K42 [31] (2006), ont été conçus avec une approche nommée *Clustered Objects* [32,29,33,34]. Le contexte du développement du système Tornado/K42 est l'apparition des machines cc-NUMA telles que : DASH [35], FLASH [36], SGI Origin [37] et plus spécialement, NUMAchine, développée à l'université de Toronto [38]. Tornado/K42 a été complètement écrit en C++.

L'approche *Clustered Object* permet à un objet d'être décomposé en plusieurs objets appelés *Representatives* qui fournissent localement les services de l'objet. Cette décomposition est transparente de point de vue extérieur à l'objet; si une tâche invoque une méthode d'un objet clusterisé, c'est la méthode de l'objet représentant local au processeur de la tâche qui est invoquée. Le représentant local est déterminé grâce à des tables de translation : chaque processeur dispose d'une table local de translation lui permettant de traduire l'identifiant global de l'objet clusterisé pour obtenir (ou créer le cas échéant) l'identifiant de l'objet représentant local à ce processeur comme l'illustre la figure 3.2. La réalisation d'un service par un représentant peut nécessiter une coopération entre les différents représentants d'un objet clusterisé, mais cette coopération relève de l'implémentation et n'est pas exposée à l'extérieur de l'objet. L'état d'un objet clusterisé peut être partagé ou répliqué dans les différents représentants.

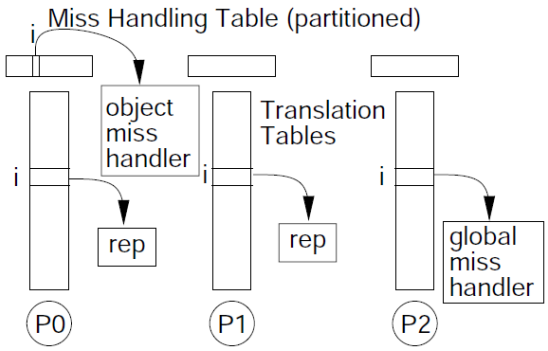


Figure 3.2 : un aperçu de l'implémentation des objets clusterisés. P0, P1 et P2 représentent des processeurs dont P0 et P1 ont déjà un objet représentant local, tandis que P2 n'en dispose pas encore. Source : Gamsa et al. [29].

Tornado/K42 est un système d'exploitation à base de micro-noyau dont les clients et les serveurs systèmes (en mode utilisateur) communiquent en utilisant les PPC (Protected Procedure Calls) [28]. L'ordonnancement est effectué en deux temps : le micro-noyau d'abord sélectionne quel *Dispatcher* (un ordonnanceur en mode utilisateur) doit être chargé sur un processeur; puis le *Dispatcher* décide quel thread utilisateur doit être exécuté sur ce

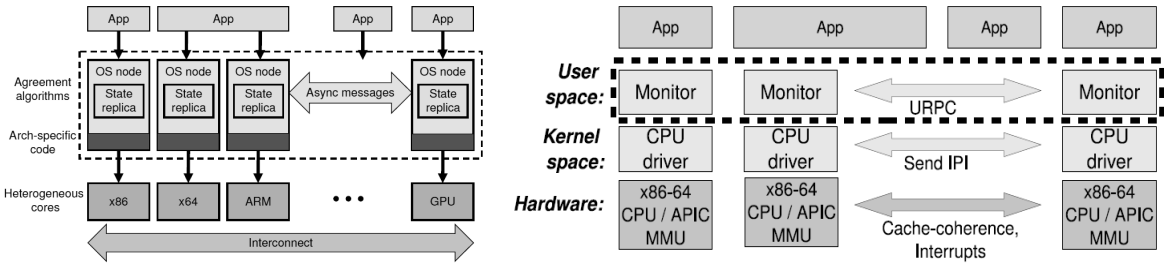
processeur. Le micro-noyau et les *Dispatchers* des processus utilisateurs coopèrent grâce à des activations (scheduler activations) [39].

L'approche Clustered Objects constitue une infrastructure permettant d'encapsuler la mise en œuvre répartie d'un service. Nous trouvons cette approche intéressante et nous partageons l'objectif visé : renforcer la localité des accès mémoire et la scalabilité du système en essayant de fournir le service demandé localement à la tâche qui le demande. En revanche, nous pensons que cette infrastructure n'est pas indispensable afin de répartir les services d'un noyau. Un noyau peut être, par construction, organisé d'une manière distribué sans avoir recours au dynamisme fourni par l'infrastructure Clustered Objets. D'autre part, l'approche Clustered Objects ne traite pas la question de la coordination des différents représentants d'un objet clusterisé, car selon elle cette coordination est à la charge du programmeur de l'objet clusterisé. En particulier, la gestion coordonnée des ressources en ce qui concerne l'allocation mémoire et l'équilibrage de charges n'a pas été abordé dans les différents travaux cités plus haut. L'ordonnancement en deux temps (l'élection d'abord d'un Dispatcher, puis l'élection par le Dispatcher d'un thread utilisateur) est complexe à mettre en œuvre. Un tel schéma d'ordonnancement *M:N* a été longuement proposé par Solaris et a été abandonné dès la version 9 du système pour des raisons de performances [99] (2002). Le schéma adopté par la suite est un schéma plus simple *1:1* selon lequel chaque thread utilisateur est visible et ordonnancé directement par le noyau; c'est le cas également dans Linux et dans FreeBSD (depuis la version 8 en 2009). Tornado/K42 est à base de micro-noyau. Les moyens de communication entre les clients et les serveurs systèmes (en mode utilisateur) ont l'inconvénient, comme dans le cas de Hurricane, d'être complexes et coûteux comparés aux moyens de communication dans le cas d'un noyau monolithique.

Les évaluations de Tornado/K42 présentées par Appavoo et al. [34] ont été menées sur une machine IBM *pSeries-680* cc-NUMA ayant seulement 24 processeurs. En utilisant de bnechsmarks, la métrique de performances utilisé est le nombre de processus indépendants exécutés par l'unité du temps. La comparaison de performances a été effectuée contre Linux (version 2.4) qui, dans cette version, souffre d'un manque flagrant de parallélisme au niveau noyau (p. ex : l'usage d'un verrou global dans le noyau nommé BKL, l'utilisation d'un ordonnanceur global partagé par l'ensemble de processeurs, etc.). Étant donné le faible nombre de processeurs utilisés, la nature des évaluations menées et le type de machines utilisée (cc-NUMA multiprocesseurs); il nous est difficile d'extrapoler les résultats pour en conclure quant à la scalabilité de Tornado/k42 dans le contexte de l'exécution des applications multi-threads et de l'utilisation de processeurs many-cores.

3.1.3 Multi-Kernels

Avec l'apparition des processeurs multi-cores tels que le SCC d'Intel à 48 cores [41], l'équipe système à l'ETH Zurich et Microsoft ont proposé le système d'exploitation Barrelfish [42] (2009) pour répondre principalement à deux enjeux : le passage à l'échelle sur des processeurs many-cores avec ou sans cohérence de caches assurée par le matériel et la prise en charge de l'hétérogénéité entre les différentes ressources de calculs dans une machine (p.ex : GPU et CPU). Baumann et al [42] ont avancé l'approche multi-kernels pour concevoir Barrelfish, qui repose sur trois principes : (i) rendre explicite les communications inter-cores, (ii) rendre la structure de l'OS neutre vis à vis de l'architecture matérielle, et (iii) répliquer tout état au lieu de le partager. La figure 3.3 illustre le concept multi-kernels et donne un aperçu de la structure de Barrelfish sur une cible x86-64 multi-cores.



(a) (b)

Figure 3.3 : (a) le concept multi-kernels, (b) un aperçu de la structure de Barrelfish. Source : Baumann et al [42].

Selon l'approche multi-kernels, l'environnement d'exécution des applications est une collection de nœuds où chaque nœud est constitué d'un core exécutant un OS à base d'un micro-noyau. Les communications entre les différentes OS se font par passage de messages : Barrelfish utilise une variante des Remote Procedure Calls au niveau utilisateur (URPC) [43]. chaque OS dispose d'un serveur système en mode utilisateur appelé *Monitor*. L'ensemble des moniteurs coordonnent collectivement l'accès à l'état global du système et maintient la cohérence des structures de données répliquées par core (les tables d'allocation mémoire et les mappings des espaces d'adressages virtuels). La notion d'un processus dans Barrelfish est différente de celle d'un processus dans un noyau monolithique. Un processus est représenté sous formes d'un ensemble de *Dispatchers* avec un *Dispatcher* par core où le processus peut être exécuté. Sur un core les *Dispatchers* sont ordonnancés par un ordonnanceur, au niveau du micro-noyau de ce core, appelé *CPU Driver*. Ce dernier communique avec les *Dispatchers* (en mode utilisateur) à travers des *up-calls* en adoptant un fonctionnement similaire à ce qui a été fait auparavant dans Psyche [44] et aux mécanismes des activations (scheduler activations) [39].

Concernant les évaluations de benchmarks, Baumann et al. [42] ont évalué 3 applications multi-threads de la suite NAS OpenMP et 2 applications de la suite SPLASH-2. La cible est une machine AMD à 16 cores exécutant Barrelfish ou Linux (version 2.6.26). Les résultats ont montrée des limitations de scalabilité comparables sur Barrelfish comme sur Linux.

Des travaux de recherches similaires visant à repenser le système d'exploitation sous forme d'une collection d'OSs communicants par passage de messages ont été menés par Wentzlaff et Agarwal avec FOS (Factored Operating System) [45] (2009). L'objectif du projet FOS au MIT [46,47] est de concevoir un système à la fois pour processeurs many-cores et pour les infrastructures de Cloud Computing. Comme dans le cas de Barrelfish, chaque core exécute une instance de l'OS complet à base de micro-noyau comme il est illustré par la figure 3.4.

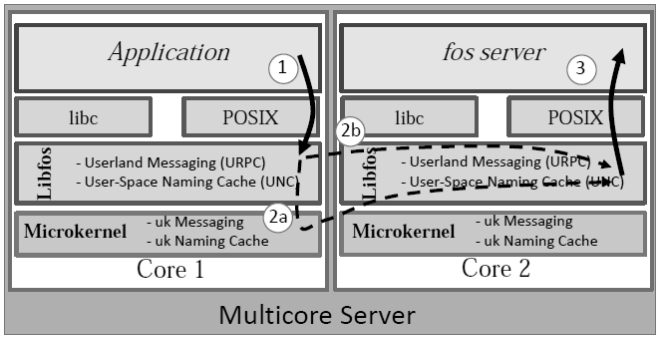


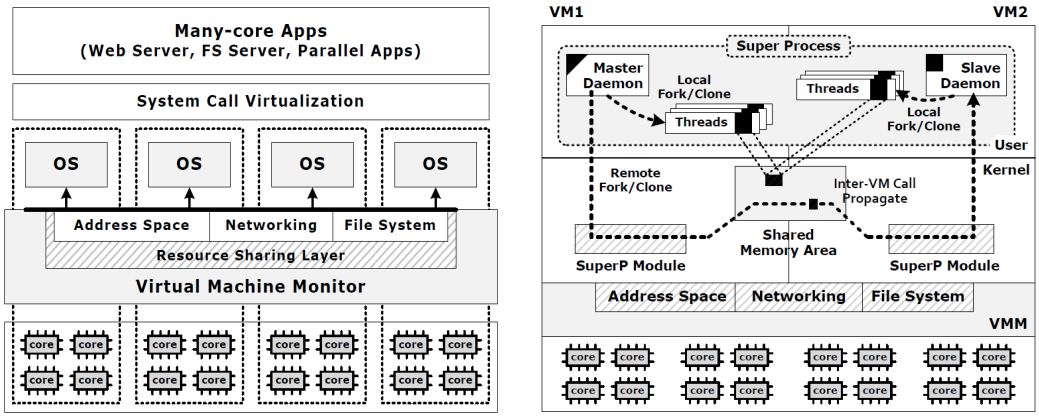
Figure 3.4 : Un aperçu de la structure de FOS. Chaque core exécute un OS à base de micro-noyau. Les différents OSs communiquent par passage de messages. Source : Wentzlaff et al [48].

En se fondant sur le partitionnement spatial des ressources de calcul, certaines instances de l'OS complet exécutent des serveurs systèmes, tandis que d'autres exécutent les applications. À la différence de Barrelfish, FOS se focalise sur la parallélisation des services systèmes pour faire face à une demande massive et variable de services. Dans FOS, un service système est rendu par un *fleet* de serveurs spatialement distribués. Ces serveurs peuvent s'exécuter sur des machines différentes ou sur une seule machine. Ces serveurs coopèrent entre-eux pour réaliser le services et leur nombre peut s'adapter à la fréquence des demandes [46]. Concernant les évaluations expérimentales, les principales publications autour du projet FOS [45,46] n'incluent pas des évaluations de scalabilité en utilisant des benchmarks.

L'approche multi-kernel est fondamentalement différente de l'approche monolithique. La première approche fait l'hypothèse de l'absence de la cohérence matérielle des caches et l'utilisation exclusive d'un paradigme de programmation par passage de messages. Alors que la deuxième approche fait l'hypothèse inverse concernant la cohérence de caches et elle repose sur le paradigme de programmation en mémoire partagée. L'approche multi-kernel ne permet donc pas l'exécution des applications parallèles existantes, développées pour un seul système d'exploitation prenant en charge une seule machine.

3.1.4 OS Clustering

En s'inscrivant dans le débat sur la scalabilité des systèmes d'exploitation pour architectures many-cores et en s'appuyant sur des techniques de virtualisation; Song et al. de l'université de Fudan ont proposé le système Cerberus [49] (2011). Cerberus est structuré selon une approche nommée *OS Clustering* dont le but est de passer à l'échelle les systèmes d'exploitations traditionnels (*Unix-like*) sur des architectures many-cores. En coordonnant l'exécution de plusieurs machines virtuelles dont chacune exécute une instance du même OS (Linux), Cerberus étend le fonctionnement d'un VMM (Virtual Machine Monitor) pour donner aux applications utilisateurs la vision d'être exécutées par un unique système d'exploitation; comme l'illustre la figure 3.5 (a).



(a) (b)

Figure 3.5 : (a) l'approche OS clustering; (b) l'architecture de Cerberus. Source : Song et al. [49].

L'idée consistant à utiliser un VMM pour exécuter plusieurs instances d'OSs afin de former un système distribué a été déjà proposée auparavant (Disco [50] en 1997 et cellular Disco [51] en 1999); Cerberus vise à fournir un environnement d'exécution compatible POSIX aux applications utilisateur et à exécuter des OSs récents sur des architectures many-cores. Pour cela, Cerberus propose une couche de virtualisation des appels systèmes qui intercepte les appels provenant des applications. Selon le service demandé, l'appel peut être transféré à une

machine virtuelle (VM) en particulier pour être réalisé par l'OS de cette VM ou il peut faire coopérer plusieurs agents (processus en arrière plan) s'exécutant sur différentes VM.

Dans Cerberus, les différents processus appartenant à une même application sont représentés par des super-procesus. Comme l'illustre la figure 3.5 (b), un *super-processus* consiste d'un processus maître en arrière plan (*deamon*) et de plusieurs *deamon* élèves (un par VM autre que celle du deamon maître). Le *deamon* maître est responsable du chargement de l'application et la création des *deamons* élèves. Un appel système concernant la création d'un processus/thread peut être réalisé soit localement dans la VM où il est issu; soit sur un autre VM en demandant le service à distance. Dans ce dernier cas, un point de sauvegarde (checkpoint) de l'état du processus est réalisé dans une zone de mémoire partagée entre VMs, puis un message est envoyé au deamon du VM cible pour demander la réalisation du service. Le deamon cible crée d'abord un processus fils, puis il accède au point de sauvegarde et il le restaure dans le contexte de ce processus fils. Afin de maintenir la vision d'un espace d'adressage virtuel partagé (exigée par la norme POSIX), Cerberus intercepte les appels systèmes relatifs à la gestion de l'espace virtuel d'un processus (p. ex : mmap, munmap, etc) et les propage aux deamons des VMs dans lesquelles il existe des processus/threads appartenant à l'application concernée. L'interception des appels système au niveau de chaque OS nécessite un module nommé *SuperP* et il doit être chargé par chaque noyau au démarrage.

Les évaluations de performances présentées par Song et al. [49] ont montré que le benchmark Dbench (ciblant le système de fichiers) rencontre une limitation de scalabilité dès 6 cores sur Linux (version 2.6.35) et dès 12 cores pour Cerberus sur une machine AMD à 48 cores. Sur une machine Intel à 16 cores, l'évaluation de deux benchmarks Appache et Memcachd a montré que le débit de traitement par core (en nombre de requêtes par seconde) chute quand le nombre de cores utilisé augmente dans le cas de Linux; tandis que ce débit par core reste quasi stable dans le cas de Cerberus.

L'approche OS Clustering constitue un compromis entre : (i) concevoir un système d'exploitation sclable par construction, et (ii) améliorer la scalabilité d'un système existant. Néanmoins, cette approche repose sur des techniques de virtualisation matérielle et nécessite des techniques d'interception des appels systèmes, des moyens de communication et de coordination inter-VMs qui sont complexes et coûteux en comparaison avec le cas d'un seul système d'exploitation natif prenant en charge directement toutes les ressources de la machine. Le principe de l'approche OS Clustering consistant à faire coopérer plusieurs instances d'un système d'exploitation pour mieux passer à l'échelle est tout à fait comparable à celui de l'approche Hierrarchical Clustering, mais la mise en œuvre est différente. OS Clustering, comme Hierarchical Clustering, ne résout pas les problèmes de scalabilité et de localité des accès mémoire au niveau de chaque OS utilisé. Song et al. n'abordent pas le problème d'avoir une gestion coordonnée et scalable des ressources concernant l'allocation mémoire et l'équilibrage de charges. En comparaison avec l'utilisation d'un unique noyau monolithique, la multiplication des niveaux d'abstraction dans Cerberus doit accroître le coût et la complexité de la mise en œuvre des migrations de tâches. Enfin, Song et al. ont mentionné que le code d'une application s'exécutant au dessus de plusieurs VMs est partagé entre ces différents VMs; ce qui pose la question de la localité des accès mémoire lors des miss instruction.

3.1.5 Approches alternatives orientées utilisateur

Boyd-Wickizer et al. ont proposé, dans le cadre d'une coopération entre le MIT, l'université de Fudan et Microsoft, le système d'exploitation Corey [52] (2008). Corey a été motivé par l'idée que les applications utilisateurs doivent contrôler explicitement le placement mémoire

et le degré de partage des ressources noyau. Ainsi, trois abstractions ont été proposées : (i) *Adresse Range* qui permet au programmeur d'une application de décider quelle partie de l'espace d'adressage est privée par core et laquelle est partagée; (ii) *Kernel Core* qui permet au programmeur d'une application d'affecter un service noyau (p. ex : piloter une carte réseau) à un core en particulier; et (iii) *Shares* qui permet au programmeur d'une application de contrôler le degré de partage de certaines tables de correspondances au niveau noyau (p. ex : la table de descripteurs des fichiers ouverts par le processus).

En donnant le contrôle aux programmeurs des applications utilisateur, Corey permet en effet de personnaliser le comportement du noyau selon les besoins et les spécificités d'une application. Les inconvénients majeurs de cette approche sont : (i) lier fortement l'application au système qui l'exécute, ce qui élimine complètement ou partiellement la portabilité des applications et complique sérieusement la réutilisation du code existant; (ii) augmenter la complexité de programmation et nécessiter d'avantage d'efforts de la part des programmeurs d'applications (gestion explicite des régions virtuelles, placement et partage des structures de données noyau, etc.); et (iii) générer des conflits entre les choix propres de chaque application dans le contexte de l'exécution dynamique de plusieurs applications simultanées.

Lieu et al. à Berkeley ont proposé Tessellation [53] (2009) préconisant le partitionnement spatio-temporel des ressources matérielles (cores, caches et bande passante de l'interconnect) entre les applications de façon à garantir une certaine garantie de service et d'isolation. Le noyau de Tessellation est une sorte de mini-hyperviseur qui fournit des conteneurs de ressources appelés "Cellules" [54]. Une cellule est l'unité de base pour le calcul et la protection. Elle constitue une isolation en performance vis-à-vis d'autres cellules et elle permet au logiciel s'exécutant en mode utilisateur dans cette cellule, d'avoir un accès à toutes ses ressources incluant les cores et pages mémoires. Les communications inter-cellules se fait par passage de messages en utilisant des canaux de communications sécurisés avec une garantie en qualité de service.

Tessellation repose également sur un paradigme de programmation par passage de messages entre cellules. Une application utilisateur sous Tessellation dispose d'un accès directe aux ressources incluant les cores et pages mémoire, ce qui n'est pas standard, rend l'application dépendante au système et ne permet pas la réutilisation du code existant. Dans Tassellation, comme dans Corey, la question de la localité des accès mémoire (données, instructions et informations de traduction) et la question de la gestion coordonnée des ressources (cores et mémoires physiques), n'ont pas été abordées.

3.1.6 Approche monolithique traditionnelle

L'approche monolithique traditionnelle des noyaux des systèmes d'exploitations repose entièrement sur le paradigme de programmation en mémoire partagée. Dans un noyau monolithique l'ensemble des services systèmes sont fournis par le noyau qui s'exécute en mode privilégié (système), tandis que les applications, les runtimes et les bibliothèques s'exécutent en mode non privilégié (utilisateur). L'approche monolithique continue à être l'approche dominante depuis Unix (1ère édition, 1971), en passant par ses différents descendants (BSD, IRIX, AIX, SunOS, Solaris, etc.) et jusqu'à aujourd'hui avec des systèmes d'exploitation dits Unix-like, tel que Linux.

Des efforts importants ont été réalisé afin d'améliorer la prise en compte par les noyaux monolithiques, du nombre croissant de processeurs à fur et à mesure l'émergence des architectures multiprocesseurs. Cantril et Bonwick [56] (2008) ont résumé le contexte historique de cette évolution en indiquant que les différentes techniques résultantes ont été

intégrées dans les noyaux modernes tels que Linux, FreeBSD et Solaris. Cui et al. [57] (2010) ont comparé les performances de Linux, FreeBSD et Solaris en utilisant des microbenchs et des applications sur une machine AMD multi-cores ayant 32 cores; ils ont trouvé que Linux et Solaris donnent de meilleures performances que FreeBSD. Étant donné que Linux est un noyau monolithique moderne représentatif, il constitue une référence souvent citée dans les travaux de recherches sur la scalabilité des systèmes d'exploitation. Nous allons donc examiner des travaux visant à améliorer la scalabilité de Linux dans le contexte des architectures cc-NUMA.

Parallélisme et synchronisation

L'évolution du parallélisme dans le noyau Linux depuis sa version 2.0 (1996) et jusqu'à récemment (la version 2.6 et ses variantes), peut être résumée en trois étapes. Dans la première, un support pour les architectures multi-processeurs (ou SMP) a été introduit à travers l'introduction d'un verrou global à attente active, nommé *BKL* (Big Kernel Lock), protégeant l'ensemble des sous-systèmes du noyau. Cette première approche pour un support des architectures SMP sans modification profonde du code du noyau a été utilisée également dans le cas de FreeBSD avec l'utilisation du "Giant Lock" [58]. Dans la deuxième étape, le parallélisme dans le code du noyau a été augmenté en introduisant un verrou gros-grain pour chaque sous-système, puis en utilisant des verrous à grain-fin (p. ex : spinlocks, semaphores, read/write locks, etc) au niveau de structures de données. Dans la troisième étape, le parallélisme des accès en lecture a été renforcé pour les traitements critiques en utilisant des techniques lock-free pour les lecteurs comme les SeqLocks [59] ou wait-free pour les lecteurs comme RCU [60,61,62]. Subissant une fréquence d'accès en lecture/écriture élevée, l'ordonnanceur global protégé par un verrou a été partitionné et remplacé par un ordonnanceur par processeur ayant son propre verrou [63].

Scalabilité sur les architectures cc-NUMA

Parallèlement à l'évolution du parallélisme dans le noyau Linux, des efforts importants ont été menés pour prendre en compte les architectures cc-NUMA au début des années 2000. Bligh et al. ont résumé les différents changements dans le noyau qui ont été introduits pour supporter ces architectures [64] (2004).

Bryant et Hawkes de SGI ont décrit leur expérience en portant une version de Linux sur une machine *SGI Altix 3000* à 128 processeurs Itanium [65] (2003). Les auteurs ont rencontré plusieurs problèmes tels que : l'utilisation du *BKL* par les systèmes de fichiers Ext2 et Ext3; la contention sur le verrou protégeant l'ordonnanceur d'un processeur chargé lorsque plusieurs processeurs oisifs tentent de lui "voler" des tâches pour équilibrer la charge; la latence élevée de l'opération "réveiller une tâche" lorsque le processeur cible, étant oisif, se trouve en attente de prise de verrou de l'ordonnanceur d'un autre processeur plus chargé. En utilisant XFS à la place de Ext2 et Ext3, la scalabilité du système de fichiers a été améliorée de 4 à 24 processeurs. En utilisant une API spécifique (*CpuMemSets*, développée par SGI) permettant au programmeur de contrôler l'allocation mémoire des processus et leurs placement sur les processeurs, les auteurs ont reporté une très bonne scalabilité du benchmark STREAM (triad test) sur 64 processeurs. Deux autres résultats ont été également rapportés sur des applications HPC : un speedup de 40 pour l'application Start-CD et de 14 pour l'application Guessian-98 sur 64 et 32 processeurs respectivement.

Bryant et al. de SGI ont poursuivit l'expérience de portage de Linux sur *Altix 3000* avec une configuration plus large ayant jusqu'à 512 processeurs Itanium [66] (2004). Bien que les auteurs de ce retour d'expérience n'aient pas présenté des résultats de benchmarks, ils ont en

revanche décrit plusieurs problèmes importants. Un de ces problèmes est les conflits de lignes de caches, causés par des mises à jour des variables globales effectuées par chaque processeur. Ces conflits de lignes de caches peuvent conduire à des situations où la progression de l'exécution du noyau sur 256 processeurs se retrouve bloquée. D'autres part, lorsque des opérations nécessitent l'inspection des variables allouées par processeur et que ces opérations concernent un nombre important de processeurs, elles génèrent un taux de miss de cache et de TLB qui impacte d'une manière importante l'exécution de ces opérations. Ils ont rencontré ce problème avec l'équilibrage de charge : périodiquement (ou quand un processeur devient oisif) chaque processeur doit consulter les ordonnanceurs des autres processeurs (précisément, les champs *nr_running* et *cpu_load* de la structure *runqueue* de chaque processeur) pour déterminer s'il existe un processeur surchargé.

À une échelle plus réduite (Core-Duo Itanium 2 processeurs à deux nœuds cc-NUMA) et en utilisant des benchmarks de type OLTP (Online Transaction Processing), Gough et al. [67] (2007) ont mesuré une latence importante lors de l'exécution de la fonction *try_to_wake_up* qui réalise l'opération "réveiller une tâche". Cette latence est causée par une forte contention sur les lignes de caches de la structure *runqueue* qui représente le descripteur de l'ordonnanceur par processeur dans Linux. Ils ont identifié que les accès distants et la prise de verrou lors de l'opération "réveiller une tâche" génère ces contentions. L'opération "réveiller une tâche" est critique notamment dans le contexte OLTP, car elle est fréquente : (i) quand un commit se termine, plusieurs centaines de processus de fond sont à réveiller; et (ii) la plupart des demandes d'entrée/sortie se font sur des processeurs différents de ceux qui reçoivent les interruptions relatives à ces requêtes, ce qui implique de réveiller à distance les processus en attentes de l'accomplissement de leurs requêtes d'entrée/sortie.

D'autres travaux pour améliorer la scalabilité du noyau Linux existent : Piggin a proposé une approche spéculative pour augmenter le parallélisme d'accès au cache de pages d'un fichier sans la prise de verrou pour les accès en lecture [68] (2006). Boyd-Wickizer et al. ont analysé la scalabilité de Linux sur une machine AMD à 48 cores [69] (2010). Ils ont identifié plusieurs goulots d'étranglement tels que : l'accès en concurrence sur des ressources globales (la table des systèmes de fichiers accessibles ou "*mount table*") et les compteurs de références concernant des objets fréquemment utilisé (les entrées dans le cache de métat-données du VFS). La réplication (par core) de la table des systèmes de fichiers accessibles et l'introduction d'un compteurs de référence distribué à la place des compteurs à base d'accès atomique ont permis de passer à l'échelle les applications évaluées jusqu'à 48 cores. Clements et al. se sont intéressés au parallélisme lors de défauts de pages [70] (2012). Lors d'un défaut de pages, la région virtuelle contenant l'adresse fautive doit être recherchée. En utilisant la technique RCU, ils ont proposé un arbre binaire équilibré représentant l'ensemble de régions présentes dans l'espace d'adressage virtuel du processus. La conception de cet arbre permet aux accès en lecture (les défauts de pages) de se faire sans conflit avec les accès en écritures (*mmap* et *munmap*) qui eux restent sérialisées par un verrou. Pour les trois applications multi-threads évaluées sur une machine AMD à 80 cores, les auteurs ont mesuré un gain en performance allant de x1.7 à x3.4 par rapport à la solution d'origine présente dans le noyau (un arbre bicolore protégé par un verrou pour toutes les opérations).

Les différents travaux cités plus haut montrent un effort continu pour améliorer la scalabilité du noyau Linux afin de lui permettre de prendre en charge des machines cc-NUMA. À notre connaissance, aucun résultat de scalabilité n'a été rapporté au-delà de 80 cores, malgré les tentatives de portage sur des machines cc-NUMA ayant plusieurs centaines de processeurs. L'analyse des différents travaux sur la scalabilité du noyau Linux, montre que la démarche suivie dans l'évolution de ce noyau est centrée sur l'augmentation du parallélisme des

services systèmes. Nous pensons que cette démarche n’est pas suffisante pour passer à l’échelle les services systèmes sur un processeur many-core cc-NUMA. Nous pensons qu’il est primordial de considérer la localité des accès mémoire lors de la réalisation de ces services. En effet, si un service système accède fréquemment à des structures de données en lecture seule, des techniques telles que RCU permettent d’augmenter le parallélisme de ces accès, mais ne résout pas la question de la localité des accès mémoire à ces structures. Le trafic mémoire généré par ces accès sollicite le LLC responsable des lignes de caches contenant ces structures, et ce LLC devient très rapidement un goulot d’étranglement. Un autre aspect à prendre également en compte, c’est la consommation énergétique dans le contexte des processeurs many-cores. En effet, le trafic distant accroît la consommation éclectique d’une manière proportionnelle au nombre de bits transférés et à la distance entre l'initiateur et la cible de ce transfert. Lors de la réalisation d’un service système, la localité des accès mémoire ne concernent pas uniquement ceux qui sont liés aux miss data, mais également ceux liés aux miss instruction et aux miss TLB. Le retour d’expérience de Bryant et al. [66] concernant le portage de Linux sur une machine cc-NUMA à plusieurs centaines de processeurs (cité plus haut) souligne l’impact négatif lié aux conflit de lignes de caches, au miss de cache et au miss de TLB; au point même de bloquer l’exécution du noyau Linux à 256 processeurs. Par conséquent, nous pensons qu’il est primordial que les structures de données d’un noyau monolithique soient distribuées; que le code et les informations de traduction soient répliqués; et que les services systèmes soient répartis afin de renforcer la localité des accès mémoire et augmenter la disponibilité du noyau.

La stratégie d’affinité mémoire First-Touch proposée par défaut dans les noyaux monolithiques actuels ignore l'existence de la phase d’initialisation séquentielle dans une application parallèle. Par conséquent, les données allouées et initialisées dans la phase d'initialisation séquentielle se retrouvent accédées à distance par les threads de traitement de la phase parallèle de l’application; ce qui limite la scalabilité d’une application parallèle et augmente le trafic distant. D’autre part, les noyaux monolithiques souffrent d’un problème de dualité entre l’objectif de l’équilibrage de charge et l’objectif de renforcement de la localité des accès mémoire aux données d’une application. En effet, un noyau monolithique peut allouer les pages physiques sur le nœud cc-NUMA où le thread demandeur se trouve; ce qui rend les accès mémoire par la suite locaux aux données contenues dans ces pages. En revanche, le noyau détruit systématiquement cette localité lorsqu’il transfère l’exécution d’un thread vers un nœud moins chargé, car le noyau est intrinsèquement incapable de migrer les données d’un thread. Par ailleurs, les noyaux monolithiques sont incapables de renforcer la localité des accès mémoire aux instructions et aux informations de traduction pour une application multi-threads. Nous pensons que ces différents problèmes ont une racine commune : l’approche monolithique manque d’une abstraction nécessaire afin de renforcer la localité des accès mémoire d’une application multi-threads. Par conséquent, il est nécessaire de remettre en question le modèle de threads proposé par les noyaux monolithiques et revoir l’organisation de l’espace d’adressage virtuel des processus.

3.1.7 Conclusion

Les différentes approches alternatives à l’approche monolithique présentées dans cette section ont un message commun : la conception monolithique des systèmes d’exploitation reposant sur les techniques de programmation en mémoire partagée ne passe pas à l’échelle. Cependant, les travaux récents sur la scalabilité des systèmes d’exploitation monolithiques montrent que la question reste ouverte. Nos travaux de recherche explorent la question de la scalabilité de l’approche monolithique dans un nouveau contexte celui de processeurs many-cores à plusieurs centaines, voire un millier, de cores. En particulier, nous nous intéressons à la scalabilité d’une même application hautement multi-threads exécutée par un unique

système d'exploitation à base de noyau monolithique et prenant en charge un processeur many-core cc-NUMA.

Comme nous l'avons mentionné plus haut, nous pensons que la démarche suivie jusqu'à présent pour faire évoluer les noyaux monolithiques n'est pas suffisante et qu'il faut mettre la question de la localité des accès mémoire au centre de cette évolution. Étant donné que nous visons : (i) restructurer d'une manière distribuée la gestion des ressources mémoires et cores, (ii) introduire des mécanismes de coordination scalable, et (iii) redéfinir le concept de thread et l'organisation de l'espace d'adressage virtuel des processus; nous avons décidé de concevoir et de réaliser un système d'exploitation expérimental afin de valider et d'évaluer nos solutions. Ce système d'exploitation nommé ALMOS (Advanced Locality Management Operating System) dispose d'un noyau monolithique distribué en mémoire partagée et il sera présenté dans le chapitre 4.

3.2 Localité des accès mémoire

Dans le contexte des architectures matérielles à caractère NUMA, le renforcement de la localité des accès mémoire des tâches consiste à faire en sorte que la majorité des accès mémoire s'effectuent localement dans le nœud où la tâche s'exécute ou à proximité. Deux écoles de pensée existent concernant la question du renforcement de la localité des accès mémoire. La première prône une gestion transparente de la localité vis-à-vis de "l'utilisateur", tandis que la deuxième prône une gestion explicite de la localité par ledit "utilisateur". Selon le niveau d'abstraction, l'utilisateur peut être : le noyau du système d'exploitation, l'environnement d'exécution d'un langage parallèle (runtime) ou encore, le programmeur d'une application parallèle.

3.2.1 Prise en charge au niveau utilisateur

Parmi les premières études concernant le placement conjoint des tâches, de leurs instructions et de leurs données est celle de Schwan et Jones [71] (1986) sur l'une des premières machines NUMA, la CM* [72] (1976). Les auteurs ont proposé un environnement de développement d'applications parallèles permettant au programmeur de contrôler explicitement le placement d'un processus et de ses segments mémoire (code, pile, données). Depuis, différents travaux ont été proposés allant dans le sens d'une prise en charge de l'aspect NUMA des architectures matérielles depuis l'espace utilisateur [73,74,75,76,77,78,79,80,81,82,83]. Des outils d'instrumentation, permettant d'aider les programmeurs des applications parallèles à inspecter les profiles d'accès mémoire de leurs programmes dans le contexte des architectures à caractère NUMA, ont été également proposés [84,85,86,87].

La gestion de l'aspect NUMA depuis l'espace utilisateur repose généralement sur les techniques suivantes [84] : (i) contrôler explicitement le placement des threads sur les cores; (ii) partitionner les données à traiter entre les threads du traitement; (iii) toucher les données avec une granularité d'une page pour provoquer leur allocation effective; et (iv) conserver, autant que possible, le même partitionnement des données entre les threads de traitement. Une ou plusieurs phases d'instrumentation permettent d'ajuster le choix d'assignation des threads sur les cores et le partitionnement des données entre les threads. L'instrumentation et/ou l'ajustement peuvent être effectués d'une manière : (i) directe, par le programmeur (éventuellement en pré-exécutant son programme tout en utilisant les outils d'instrumentation) [76,73,77,74,78,75]; ou (ii) indirecte, par le runtime du langage de programmation utilisé [82,83] ou encore par un runtime dédié ayant une porté plus large (agissant sur une ou plusieurs applications) [79,80,81]. Ces différentes approches reposent sur

un support explicite de la part du noyau d'un système d'exploitation. Ainsi, des services systèmes sont nécessaires pour permettre à l'espace utilisateur de contrôler : (i) le placement des threads sur les cores, (ii) les stratégies d'allocation et de migration des pages physiques, et (iii) l'accès aux différents compteurs matériels (la disponibilité de ces derniers varie d'une architecture à une autre). Ces services systèmes ne sont pas standard et chaque noyau propose ses propres interfaces de contrôle (p. ex : *libnuma* [20] dans le cas de Linux et la *Locality Group API* [21] dans le cas de Solaris).

La gestion explicite des aspects NUMA de la part du programmeur impacte négativement la portabilité des applications et nécessite une réécriture des applications existantes. Une prise en charge par le runtime d'un langage de programmation parallèle n'est bénéfique que pour les applications écrites par ce langage ou l'environnement parallèle qu'il utilise (p. ex : Map&Reduce, OpenMP, etc). Les décisions prises par une application ou un runtime peuvent être contradictoires à celles prises par un autre, ou encore contradictoires à celles du noyau. Une gestion réalisée par un runtime à portée globale a l'avantage de ne pas imposer une modification préalable aux applications instrumentées et ne pas lier cette gestion à un langage en particulier. Néanmoins, elle partage le même inconvénient que les deux autres approches (par le programmeur et par le runtime du langage) puisqu'elle ne peut pas faire face aux conséquences de l'équilibrage de charges réalisé par le noyau. D'autre part, son contrôle concernant le placement et/ou la migration de pages physiques se limite aux pages de données des applications utilisateur. De plus, étant donné qu'elle repose sur un échantillonnage périodique de certains compteurs matériels, la gestion de l'aspect NUMA par un runtime à portée globale ne peut convenir que pour des applications ayant un important temps d'exécution (p. ex : des applications de classe HPC).

3.2.2 Prise en charge au niveau noyau

La prise en charge de l'aspect NUMA au niveau du noyau repose sur la gestion des ressources matérielles et nécessite une connaissance fine de la topologie matérielle de l'architecture cible. Des travaux visant la prise en charge de l'aspect NUMA au niveau noyau ont été proposés dès la fin des années 1980 [88,89,90,91,92]. Ces études ont été conduite sur des machines NUMA (notamment : BBN Butterfly, IBM ACE et IBM RP3) ne disposant pas de cohérence de caches assurée par le matériel. Les stratégies de migrations et réplications proposées partagent un double objectif : fournir l'abstraction de mémoire partagée cohérente pour les applications et renforcer la localité des accès mémoire.

En modifiant le noyau du système d'exploitation IRIX (un descendant de Unix proposé par SGI), Chandra et al. [93] (1994) ont évalué différentes stratégies d'ordonnancement et une stratégie de migration automatique de pages sur une machine cc-NUMA DASH [94] à 16 processeurs. Le mécanisme de migration de pages repose sur la gestion logicielle des miss TLBs des processeurs MIPS R3000 de la machine DASH. À chaque miss TLB, le noyau vérifie si la page physique est locale ou distante. Si la page est distante, le noyau procède à sa migration. Quand une page a été migrée, elle n'est éligible à une nouvelle migration qu'au bout d'une seconde. Les stratégies d'ordonnancement évaluées sont : affinité au cache, affinité au cluster, et affinité combinée au cache et au cluster. Les auteurs de cette étude ont trouvé que l'affinité combinée appliquée conjointement avec la migration automatique de pages améliorent effectivement les performances.

Verghese et al. [95] (1996) ont étudié l'amélioration de performances apportée par un support de migration et réplication de pages au niveau noyau. Pour cela, ils ont introduit dans le noyau d'IRIX (version 5.2) un mécanisme de migration/réplication basé sur un échantillonnage d'un compteur de miss présent dans le contrôleur de répertoire de cache L2

de chaque nœud cc-NUMA de l'architecture cible. L'architecture cible est un prototype virtuel d'une machine FLASH à 8 processeurs, modélisé en utilisant SimOS [96] (un simulateur de système complet disposant d'un modèle précis au cycle du répertoire de cache L2 de FLASH). À chaque fois le compteur de miss de cache atteint une valeur de seuil, une interruption est levée par le matériel et le noyau est amené à décider s'il faut répliquer ou migrer une page. Cette décision est basée sur la valeur de différents compteurs associés à une page tels que le compteur de partage, le compteur de migration et le compteur d'écriture. En évaluant différents benchmarks mono/multi programmé(s) (en utilisant des applications parallèles avec une granularité de processus), les auteurs ont mesuré un gain allant jusqu'à 30 %.

Pour ses machines Origin 2000 [37] (2003), SGI a doté son système d'exploitation IRIX (à partir de la version 6.5) d'une stratégie de migration automatique de pages physiques au niveau noyau [97]. Cette migration automatique repose sur la présence d'une assistance matérielle présente dans les machines SGI Origin. Le matériel est capable de générer une interruption si un nombre "excessif" de références distantes à une page physique (depuis d'autres nœuds cc-NUMA) a été comptabilisé. L'ISR (Interrupt Service Routine) de cette interruption décide alors si la page doit être migrée ou non. Cette décision est basée sur un certain nombre de filtres et de seuils qui servent à limiter le nombre de migrations d'une page.

La prise en charge de la localité des accès mémoire des applications utilisateur par les noyaux récents (p. ex : Linux et Solaris) se limite sur des stratégies d'affinité mémoire statique (p. ex : First-Touch, Interleave, etc.). Ces stratégies d'affinité mémoire ne reposent pas sur une assistance spécifique du matériel. Le programmeur d'applications peut explicitement influencer le choix de la stratégie utilisée au cours de l'exécution de son programme en faisant appel à des APIs [20,21] spécifiques à chacun de ces systèmes d'exploitation.

Les différents mécanismes de réplication/migration de pages physiques cités plus haut s'appliquent uniquement aux pages physiques des applications utilisateur et partagent le même inconvénient : ils s'appliquent avec une granularité de processus et non pas de thread. Plus particulièrement, un mécanisme de réplication de pages nécessite, par construction, la traduction de la même adresse virtuelle en deux ou plusieurs adresses physiques différentes selon le nœud cc-NUMA. Cette traduction n'est pas possible avec la définition même d'un processus (un unique espace d'adressage virtuel) dans les noyaux monolithiques tels que IRIX, Linux ou Solaris. Quant aux mécanismes de migration de pages proposés, ils reposent sur une particularité architecturale (résolution par logiciel des miss TLB [93]) ou sur une assistance matérielle spécifique (compteur des accès distants à une page [97] ou compteur des miss de cache au niveau du répertoire de cache d'un nœud [95]). D'autre part, la réplication de pages pose un inconvénient plus impactant dans le contexte de processeurs many-cores que dans le contexte des machines multi-processeurs cc-NUMA. En effet, un des problèmes accentués par la réplication de pages est l'augmentation du trafic avec la mémoire externe, car la réplication de pages sous-exploite la capacité totale de caches du processeur. Cette augmentation de trafic ne fait qu'augmenter la pression sur le peu de contrôleurs mémoire qu'un processeur many-core en dispose (typiquement deux à quatre contrôleurs mémoire). Par conséquent, la réplication de pages, notamment celles de données, ne doit pas être appliquée systématiquement.

Dans cette thèse, nous proposons de répliquer à la demande les informations de traductions et les pages de code d'une application. Comme il est présenté dans la section 5.3.5, la réplication de code est effectuée par nœud cc-NUMA, mais elle peut très bien être effectuée

par cluster logique (un regroupement de nœuds voisins). Concernant les pages de données d'une application utilisateur, nous ne proposons pas une réplication, mais une migration (même pour celles en lecture seule). Les mécanismes de réplication et de migration proposés ne reposent sur aucune particularité architecturale et ne nécessitent pas d'assistance spécifique de la part du matériel. Dans le contexte des architectures multi-processeurs cc-NUMA, nous pensons que les stratégies de réplication/migration citées plus haut peuvent être plus performantes si elles sont utilisées conjointement avec le modèle de processus hybrides présenté dans la section 5.3.5. En fin, les différentes études citées plus haut ne traitent pas la question de la localité des accès mémoire liés aux miss TLBs.

3.2.3 Prise en charge au niveau matériel

La prise en charge de la localité des accès mémoire au niveau matériel vise à décharger le logiciel de la réplication et de la migration de blocs de mémoire. L'une des premières approches matérielles est celle nommée COMA (Cache Only Memory Access) utilisée dans les machines DDM [100] (1992) et KSR-1 [101] (1992). La motivation de ce type d'architecture est la réduction des latences importantes des accès distants à la mémoire dans les architectures NUMA. L'idée principale est de permettre à un nœud de contenir localement le working-set des accès mémoire effectués par ses processeurs. Puisque ce working-set est potentiellement conséquent, comparé à la capacité d'un cache présent sur puce, une architecture COMA réquisitionne une partie de la mémoire externe d'un nœud (appelé "Attractive Memory") pour y stocker le working-set du nœud. Les défis, voire les difficultés, posés au niveau matériel par ce type d'architectures sont d'une part, la localisation et la réplication d'un bloc mémoire; et d'autre part, le surcoût en mémoire [102].

Une architecture matérielle peut implémenter un mécanisme transparent de migration avec une granularité de page physique au lieu d'un bloc mémoire. En effet, De Massas a proposé une architecture matérielle capable d'améliorer la localité des accès mémoire en relocalisant les pages physiques d'une manière complètement transparente au logiciel [55] (2009). Cette architecture est basée sur trois choix principaux : (i) l'adresse physique est traduite en adresse machine par un mécanisme matériel indépendant du système d'exploitation; (ii) l'espace d'adressage physique est distribué; et (iii) les contrôleurs mémoires sont reliés entre-eux par un canal de communication sous forme d'un anneau global dédié aux migrations de pages. Le premier choix permet de contrôler le placement des pages physiques d'une manière transparente au logiciel. Pour cela, chaque cache L1 doit disposer d'une TLB lui permettant lors d'un miss data/instruction de traduire l'adresse physique en adresse machine et de retrouver ainsi le contrôleur mémoire contenant le bloc mémoire demandé. Quand l'accès à cette TLB fait miss, le cache L1 est capable, grâce au second choix, de déterminer le contrôleur mémoire contenant la table matérielle de traduction à consulter en décodant les bits MSB de l'adresse physique. Dans cette architecture, chaque contrôleur mémoire garde des compteurs d'accès par page et il est en charge de migrer une page vers un autre contrôleur mémoire soit périodiquement, soit quand la fréquence des accès au contrôleur mémoire dépasse certain seuil. Afin de déterminer le meilleur emplacement d'une page, le contrôleur mémoire initiateur doit envoyer un broadcast à tous les autres contrôleurs pour demander leurs disponibilités. Une fois un contrôleur cible a été choisi, ce dernier doit désigner une page locale victime pour être permutée avec la page importée de l'initiateur. Une fois cette permutation de pages effectuée, un braodcast d'invalidation est envoyé vers tout les caches L1 afin que chacun invalide éventuellement les deux entrées de traduction affectées par cette permutation.

L'objectif visé par l'architecture de De Massas, à savoir, une gestion de la localité des accès mémoire d'une manière complètement transparente au logiciel y compris le système

d’exploitation, peut être particulièrement intéressant dans le cas où les systèmes d’exploitations existants sont agnostiques à l’aspect NUMA. L’architecture proposée apporte 10 à 20% de gain en temps d'exécution comparé à un placement statique des tâches par l’OS sur une architecture à 16 cores. Cependant, notons que : (i) les évaluations expérimentales menées sur cette architecture ne portent pas sur le passage à l'échelle; et (ii) la pagination de l’espace physique, les protocoles employés à base de broadcast, la double migration à chaque décision, la table de pages par nœud et les compteurs matériels par page et par nœud augmentent sensiblement le coût matériel et la complexité de l’architecture.

D’autre part, cette architecture repose sur l’utilisation d’une table de pages par nœud et d’une TLB par cache L1 (différente de celle utilisée pour la traduction des adresses virtuelles en adresses physiques). Or, le taux de miss de ces TLB par cache L1 dépend fortement du comportement de l’application utilisateur et du système d’exploitation. Le trafic généré par les miss TLB de l’ensemble des cores qui exécutent des tâches ayant accès aux mêmes pages physiques (un fichier à indexer, une image à traiter, etc) va être traité par le même contrôleur mémoire responsable de ces pages physiques; et cela quelque soit l’emplacement des pages matérielles contenant les données. Nous montrons dans le chapitre 5 (section 5.3.5) que ce trafic lié aux miss TLB est un facteur important de limitation de la scalabilité. Par conséquent, le contrôleur mémoire devient très rapidement un goulot d’étranglement avec l’augmentation du nombre de caches L1 et donc de cores. Concernant le trafic lié au miss des caches instructions, les pages matérielles contenant les instructions, par exemple le code du noyau, sont sollicitées par l’ensemble des tâches. Or, l’architecture proposée par l’auteur applique la même approche (relocalisation de pages) aussi bien pour les pages contenant des instructions que pour celles contenant des données. Par conséquent, quelque soit l’emplacement finalement choisi pour le code du noyau, le contrôleur mémoire qui les hébergent devient un goulot d’étranglement dans le traitement du trafic des miss des caches instructions avec l’augmentation du nombre de caches L1 et donc de cores. Enfin, le choix architectural d’utiliser un anneau global dédié au trafic de migration de pages (double migration par décision) est intrinsèquement un facteur limitant pour la scalabilité, car il devient rapidement un goulot d'étranglement avec l’augmentation du nombre de contrôleurs mémoires voulant effectuer une double migration de pages. Cette situation peut être très vite atteinte dans le cas d’un environnement d’exécution dynamique multi-programmés (p. ex : un environnement UNIX-like).

La question de la localité des accès aux blocs mémoire a été également posée dans le contexte des architectures (mono ou multiprocesseurs) ayant une grande capacité de cache LLC [103] (2002). Dans le but d'accélérer l’accès aux blocs mémoire et réduire la consommation dans de telles architectures, le cache LLC est partitionné en plusieurs partitions (slices). Par conséquent, le temps d’accès à un bloc mémoire est variable selon la localisation du cache demandeur et la partition du LLC contenant le bloc; formant ainsi une architecture dite NUCA (Non-Uniform Cache Access) [103]. Une architecture D-NUCA (Dynamic NUCA) a été proposée par Kim et al. [103] implémentant un mécanisme de migration transparente de blocs mémoire entre partitions afin de renforcer la localité des accès mémoire.

Avec l’émergence des architectures de processeurs many-cores, différentes études ont proposé des solutions matérielles aux problèmes de la localité des accès aux blocs mémoire et de l’utilisation de la capacité totale du cache LLC [104,105,106]. Dans ces architectures, le cache LLC (typiquement de deuxième niveau) est distribué sur les tuiles/clusters et ces derniers sont reliés entre-eux par un NoC [104,13,12,106].

Hardavellas et al. [107] (2009) ont analysé les accès au cache L2 générés par l’exécution,

sous Solaris 8, d'un ensemble de benchmarks de différentes domaines : OLTP, serveur web, multi-programmé et programme scientifique. L'architecture matérielle utilisée est une architecture à base de tuiles ayant 16 cores et simulée en utilisant le simulateur de système complet, FLEXUS [108]. Cette étude est particulièrement intéressante, car en analysant les accès au cache L2, les auteurs ont trouvé que : (i) les accès aux instructions et aux données partagées en lecture/écriture représentent une part importante des accès au cache L2; (ii) une stratégie de réplication pour les instructions est souhaitable; et (iii) les stratégies de réplication ou de migration pour les données partagée en lecture/écriture sont complexes et inutiles. Les auteurs ont alors proposé une architecture nommé R-NUCA permettant de regrouper logiquement les tuiles afin d'y appliquer des stratégies différentes de placement des blocs mémoire. Chaque type d'accès au cache L2 est sujet à une stratégie matérielle différente : les accès aux instructions sont répliqués par group de tuiles et les accès aux données partagées sont entrelacés d'une manière rotationnelle à l'intérieur d'un groupe de tuiles. Le regroupement des tuiles, la localisation des blocs mémoire et le classement des types d'accès sont à la charge du système d'exploitation. Le protocole logiciel/matériel sous-jacent repose entièrement sur la gestion par logiciel des miss TLB.

Nous partageons l'analyse de Hardavellas et al. [107] sur l'importance de la mise en place de stratégies adaptées pour le trafic des miss instruction et pour celui des miss data concernant les données partagées en lecture/écriture, au lieu d'appliquer systématiquement une stratégie de migration de blocs mémoire. Les applications de type OLTP utilisées par Hardvalls et al. sont connues pour être particulièrement stressant pour les caches d'instruction et ils n'ont pas distingué dans leur étude le trafic lié aux miss TLB. Nous avons observé [2] que l'augmentation en nombre de cores utilisés par une application multi-threads peut exacerber les contentions liés au miss des caches d'instruction et de TLB (instruction et data) et cela même pour des applications multi-threads qui ne sont pas connues pour être particulièrement stressant pour ces caches. À la différence de l'approche proposée par Hardavellas et al., nous pensons que la réplication des blocs mémoire d'instructions doit être entièrement effectuée par le noyau du système d'exploitation d'une manière portable et donc, sans assistance matérielle (c.f : section 4.5.3). Cette réplication transparente aux applications utilisateur ne peut être effectuée par le noyau qu'avec la granularité de page. Ainsi, les pages répliquées peuvent être partagées et allouées directement sur un tuile/cluster ou partagées et allouées par entrelacement sur un ensemble de tuiles/clusters voisins (cluster logique). D'autre part, nous pensons que le trafic lié aux miss TLB doit être pris en compte et sa localité renforcée par thread (c.f : section 4.5).

3.2.4 Conclusion

Nous avons présenté dans cette section l'état de l'art concernant la prise en charge de la localité des accès mémoire à trois niveaux différents : utilisateur, noyau du système d'exploitation et matériel. La prise en charge transparente par matériel souffre de deux inconvénients : (i) accroître la complexité de l'architecture matérielle; et (ii) limiter la scalabilité à un faible nombre de cores. Nous souhaitons donc investiguer des solutions purement logicielles afin de supporter les architectures qui n'offrent pas d'assistance matérielle pour la gestion de la localité des accès mémoire.

Nous pensons que la prise en charge de l'aspect NUMA des architectures matérielles doit être effectuée au niveau noyau. Cette prise en charge doit être faite d'une manière transparente aux applications utilisateurs pour deux raisons : la première est d'assurer une compatibilité avec les applications parallèles préexistantes sans pour autant sacrifier les performances. La seconde raison est que, dans un environnement généraliste multi-applications et dynamique, les décisions prises par chaque application expriment un besoin individuel qui peut être en

contradiction avec celles prises par les autres applications. Pour que cette prise en charge soit efficace, nous pensons que le noyau doit pouvoir contrôler les accès mémoire avec une granularité fine, celle de thread. Pour cela, nous pensons que la notion d'un thread telle qu'elle est définie dans les noyaux monolithiques n'est pas adaptée et doit être remise en question. Dans la section 4.5.3, nous présentons un nouveau modèle de threads, appelé "Processus Hybrides" qui permet de doter les noyaux monolithiques de l'abstraction nécessaire pour pouvoir contrôler finement les accès mémoire des threads.

3.4 Conclusions

Dans ce chapitre, nous avons présenté notre synthèse de l'état de l'art portant sur deux aspects : la scalabilité des systèmes d'exploitation et la localité des accès mémoire. Les différentes approches de conception alternatives ne répondent pas aux contraintes de portabilité et réutilisations des applications multi-threads existantes. En ce qui concerne les systèmes d'exploitation existants, nous pensons que la démarche suivie pour faire évoluer les noyaux monolithiques existants n'est pas suffisante et qu'il faut mettre la question de la localité des accès mémoire au centre de cette évolution.

La plupart des travaux existants sur la scalabilité des systèmes d'exploitation ont été conduits sur des machines cc-NUMA soit à faible nombre de processeurs, soit à base de processeurs multi-cores. Nos travaux de recherche explorent la question de la scalabilité de l'approche monolithique dans un nouveau contexte celui de processeurs many-cores à plusieurs centaines, voire un millier, de cores. Nous nous intéressons également à la scalabilité d'une même application hautement multi-threads exécutée par un unique système d'exploitation à base de noyau monolithique et prenant en charge un processeur many-core cc-NUMA. En particulier, nous visons : (i) restructurer d'une manière distribuée la gestion et l'allocation des ressources mémoires et cores, (ii) introduire des mécanismes de coordination scalable, et (iii) redéfinir le concept de thread et l'organisation de l'espace d'adressage virtuel des processus. Étant donné que cela représente des modifications trop profondes pour modifier un noyau monolithique existant tel que Linux, nous avons décidé de concevoir et de réaliser un système d'exploitation expérimental afin de valider et d'évaluer nos solutions. Ce système d'exploitation nommé ALMOS (Advanced Locality Management Operating System) dispose d'un noyau monolithique distribué en mémoire partagée. La gestion de la localité des accès mémoire dans ALMOS est prise en compte au niveau noyau d'une manière transparente aux applications utilisateur. Cette prise en compte concerne les accès mémoire liés aux miss data, miss instruction et miss TLB de chaque thread; elle est effectuée sans assistance matérielle particulière. ALMOS est présenté en détail dans le chapitre 4.

Chapitre 4

Principe de la solution proposée

Ce chapitre présente les principes généraux sur lesquels repose le système d'exploitation expérimental ALMOS. Il s'agit d'un nouveau système d'exploitation que nous avons conçu et réalisé durant notre thèse afin d'apporter des réponses aux questions posées dans le chapitre 2. Nous commençons par présenter une vue globale d'ALMOS, puis nous décrivons les principaux mécanismes de son noyau permettant la prise en compte des caractéristiques NUMA des architectures cibles.

4.1 Le système d'exploitation ALMOS

Nous visons le renforcement de la localité des accès mémoire des tâches et la gestion coordonnée des ressources mémoire et cores dans un noyau monolithique prenant en charge une architecture many-core cc-NUMA ayant plusieurs centaines de cores. Pour cela, nous avons conçu et réalisé un nouveau système d'exploitation nommé ALMOS (Advanced Locality Management Operating System) [7,4].

Au-delà de l'organisation des structures de données et la gestion décentralisée des ressources, notre approche de conception introduit deux nouvelles abstractions absentes dans les noyaux monolithiques. Il s'agit de la notion de "répliqua noyau" et de la notion de "processus hybride". Ces deux abstractions permettent au noyau d'une part, de renforcer la localité des accès mémoire liés aux miss instruction et aux miss TLB lors de l'exécution d'une tâche; et d'autre part, de disposer d'un contrôle plus fin sur le placement des pages physiques passant d'une granularité de processus à une granularité de fil d'exécution. Notre approche vise à rendre la prise en compte de l'aspect NUMA de l'architecture matérielle ainsi que la gestion de ses ressources, transparentes aux programmeurs des applications parallèles, sans pour autant sacrifier les performances. Ce qui permet de maintenir une compatibilité avec les standards et d'assurer la portabilité des applications existantes.

ALMOS offre un environnement d'exécution multi-processus compatible avec le standard POSIX. Chaque processus peut contenir plusieurs threads conformément à la norme PThreads. Comme dans un UNIX, un nouveau processus est crée par duplication (*fork*) tandis qu'un nouveau programme est lancé par transformation du processus appelant (*exec*); et d'autre part, l'abstraction principale des ressources systèmes accessibles par l'utilisateur est le fichier. Un fichier peut représenter un fichier régulier, un périphérique ou des structures internes du noyau. Bien que distribué, le noyau d'ALMOS est monolithique puisque tous les services systèmes s'exécutent au niveau de privilège noyau. Ces services systèmes sont regroupés par la nature de leurs services formant ainsi des sous-systèmes dédiés.

Le noyau d'ALMOS comporte les sous-systèmes primordiaux que nous pouvons retrouver dans des systèmes plus matures comme Linux, BSD ou Solaris. Il s'agit d'un sous-système gestionnaire des ressources cores et mémoires, un sous-système gestionnaire des processus et threads, un sous-système gestionnaire de l'espace virtuel des processus et un sous-système gestionnaire des fichiers et périphériques. La figure 4.1 montre un aperçu global du système d'exploitation ALMOS.

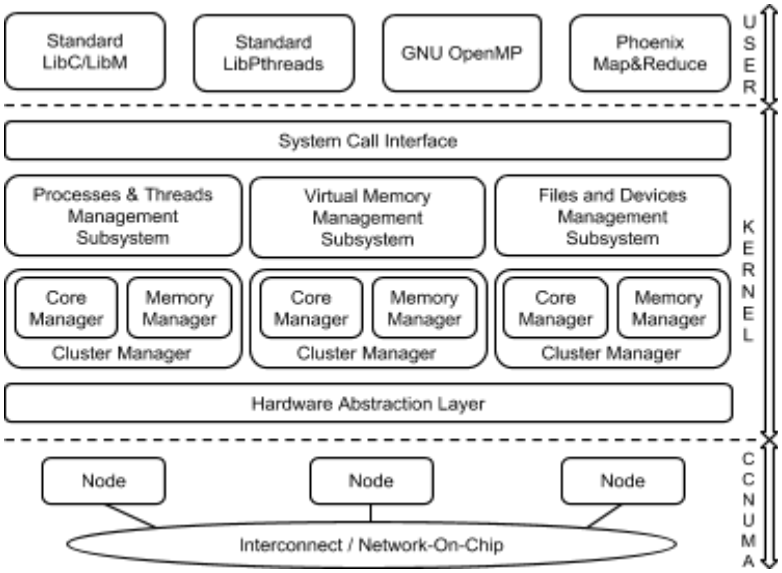


Figure 4.1 : un aperçu global du système d'exploitation ALMOS

Dans le but de valider la compatibilité de notre conception avec les standards existants et d'autre part, de faciliter les expérimentations, l'environnement utilisateur du système d'exploitation ALMOS : (i) dispose d'une bibliothèque C (libc) standard assez complète ainsi que d'une bibliothèque mathématique standard; (ii) assure un environnement d'exécution parallèle selon le standard PThread (libpthread); et (iii) Intégre quelques exécutifs (runtime) existants et qui reposent sur les deux bibliothèques systèmes (libC et libpthread) tels que : le runtime GNU OpenMP [110] implémentant le standard de programmation parallèle OpenMP et le runtime Phoenix [111,112] implémentant un autre modèle de programmation parallèle Map&Reduce [113].

Concernant nos hypothèses sur l'architecture matérielle sur laquelle ALMOS peut s'exécuter, nous ne faisons pas d'hypothèses particulières hormis le fait qu'il s'agit d'une architecture cc-NUMA pouvant contenir plusieurs centaines de nœuds. Comme dans tout système d'exploitation portable, nous avons mis en place les abstractions et les couches logicielles nécessaires afin que les mécanismes et services du noyau fonctionnent d'une manière indépendante de l'architecture sous-jacente. Par exemple, le noyau d'ALMOS représente la notion de voisinage entre les nœuds cc-NUMA de l'architecture par une structure de données abstraite construite lors de la phase de démarrage par les couches basses implémentant la HAL (Hardware Abstraction Layer) du noyau. Par conséquent, les mécanismes introduis dans le noyau d'ALMOS opèrent sur une représentation abstraite de la topologie matérielle et ne sont pas liés à une topologie matérielle particulière (p. ex: hiérarchie d'interconnects, NoC en Mesh 2D, NoC en Torus, etc).

Enfin, concernant nos hypothèses sur les applications qui peuvent s'exécuter sur ALMOS, nous faisons l'hypothèse que ces applications peuvent avoir un fort degré en parallélisme de tâches. Ainsi, une seule application multi-tâches peut solliciter l'ensemble de cores présents. Nous ne faisons pas d'hypothèses particulières sur la chaîne de compilation utilisée pour générer l'exécutable d'une application même si nous utilisons GCC [114] pour nos expérimentations.

4.2 Architecture distribuée du noyau d'ALMOS

L'architecture distribuée du noyau d'ALMOS est supportée principalement par les 5 objets ou

notions suivants : le cluster-manager, l'arbre de décision quaternaire répartie, la stratégie de migration automatique des pages, la notion de "réplica noyau", et la notion de "processus hybride". Dans cette section nous introduisons le cluster-manager. Les quatre autres objets sont décrits dans les sections suivantes : 4.3 DQDT (Distributed Quaternary Descion Tree); 4.4 Localité des accès mémoire aux données; 4.5 Localité des accès mémoire liés aux miss instruction et TLBs.

L'organisation distribuée du noyau d'ALMOS s'appuie sur les objets nommés gestionnaires de clusters (*cluster-manager*). L'objectif est d'une part, de décentraliser la gestion des ressources et d'autre part, d'assurer une localité d'accès mémoire lors de cette gestion. Un *cluster-manager* est un objet responsable de la gestion des ressources physiques (essentiellement cores et mémoire physique) d'un nœud cc-NUMA. Il existe autant de *cluster-manager* qu'il y a de nœuds cc-NUMA. Chacun de ces *cluster-manager* est placé localement dans la mémoire physique de son nœud. Comme il est illustré par la figure 4.2, un *cluster-manager* contient, entre autres, un *memory-manager* et un ou plusieurs *core-manager*. Un *core-manager* est un objet responsable de la gestion d'un core et plus précisément de l'ordonnancement des tâches affectées à ce core ainsi que de la gestion des événements reçus par ce core. L'ordonnancement est assuré par un serveur d'ordonnancement appelé (*scheduler-server*) tandis que la gestion des événements est assurée par un gestionnaire dédié (*events-manager*). Dans le restant de cette section, nous décrivons les principaux composants d'un *cluster-manager*.

4.2.1 Memory-Manager

Un *memory-manager* est un objet responsable de la gestion de la mémoire physique d'un nœud cc-NUMA. Il assure localement, tout type d'allocation/libération dynamique de mémoire physique. Un *memory-manager* s'appuie sur trois allocateurs selon la nature de la requête de l'allocation mémoire. Le premier allocateur est un allocateur de pages. L'algorithme d'allocation implémenté par cet allocateur est l'algorithme "Buddy" [115,116]. Le second allocateur est un allocateur d'objets noyau de taille fixe doté d'un cache par type d'objet [117]. Un objet noyau est définit par une structure de données d'un certain type (p. ex : un descripteur de tâche, un descripteur de fichier ouvert, un descripteur d'une région virtuelle, etc) auquel sont associés un constructeur et un destructeur. Le troisième allocateur est un allocateur de structures de données temporaires ayant des tailles variables mais relativement petites par rapport à la taille d'une page. Il gère un tas de taille fixe (non extensible) dont les pages physiques ont été réservées à l'initialisation.

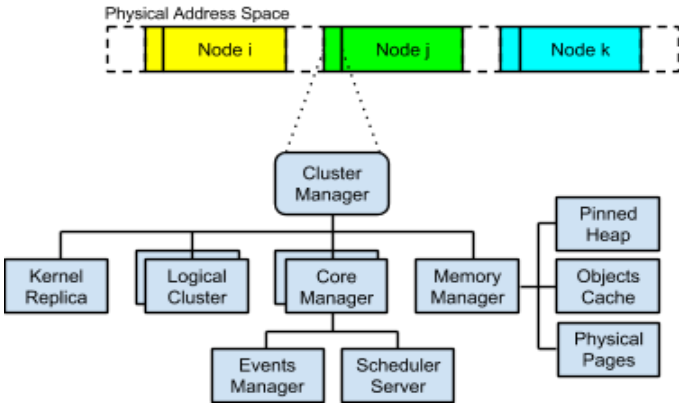


Figure 4.2 : un Cluster-Manager est un gestionnaire de ressources d'un nœud cc-NUMA
Il est physiquement placé dans le banc mémoire de son nœud.

4.2.2 Scheduler-Server

Comme nous l'avons introduit dans la section 2.3, les noyaux monolithiques existants de type Linux ou BSD mettent en place un ordonnanceur par core. Selon cette approche l'ordonnanceur est protégé par un verrou [63], car les files d'attentes (run-queues) peuvent être accédées à distance lorsque une tâche s'exécutant sur un autre core cherche à réveiller une tâche appartenant à l'ordonnanceur en question. La primitive d'ordonnancement, "réveiller une tâche", est un service de base utilisé dans toutes les synchronisations à attente passive, qui peuvent impliquer un nombre important de tâches (p. ex : une barrière, une variable de condition, un sémaphore, attente sur une page en cours d'E/S, etc).

La figure 4.3 illustre les étapes à exécuter par une tâche sur le core *A* qui cherche à réveiller une tâche *X* sur le core *B* dans une configuration à 64 nœuds. L'exécution à distance de la section critique génère un trafic d'accès mémoire subissant le facteur NUMA de l'architecture. Plus le core cible est loin, plus les latences d'accès sont grandes; et par conséquent, moins bonnes sont les performances. Plus le nombre de tâches à réveiller est grand, plus cette perte en performances l'est également. Ces accès distants impliquent un coût caché en latence, lié à la gestion de la cohérence des lignes de caches appartenant aux structures de données de l'ordonnanceur cible. En conclusion, cette approche reposant sur l'accès distant aux structures de données d'un ordonnanceur ne passe pas à l'échelle efficacement dans le contexte d'une architecture cc-NUMA.

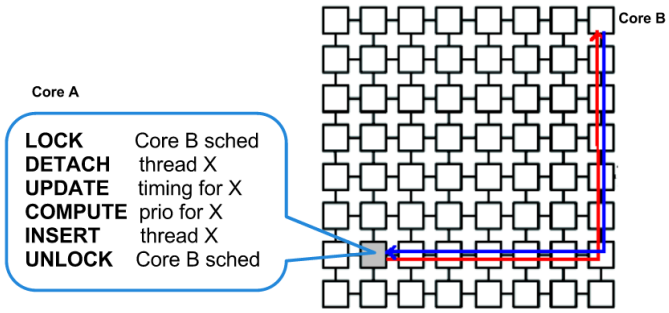


Figure 4.3 : illustration des étapes à exécuter par une tâche sur le core *A* qui cherche à réveiller une tâche *X* sur le core *B* dans une configuration à 64 nœuds cc-NUMA.

Dans le but d'avoir des primitives d'ordonnancement scalable, nous proposons dans le noyau d'ALMOS un schéma d'ordonnancement reposant sur le modèle de traitement réparti client-serveur. Notre proposition repose sur les deux observations suivantes :

- Une tâche *X* bloqué en attente passive ne peut être débloqué que par une seule tâche *Y* qui possède la ressource attendue par la tâche *X*. La tâche *Y* peut être une tâche s'exécutant sur n'importe quel core.

- Le but de la primitive d'ordonnancement "réveiller une tâche" est de rendre la tâche à réveiller éligible (selon sa priorité) à la prochaine élection sur le core cible.

- Pour un core donné, l'élection d'une nouvelle tâche ne peut être exécutée qu'aux différents points d'ordonnancement (p. ex : la tâche sortante cède le core, interruption matérielle, fin d'un appel système, etc).

Selon notre approche, l'ordonnanceur est un serveur qui attribue un identifiant unique (local) aux tâches qui lui sont affectées lors de leurs admission. Le client est toute tâche qui s'exécute sur un core différent (potentiellement distant) et qui cherche à réveiller une tâche

appartenant au serveur. Du coté du client, la primitive d'ordonnancement “réveiller une tâche” se résume à une notification de réveil envoyée au serveur en indiquant l’identifiant de la tâche. Cette notification de réveil est codée sur un mot (4 octets dans une architecture matérielle 32 bits) dont l’envoi est réduit à une écriture. Du coté du serveur, les notifications pendantes sont vérifiées aux différents points d'ordonnancement. Pour toute notification pendante, le serveur (la tâche sortante ou en cours de l’exécution) exécute l’ensemble d’étapes permettant de réaliser effectivement l’opération de réveil de la tâche notifiée. Une fois que toutes les notifications pendantes ont été prise en compte, une élection peut alors être invoquée (si la tâche en cours doit céder le core).

Cette approche de conception d’un ordonnanceur selon le modèle client-serveur a les avantages suivants :

- Un client qui cherche à réveiller *N* tâches sur différents cores effectue les notifications avec un minimum d’accès distant (au plus *N* écritures distantes). Ce qui est très bien adapté pour les architectures cc-NUMA.

- La notification faite par le client est une requête asynchrone non bloquante, ce qui permet d’exécuter l'opération de réveil pour N tâches d’une manière parallèle pour le client.

- Ce schéma distribué d'ordonnancement rend le fonctionnement d’un ordonnanceur plus déterministe, car d’une part, il n’y a plus de contention sur les structures de données de l’ordonnanceur d’un core et d’autre part, les accès mémoire lors de l’exécution effective de la primitive du réveil sont locaux au nœud du serveur (ils ne sont plus sujet aux contentions dans l’interconnect ni aux latences variables dues au caractère NUMA de l’architecture).

Cette approche d'ordonnancement réparti client-serveur peut avoir, potentiellement, les inconvénients suivants :

- Elle augmente le temps d’exécution de l’ordonnanceur car il faut d’abord exécuter les notifications pendantes avant d’effectuer une élection.

- Elle cause une latence additionnelle, mais bornée, entre le moment où une tâche prioritaire a été notifiée pour un réveil et le moment de l’exécution effective de l’opération et par conséquent, l’exécution de l’élection.

Le premier inconvénient dépend du nombre de notifications pendantes présentes simultanément lors d’un point d'ordonnancement. Or ce nombre est lié d’une part au nombre de tâches affectées à un core et d’autre part à la probabilité de devoir réveiller simultanément plusieurs tâches. Dans une architecture many-cores le noyau doit privilégier le parallélisme réel (une tâche par core) plutôt que la concurrence (multiplexage temporel) et par conséquent, il doit minimiser le nombre de tâches affectées à un core voire n’affecter qu’une tâche à un core.

Le deuxième inconvénient peut être relativisé : Dans l’approche d'ordonnancement à base de verrou, la tâche qui cherche à réveiller une autre tâche exécute elle-même l’opération de réveil qui inclut le calcul de la nouvelle priorité de la tâche à réveiller. Par conséquent, elle peut provoquer une interruption à distance obligeant ainsi la tâche sortante sur le core cible à passer dans un point d'ordonnancement et effectuer une élection. Dans notre proposition le

client notifie uniquement l'éligibilité de la tâche et c'est au serveur d'exécuter effectivement l'opération de réveil. En pratique, le calcul de la priorité dynamique d'une tâche est heuristique et varie d'un noyau à un autre voire même d'une stratégie d'ordonnancement à une autre. Dans une version du noyau de Linux [63] le calcul de la priorité dynamique d'une tâche n'est pas lié au temps d'exécution des autres tâches affectées au même core. Un tel calcul de priorité dynamique peut être effectué par le client selon notre approche, car il dépend uniquement de la tâche à réveiller. Dans ce cas, le client peut ainsi, décider s'il faut interrompre à distance le core cible pour forcer ainsi une élection.

4.2.3 Events-Manager

Le modèle client-serveur de l'ordonnanceur, décrit dans la sous-section précédente, est intéressant dans le contexte cc-NUMA quand il s'agit de réaliser un service système impliquant des accès distants à des structures de données se trouvant dans d'autres nœuds. Cela reste valable pour n'importe quel service système impliquant deux nœuds quand les accès mémoire distants génèrent un trafic important et que la distance entre ces deux nœuds est importante. La notification par l'écriture d'un seul mot, vue dans l'ordonnanceur, est un cas particulier profitant du fait qu'il ne peut pas y avoir plus d'un client qui notifie le réveil d'une tâche à son serveur, car une tâche ne peut pas s'endormir à la fois sur plusieurs ressources. Nous généralisons cette approche client-serveur à d'autres services système en introduisant une démarche qui se résume en deux points :

- Découper le travail à réaliser par le service système en deux portions. La première opère sur des structures de données locales à la tâche à l'origine du service, tandis que la deuxième opère sur les structures de données distantes du nœud cible.

- Exécuter la première portion du travail localement par le client et invoquer l'exécution de la deuxième portion du travail sur un serveur appartenant au nœud cible en postant un événement.

Le deuxième point nécessite un mécanisme similaire, dans sa sémantique, aux RPC (Remote Procedure Call) [118], LRPC (Lightweight RPC) [119] et PPC (Protected Procedure Call) [28]. En revanche, la différence est dans la mise en œuvre, car le client et le serveur sont dans la même machine (contrairement au RPC) et dans le même domaine de protection, celui du noyau (contrairement aux LRPC et PPC). Nous introduisons dans le noyau d'ALMOS un mécanisme permettant d'invoquer, à distance dans le nœud cible, l'exécution d'une fonction précise. Ce mécanisme repose entièrement sur la mémoire partagée cohérente des architectures cc-NUMA et le faite que le client et le serveur sont dans le même domaine de protection. L'implémentation de ce mécanisme est lock-free et elle est assurée par la mise en place d'un gestionnaire d'événements par core (*Events-Manager*). Un événement est caractérisé principalement par trois informations : l'adresse de la fonction à exécuter (handler), l'adresse de l'unique argument utilisé par cette fonction et une priorité. Un événement est envoyé par un client à un serveur. En réponse à cet événement, le serveur exécute le travail (le handler) associé à l'événement selon sa priorité. Selon la priorité de l'événement, son envoi par le client peut être associé à une IPI (Inter Processors Interrupt) afin de demander une prise en compte immédiate par le serveur.

Pour le client l'interface visible du mécanisme d'événements se résume à des accesseurs permettant de préparer un descripteur d'événement et à une fonction *event_send* permettant d'envoyer l'événement à un point d'écoute (listener) de l' *events-manager* du core cible. Il existe deux points d'écoute pour un *events-manager*. Le premier est dédié aux événements inter-cores, tandis que le deuxième est dédié aux événements dits différés sur le même core.

Chaque point d'écoute dispose, selon l'interface du mécanisme d'événements, d'une fonction *event_listener_notify* permettant au serveur de prendre en compte les événements pendants à un instant donné. Nous reviendrons sur le deuxième point d'écoute d'événements différés mais pour l'instant, considérons qu'il existe un seul point d'écoute, celui des événements inter-cores. Enfin il existe, au moins, une tâche noyau attachée au *core-manager* jouant le rôle du serveur d'événements. Dans le restant de cette section, nous montrons comment le mécanisme d'événements est employé afin de mettre en place un service système réparti renforçant la localité des accès mémoire.

Prenons l'exemple du service système "création d'une tâche". Comme nous l'avons mentionné dans la section 2.4, la réalisation de ce service dans les noyaux monolithiques existant tel que Linux ou BSD ne prend pas en compte le caractère NUMA de l'architecture. Les structures de données du noyau associées à la nouvelle tâche sont allouées localement depuis le nœud de la tâche appelante même si la nouvelle tâche doit être lancée sur un nœud diffèrent. Ceci a un impact négatif sur les performances de la nouvelle tâches lorsqu'elle réfère à ses propres structures de données noyau puisqu'elles lui sont distantes (dans le nœud de la tâche créatrice). Cet effet est aggravé dans le cas où une tâche crée N autres tâches placées dans des nœuds distants, ce qui est typiquement le cas dans la phase d'initialisation d'une application parallèle. Dans ce cas le trafic distant, en provenance de ces nœuds vers le nœud (origine) de la tâche initiatrice, contribue à la contention sur le LLC du nœud origine ainsi qu'a sa pollution.

Notre solution consiste à utiliser le mécanisme d'événements, comme illustré par le schéma de la figure 4.4 où une tâche sur le core *A* crée une nouvelle tâche sur le core *B* en 7 étapes.

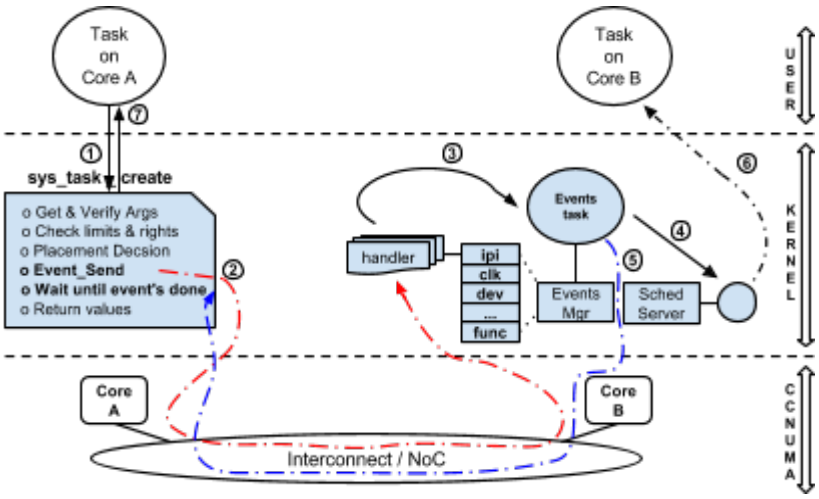


Figure 4.4 : Un schéma montrant l'approche distribuée de la réalisation d'un service système ("création d'une tâche") en utilisant le mécanisme d'événements.

La tâche sur le core *A*, le client, fait un appel système (1) afin de demander au noyau la réalisation du service "création d'une tâche" fourni par la fonction *sys_task_create*. Cette fonction réalise uniquement le traitement dont les accès mémoire sont majoritairement locaux au même nœud que le core *A*. Une fois que la décision de placement de la nouvelle tâche sur le core *B* a été prise, un événement est alors préparé et envoyé (2) au point d'écoute des événements inter-cores de l'*event-manager* du core *B*. La fonction à exécuter (*handler*) associée à cet événement est responsable de la création effective de la nouvelle tâche en allouant et initialisant l'ensemble de structures de données nécessaires au fonctionnement de

la nouvelle tâche. La priorité de cet événement est *IPI*, ce qui implique l'envoi d'une IPI pour provoquer une interruption sur le core *B*. Une fois l'événement envoyé, le client se met en attente de la fin de l'exécution à distance de sa fonction *handler*. L'interruption du core *B* provoque l'exécution de la tâche noyau *events task* qui est le serveur d'événements du core *B*.

Le serveur exécute (3) la fonction event_listener_notify pour prendre en compte tous les événements pendants sur le point d'écoute des événements inter-cores. La prise en compte d'un événement consiste à le détacher de sa liste de priorité et à exécuter sa fonction *handler*. La fonction *handler* crée effectivement la tâche tout en faisant des accès mémoire majoritairement locaux. En particulier touts les accès mémoire aux différentes ressources du *cluster-manager* du serveur sont locaux. Une fois la nouvelle tâche créée, elle est attachée (4) au serveur d'ordonnancement du core *B*. Avant de finir son exécution, la fonction *handler* signale alors (5) la fin de son exécution en positionnant un drapeau dans une case mémoire indiquée par l'argument référencé par le descripteur de l'événement. Le serveur finit par céder le core en permettant à la nouvelle tâche d'être élue et chargée (6) sur le core *B*. Du coté du client, l'exécution se poursuit dès que la vérification du drapeau, signalé par le *handler,* est réussie. Le client positionne alors les bonnes valeurs de retour avant de finir l'exécution du service système et retourner (7) en mode utilisateur.

Le service système "création d'une tâche" représente un exemple de service système réparti client-serveur dans le noyau d'ALMOS. D'autres services systèmes du noyau concernant la gestion des tâches sont également répartis selon la même démarche. Il s'agit du service réalisant une duplication de processus (fork) et du service réalisant la migration d'une tâche. Ces deux services nécessitent d'avantage d'allocations mémoire et d'accès distants que la création d'une nouvelle tâche.

Notre mécanisme d'événements ne fait aucune hypothèse sur le service rendu lié à la prise en compte de l'événement, ce qui renforce sa généricité. Ainsi nous utilisons ce mécanisme dans la gestion des périphériques afin de réduire le chemin d'exécution critique des ISR (Interrupt Service Routine) en divisant le travail d'une ISR en deux parties. La première est critique car elle concerne la réactivité du système puisqu'elle doit être faite en conservant masquées les interruptions matérielles du core concerné. Elle consiste principalement à acquitter l'interruption et à réceptionner les informations si nécessaire depuis les tampons mémoire du périphérique. La deuxième, moins critique, consiste à traiter les informations réceptionnées en ayant les interruptions matérielles réactivées. D'autre part, certain périphérique ont une fréquence d'interruption élevée. Dans ce cas, le traitement moins critique peut être déportée, en utilisant le mécanisme d'événements, sur un core oisif non directement soumis aux interruptions matérielles des périphériques. Ceci permet d'augmenter le débit de traitement des informations reçues. Si la fréquence d'interruption est relativement petite, un traitement mois critique peut être différé pour une exécution en dehors du chemin critique par le même core en utilisant un événement local.

Il existe un autre scénario où l'exécution différée d'une fonction est souhaitable, voire nécessaire, lorsqu'une tâche (utilisateur ou noyau) doit mettre fin à son propre exécution. La tâche exécute une fonction de terminaison (*task_destroy*) où un certain nombre d'étapes de

finalisation sont effectuées mais elles n'incluent pas la libération de certaines ressources mémoire utilisées par la tâche puisque qu'elle-même est en cours d'exécution (p. ex: la pile noyau, le descripteur de la tâche, etc). Nous utilisons le mécanisme d'événements pour réaliser un service de "ramasse-miettes" permettant de récupérer toutes les ressources mémoire associées à la tâche après sa terminaison. Il s'agit également d'un événement local.

Enfin, comme nous l'avons vu plus haut, un *events-manager* par core dispose de deux points d'écoutes d'événements distincts, le premier est dédié aux événements inter-cores tandis que le deuxième est dédié aux événements locaux. Deux raisons à cette séparation, la première : les événements locaux sont moins prioritaires que les événements inter-cores. La deuxième raison : l'envoi d'un événement local par un client (le producteur) et sa notification par le serveur (le consommateur) ne nécessitent pas l'emploi d'algorithmes lock-free comme dans le cas des événements inter-cores, puisque le client et le serveur sont sur le même core.

4.2.4 Conclusion

Le *cluster-manager* représentent la pierre angulaire de l'architecture distribuée du noyau d'ALMOS. Il fournit deux services principaux aux tâches qui sont placées sous sa responsabilité, à savoir, l'allocation mémoire et l'ordonnancement. Étant donné qu'il est lui-même placé localement dans le banc mémoire de son nœud, toutes les accès mémoire lors de la réalisation de ces deux services sont, par conséquent, locaux. La mise en place d'un *memory-manager* permet de répondre localement aux différents types d'allocations mémoire (tant qu'il y a suffisamment de mémoire locale). La mise en place d'un schéma d'ordonnancement distribué selon le modèle *client-serveur*, permet d'avoir des primitives d'ordonnancement scalables minimisant le trafic mémoire distant. La mise en place d'un *events-manager* par core permet principalement de découper en deux parties la réalisation des services système afin de minimiser le trafic inter-nœuds et réduire la contention sur les ressources des nœuds.

4.3 DQDT (Distributed Quaternary Decision Tree)

La décentralisation de la gestion des ressources, grâce à la mise en place d'un *cluster-manager* par nœud cc-NUMA, a comme conséquence un manque de connaissance globale concernant la disponibilité des ressources. Cette connaissance est nécessaire au noyau pour pouvoir décider sur quel nœud une requête d'allocation mémoire distante peut être satisfaite, sur quel core une nouvelle tâche doit être placée ou encore, vers quel core une tâche doit être transférée lors d'une opération d'équilibrage de charge. Dans cette section, nous décrivons notre solution à ce problème qui consiste à doter le noyau d'ALMOS d'un mécanisme décentralisé de représentation distribuée des ressources globalement disponibles. Cette représentation prend en compte le caractère NUMA de l'architecture. Ce mécanisme décentralisé et ses politiques de prise de décision constituent la DQDT (Distributed Quaternary Desion Tree).

4.3.1 Conception de l'arbre de décision réparti DQDT

Le noyau doit renforcer la localité des accès mémoire. Cette notion de localité intervient lorsque le noyau cherche à créer une tâche, placer une tâche sur un core ou encore lors qu'une tâche demande d'allouer de la mémoire. Sans perdre la généralité de nos propos, la figure 4.5 illustre ce dernier cas. La figure illustre un mesh où chaque carré représente un nœud cc-NUMA et chaque trait représente un lien de l'interconnect ou un lien du NoC. Dans (a), il existe deux tâches déjà placés sur deux nœuds différents (bleu et rouge). Dans (b), ces deux

tâches ont besoin d'allouer de la mémoire et le noyau a effectué l'allocation d'une manière aléatoire sur deux nœuds distants (bleu et rouge). (c) montre les conséquences de ce choix. En effet, les accès mémoire concernant ces données ainsi allouées se traduisent par des accès distants.

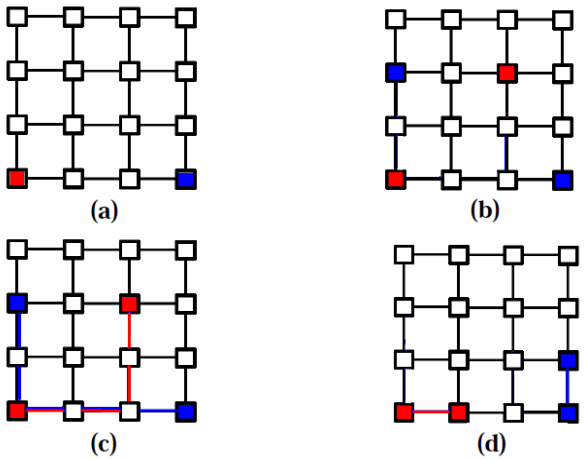


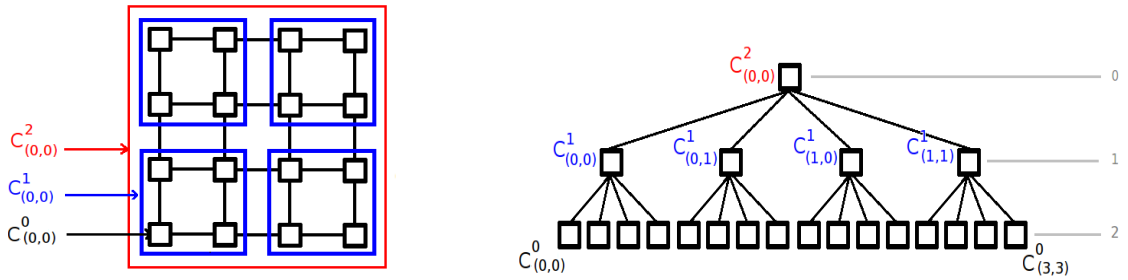
Figure 4.5 : Conséquences du choix du nœud cible lors d'une allocation mémoire dans un mesh de 16 nœuds cc-NUMA.

Ces accès distants entraînent : (i) une latence proportionnelle au nombre de nœuds traversés, (ii) une latence supplémentaire due à la contention potentielle sur le LLC du nœud cible et (iii) une augmentation de la consommation dû aux mouvements de données. Afin d'éviter ces problèmes, le noyau doit allouer la mémoire nécessaire pour une tâche localement sur le nœud où la tâche s'exécute, ou à défaut (si la mémoire manque localement) dans des nœuds "voisins" comme dans le cas (d).

Cela montre clairement la nécessité de définir la notion de voisinage entre les nœuds cc-NUMA et donc entre les ressources mémoires physiques et cores. Comme nous l'avons vu dans la section 4.2.1, la gestion des ressources d'un nœud est assurée par son *cluster-manager*. Notre DQDT a comme premier objectif : définir un voisinage entre les *cluster-managers*. Cette notion de voisinage est multi-niveaux selon la proximités des nœuds cc-NUMA. Un niveau de voisinage est représenté par un cluster logique. Un cluster logique est constitué, d'au plus, de quatre clusters physiques[5] ou de quatre clusters logiques de niveau inférieur. Dans le cas d'un interconnect (ou d'un NoC) sous forme de mesh 2D, cette relation peut être décrite comme suit : le cluster physique du coin bas gauche du cluster logique de niveau *i* et de coordonnées (x,y) est $C^0(x_{2i}, y_{2i})$.

Reprenons notre mesh de 16 nœuds cc-NUMA, la figure 4.6 (a) illustre le regroupement des clusters physiques en clusters logiques. La structure DQDT est un arbre quaternaire où chaque feuille représente un cluster physique, tandis que chaque nœud de l'arbre d'un niveau donné *k* représente un cluster logique de niveau *H-k* où *H* est l'hauteur de l'arbre. Notons que l'hauteur de l'arbre est de cinq pour une architecture ayant 1024 nœuds cc-NUMA ($log_4(1024)$). La figure 4.6 (b) illustre l'arbre DQDT représentant le mesh de 16 nœuds. Cet arbre dispose donc de 16 feuilles qui représentent les clusters physiques, de 4 nœuds qui représentent les clusters logiques de niveau 1 et d'un nœud racine qui représente le seul cluster logique de niveau 2. L'hauteur de cet arbre est de 2.

5 Dans le restant de cette section et sauf indication contraire, nous appelons un *cluster-manager* un cluster physique.



(a) (b)

Figure 4.6 : (a) Regroupement des nœuds cc-NUMA en clusters logiques selon la notion de voisinage définit par la structure de la DQDT. (b) L'arbre DQDT représentant ce regroupement.

Dans notre structure DQDT, les feuilles et les nœuds sont décrits par une même structure de données *dqdt_cluster_s*. Autre qu'un pointeur vers le nœud père et ses quatre nœuds fils, cette structure de données contient un tableau de 5 entrées. Chaque entrée est une structure de données *dqdt_estimation_s* contenant différent estimations de ressources disponibles dans un cluster logique à un instant donné. Les champs de cette structure sont les suivants : le nombre de pages physiques libres *M*, le nombre de tâches actives *T* et le taux d'utilisation des cores *U*. Les valeurs contenues dans la 5ème entrée dans ce tableau sont en relation avec les valeurs des champs respectifs des quatre premières entrées. il s'agit de la somme des nombres de pages physiques libres, la somme des nombres de tâches prêtes à s'exécuter et le taux de l'utilisation des cores. La figure 4.7 illustre la structure de chaque nœud/feuille de l'arbre DQDT représentant le mesh de 16 nœuds cc-NUMA.

Un cluster logique $C^i(x,y)$ de niveau *i* contient donc une estimation de l'utilisation des ressources mémoire et cores pour l'ensemble du cluster (la 5ème case) ainsi qu'une estimation de l'utilisation des ressources pour chacun de ses clusters fils (les 4 premières cases). Pour les clusters physiques (feuilles de l'arbre) les informations inscrites dans les 4 premières cases du tableau de structures *dqdt_estimations_s*, correspondent à des estimations de l'utilisation de ressource cores (obtenues à partir des *core-managers* respectifs). Le nombre de pages physiques libres n'est pas utilisé (mis à zéro) dans ces entrées. En revanche, la 5ème case de ce tableau correspond à la somme de nombre de tâches prêtes à s'exécuter, à la moyenne des taux d'utilisation des cores, et au nombre de pages physiques libres dans ce nœud (obtenu à partir du *memory-manager*).

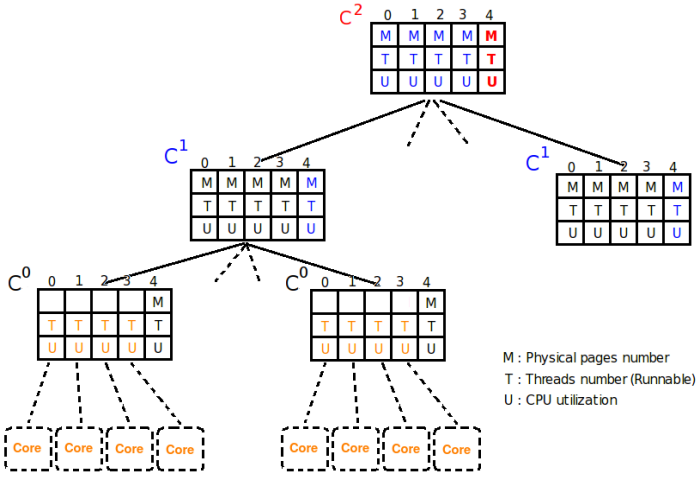


Figure 4.7 : Illustration de la structure de chaque nœud/feuille de l'arbre DQDT. La présence des cores en pointillé permet de faire le lien avec les indicateurs U dans les clusters physiques.

Grâce aux indicateurs de ressources présents dans chaque nœud/feuille de l'arbre DQDT, le noyau est capable d'orienter ses recherches à partir d'un cluster $C^i(x,y)$ en effectuant un minimum d'accès mémoire distant : le noyau parcourt l'arbre soit vers le cluster supérieur $C^{i+1}(x/2,y/2)$, car les ressources recherchées ne peuvent pas être satisfaites au niveau courant, soit vers les niveaux inférieurs en suivant les estimations de chaque niveau jusqu'à l'aboutissement au cluster physique contenant les ressources recherchées. Ce parcours est en O(1) puisque la hauteur de l'arbre est au maximum[6] de 5. Afin de renforcer la localité des accès mémoire durant ce parcours, le noyau place les feuilles et les nœuds de l'arbre DQDT en respectant la notion de voisinage définie par celui-ci. Ainsi les feuilles de l'arbre sont placées dans leurs nœuds cc-NUMA respectifs. Le noyau place également chaque nœud de l'arbre, représentant un cluster logique, dans l'un de ses nœuds cc-NUMA.

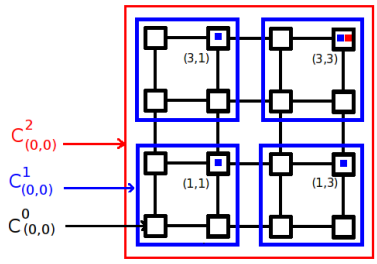


Figure 4.8 : Placement distribué des structures de données de la DQDT, représentant un mesh de 16 nœuds cc-NUMA, en respectant la notion de voisinage définie par celle-ci.

La figure 4.8 illustre ce placement. Les nœuds cc-NUMA de coordonnés (1,1), (1,3), (3,1) et (3,3) contiennent les quatre nœuds correspondant aux quatre clusters logiques de niveau 1 (marqués en bleu). Le nœud cc-NUMA de coordonnées (3,3) (marqué en rouge) contient la racine de l'arbre qui représente un cluster logique de niveau 2.

La structure DQDT ne contient pas de verrous d'aucune sorte. Sa topologie est invariante (il n'existe pas d'opérations de type suppression ou ajout d'un nœud). Concernant les estimations de disponibilité des ressources pour chaque cluster logique, il existe plusieurs lectures mais un unique écrivain. En effet, l'arbre DQDT est consulté, en lecture seule et en parallèle, par plusieurs instances du noyau cherchant à prendre une décision. Pour ce qui concene la mise à jour, le noyau place dans chaque nœud une tâche de fond responsable de la mise à jour périodique de l'ensemble des structures *dqdt_cluster_s* placés dans ce nœud. La période de mise à jour est un paramètre important puisque plus cette période est longue, plus les estimations en question deviennent obsolètes. Ce qui peut augmenter le nombre de mauvaises décisions. Par conséquent, nous avons choisi d'inclure dans cette mise à jour périodique uniquement les estimations de disponibilité de mémoire physique et l'usage des cores (les champs *M* et *U*). En effet, étant donné la quantité en mémoire physique disponible pour chaque nœud cc-NUMA dans les architectures d'aujourd'hui et la fréquence des allocations de pages physiques, une mise à jour périodique nous semble adéquate pour l'indicateur *M*. D'autre part, le taux d'usage d'un core, l'indicateur *U*, est intrinsèquement une information calculée d'une manière périodique. Le calcul du taux d'utilisation d'un core est abordé dans la section 4.3.4.

Concernant le nombre de tâches actives, l'indicateur *T*, il s'agit d'un compteur exact et non pas d'une estimation. Il indique le nombre de tâches (en attente, à l'état prêt ou en cours

6 Si nous faisons l'hypothèse que l'architecture matérielle sous-jacente dispose de 1024 nœuds. Cela représente 4096 cores si chaque nœud dispose de 4 cores.

d'exécution) appartenant aux applications utilisateur et assignées sur l'un des cores du cluster. Lorsque un core est sélectionné suite à l'aboutissement d'un parcours de l'arbre DQDT, les deux indicateurs *T* du cluster logique de niveau 0 (le premier est celui de la case correspondant au core et le deuxième est celui de la 5ème case) sont incrémentés d'une unité. Cette incrémentation atomique est ensuite propagée à tous les clusters logiques pères jusqu'au sommet de l'arbre. Le nombre d'incrémentations atomiques nécessaires pour mettre à jour l'arbre DQDT est au plus de *2 x H* où *H* est l'hauteur de l'arbre. Ce qui est équivalant à 10 incrémentations atomiques pour une configuration maximale de l'arbre. Lors de la terminaison d'une tâche utilisateur, une mise à jour de l'arbre est effectué afin de décrémenter les indicateurs T des clusters logiques respectifs de niveau 0 à *H*.

L'arbre DQDT est consulté à chaque fois qu'une instance du noyau cherche à prendre une décision dynamique concernant l'allocation d'une ressource. Cette prise de décision concerne l'allocation mémoire, le placement des tâches et l'équilibrage de charges entre les cores. Elle est basée sur un parcours de l'arbre DQDT en consultant l'un ou l'ensemble des indicateurs de ressources en fonction de l'événement à l'origine cette prise de décision et la nature de la ressource recherchée.

4.3.2 Allocation mémoire

Comme nous l'avons vu dans la section 4.2.1, l'allocation mémoire est assurée au niveau d'un nœud cc-NUMA par son *cluster-manager* à travers le composant *memory-manager*. Si la requête d'allocation mémoire ne peut êtres satisfaite localement, la DQDT est alors consultée afin de déterminer sur quel nœud la requête peut être effectuée. Prenons l'exemple de l'allocation d'une page physique. Pour une tâche localisée dans un cluster physique $C^0(x,y)$, le noyau cherche d'abord à satisfaire sa demande en ressource mémoire localement. En consultant la structure *dqdt_cluster_s* correspondant à son cluster physique, le noyau peut connaître le nombre de pages physiques actuellement disponibles dans ce cluster. En cas de manque de ressources locales dans le cluster $C^0(x,y)$, c'est-à-dire, si le nombre de pages physiques est en dessous du seuil de disponibilité, le noyau accède au niveau supérieur. Ce niveau de voisinage est représenté par la structure dqdt_cluster_s père directe, c'est-à-dire, le cluster logique $C^1(x/2,y/2)$. Le noyau peut donc décider si sa requête peut être satisfaite ou non dans ce cluster logique (grâce à la 5ème) et si oui depuis quel cluster physique (grâce aux 4 premières entrées). Si l'un des clusters physiques a été trouvé, le noyau réalise l'allocation mémoire depuis son *memory-manager*[7].

Le noyau peut aussi élargir son champs de recherche en augmentant le niveau de voisinage. Il s'agit d'un parcours en deux temps, ascendant puis descendant. Le parcours ascendant a deux conditions d'arrêt, à savoir, un cluster logique contenant suffisamment de ressources mémoire est trouvé, ou la racine de l'arbre est rencontrée et il ne contient pas les ressources recherchées. Cette dernière condition signifie qu'il n'y a plus de mémoire physique disponible dans l'ensemble des nœuds cc-NUMA. Durant son parcours ascendant, si une tâche trouve un cluster logique répondant à sa requête d'allocation, elle commence un parcours descendant en consultant, pour chaque cluster logique, les quatre premières cases estimations et en suivant le pointeur vers le cluster logique fils (de niveau inférieur) choisit (l'algorithme de choix est décrit plus loin). Ce parcours s'arrête quand le noyau atteint le cluster physique qui peut satisfaire la requête d'allocation. La figure 4.9 illustre ce parcours dans un DQDT complet (hauteur 5).

7 Un cluster physique est synonyme d'un *cluster-manager* comme il est indiqué dans une note précédente.

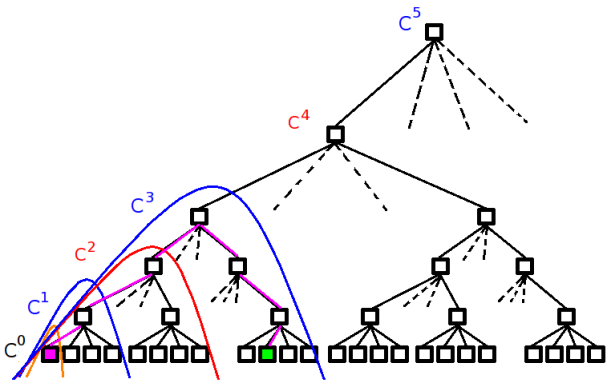


Figure 4.9 : Un exemple de parcours ascendant et descendant dans l'arbre DQDT ayant une configuration maximale. Le parcours ascendant s'arrête au cluster logique de niveau 3 ayant suffisamment de ressources.

Lors du parcours descendant, il peut y avoir, pour un cluster logique de niveau *i*, plusieurs clusters logiques de niveau *i-1* pouvant satisfaire une demande en mémoire physique. Un deuxième critère de sélection est alors employé, à savoir, la distance entre le nœud cc-NUMA originaire de la demande et le nœud cc-NUMA contenant la description du cluster logique de niveau *i-1* (c.f : la figure 4.8). Ainsi parmi les 4 clusters logiques de niveau *i-1*, le cluster logique choisi est celui ayant suffisamment de ressources mémoire et qui est le plus proche du nœud cc-NUMA originaire de la demande. La distance entre les nœuds cc-NUMA est une information dépendante de l'architecture matérielle sous-jacente et elle est mise à disposition du système d'exploitation via des APIs standards (p. ex : ACPI/SLIT - System Locality Information Table [120]).

Enfin, la prise de décision concernant le cluster physique cible contenant les ressources demandées est en O(1). En effet, le nombre de nœuds consultés est au plus *2 x H* où *H* est l'hauteur de l'arbre. Cela représente donc au plus 10 consultations différents dans une architecture matérielle ayant 1024 nœuds.

4.3.3 Placement des tâches

Le placement des tâches n'est pas en soit une fonctionnalité explicite dans les noyaux monolithiques d'aujourd'hui tels que Linux, BSD ou Solaris. En effet, lors de l'opération "création d'une tâche" la nouvelle tâche est affectée systématiquement au core sur lequel l'opération s'exécute. La stratégie d'équilibrage de charge fournie par l'ordonnanceur permet à posteriori de transférer l'exécution de la tâche sur un core cible plus adapté. Le choix du core cible est étroitement lié à la stratégie d'équilibrage de charge et elle diffère d'un noyau à un autre. En pratique, chacun de ces noyaux met en place sa propre API (non standard) afin de permettre au programmeur d'une application de déterminer l'ensemble de cores cibles à utiliser lors d'une migration. Cette approche, consistant d'abord à créer la tâche sur un core et ensuite à transférer son exécution vers un autre core, n'est pas adaptée pour une architecture NUMA pour les raisons que nous avons citées dans la section 2.4.

Dans le noyau d'ALMOS le placement des tâches est une fonctionnalité explicite, indépendante de l'ordonnanceur et assurée par la DQDT. Comme nous l'avons vue dans la section 4.2.3, notre approche de réalisation repartie du service système "création d'une tâche" nécessite une prise de décision concernant le choix du core cible avant même d'invoquer la création effective de la tâche. Dans le noyau d'ALMOS, le placement des tâches peut être vu comme une planification globale décentralisée permettant d'allouer les ressources de calcul. Le serveur d'ordonnancement par core, présenté dans la section 4.2.2, est responsable

uniquement d'appliquer les différentes politiques d'ordonnancement réclamées par les tâches qui lui sont affectées suite à cette planification. Comme le nombre de tâches actives dans le système peut excéder le nombre de cores et varie en cours du temps, une décision de migration peut être nécessaire afin d'équilibrer la charge entre les cores. Nous abordons l'équilibrage de charge entre les cores dans la section 4.3.4. Dans le restant de cette section nous introduisons nos différentes stratégies de placement que nous avons mis en place dans le noyau d'ALMOS.

Dans son principe, la prise de décision concernant le placement des tâches sur les cores est similaire à celle permettant l'allocation de mémoire physique. Dans le cas du placement de tâches la décision de sélectionner un cluster logique, lors des deux parcours ascendant et descendant de l'arbre DQDT, est une décision multicritères reposant principalement sur le nombre de tâches actives. Sous certaines conditions le taux d'utilisation des cores, la distance inter-nœuds et la disponibilité en mémoire physique sont également prise en compte. Avant de présenter nos stratégies de parcours de la DQDT dans les différentes situations conduisant le noyau à prendre une décision, nous classifions d'abord ces situations selon leur besoin de proximité avec les données déjà placées et en cours d'utilisation : (1) la création d'un thread utilisateur, (2) la bifurcation d'un processus utilisateur et (3) la création d'une application utilisateur.

1. Création d'un thread utilisateur

Les threads d'une application utilisateur sont sensés coopérer et doivent échanger des résultats intermédiaires entre-eux. Il est donc possible de déterminer un nœud optimal minimisant la distance avec le thread créateur. Deux questions se posent concernant le placement du nouveau thread à créer (nous allons utiliser indifféremment les termes thread et tâche). La première : à quel niveau logique consulté dans la phase ascendante du parcours le noyau doit s'arrêter ? La deuxième : le noyau peut-t-il accepter un core "proche" ayant un faible nombre de tâches actives, mais non oisif (sa charge n'est pas nulle) ? Imposer un niveau logique maximal implique d'interdire des cas de figure où plusieurs tâches d'une même application se retrouvent "très éloignées" les unes des autres. Bien que ce choix peut améliorer les performances en limitant la distance entre les tâches, il favorise la concurrence sur les cores si le nombre de tâches d'une application est supérieur au nombre de cores oisifs appartenant au cluster logique du niveau maximal imposé. Accepter un core "proche" non oisif avec un nombre "faible" de tâches actives, revient à dire, améliorer la localité des tâches au détriment des performances si l'exécution de la nouvelle tâche rentre en concurrence avec celles déjà affectées sur ce core. Des tâches "très éloignées", un core "proche" ou encore un nombre "faible" de tâches actives sont des qualifications approximatives qui dépendent du rapport entre localité et concurrence. Le placement est considéré réussi si l'exécution d'une tâche affectée à un core "proche" ne rentre pas en concurrence avec d'autre tâches potentiellement actives sur ce dernier. Autrement dit, le placement est considéré réussi si le serveur d'ordonnancement du core sélectionné ne réclame pas un équilibrage de charge par la suite.

Notre stratégie de placement d'un nouveau thread utilisateur consiste donc à minimiser le partage des cores (i.e. maximiser le parallélisme réel) tout en favorisant une proximité pour le nouveau thread par rapport au thread créateur. Le noyau effectue jusqu'à trois parcours de la DQDT et la décision du placement est obtenue dès le premier parcours réussie. Chaque tentative implémente une stratégie différente pour les parcours ascendant et descendant de l'arbre DQDT. Le niveau logique maximal imposé lors d'un parcours ascendant est un paramètre dont sa valeur par défaut est l'hauteur de l'arbre. Soit N_i , M_i , T_i , U_i , D_i et A_i sont

respectivement : Le nombre de cores appartenant au cluster logique de niveau *i*; le nombre de tâches actives du même cluster; le taux d'usage des cores du même cluster; la distance entre le nœud cc-NUMA contenant la structure dqdt_cluster_s décrivant le cluster logique de niveau *i* et le nœud cc-NUMA où se trouve la tâche créatrice; et une valeur aléatoire.

Le but de du premier parcours est de trouver un core proche oisif. Le parcours ascendant commence à partir du cluster logique de niveau 0 de la tâche créatrice. La condition d'arrêt est la suivante : $(T_i < N_i)$. Lors du parcours descendant, un cluster logique *i* peut avoir plusieurs clusters logiques de niveau *i-1* ayant $T_{i-1} < N_i$. Afin d'effectuer le meilleur choix, le noyau constitue un tableau trié où chaque case dans ce tableau contient un pointeur vers un cluster logique de niveau *i-1* éligible. Le tri[8] est effectué par ordre décroissant par rapport à la valeur $(N_{i-1}- T_{i-1})$, puis par ordre croissant par rapport à la valeur D_{i-1}. Le tri décroissant par rapport à la la valeur $(N_{i-1}- T_{i-1})$ permet d'occuper les cores d'un cluster logiques éligible avant de passer à un autre. Le noyau poursuit son parcours descendant en choisissant la premier case du tableau. En cas d'échec de ce parcours, le noyau utilise la case suivante en tant que point de départ d'un parcours descendant. Une fois un cluster logique de niveau 0 ayant $T_0 < 4$ a été trouvé par un parcours descendant, un core osif est alros sélectioné et une mise à jours du nombre de tâches actives est alors effectuée.

En cas d'échec du premier parcours, le deuxième parcours a pour objectif de trouver un core proche dont la charge est strictement inférieure à 100 % et ayant un faible nombre de tâches actives. La condition d'arrêt du parcours ascendant est la suivante : $(T_i < N_i)$ *OR* $(U_i < 100)$. Lors du parcours descendant à partir d'un cluster logique de niveau *i*, la condition d'éligibilité est la suivante : $(T_{i-1} < N_{i-1})$ *OR* $(U_{i-1} < 100)$ et le tri entre les clusters éligble est par rapport à la vleur U_{i-1}, puis par rapport à la vleur T_{i-1}, puis par rapport à la valeur D_{i-1}. Une fois un cluster logique de niveau 0 ayant $(T_0 < 4)$ ou $(U_0 < 100)$ a été trouvé par un parcours descendant, le core ayant la charge minmum dans ce cluster est alors sélectionné et une mise à jours du nombre de tâches actives est alors effectuée. Si ce deuxième parcours échoue, la troisième tentative est alors effectuée. Il s'agit d'effectuer la même recherche que celle de la deuxième tentative mais en acceptant une charge à 100 %.

2. Bifurcation d'un processus utilisateur

Dans le monde des systèmes d'exploitation respectant la norme POSIX, un processus peut bifurquer en faisant un appel système spécifique, *fork*. l'effet de cet appel système est de créer un processus fils partageant plusieurs ressources avec le processus ayant invoqué l'appel système (processus père). La stratégie de placement pour un processus fils doit prendre en compte la relation de partage avec son processus père mais elle reste moins forte que dans le cas d'une création d'un thread. Par conséquent, notre stratégie de placement concernant un processus fils est la même que celle d'un thread à l'exception des deux choix suivants :

- Le parcours ascendant, lors des trois tentatives, commence à partir du cluster logique de niveau 2 contenant le nœud cc-NUMA courant originaire de la demande de placement.

- Lors du parcours descendant de la première tentative, le tri des clusters logiques éligibles au niveau *i* est effectué selon un ordre croissant par rapport à la valeur T_{i-1}, puis par rapport à la valeur D_{i-1}.

8 le tri en question est multi-critères : le deuxième critère s'applique pour discriminer les valeurs égales suite à l'application du premier critère. Le troisième critère s'applique pour discriminer le résultat du tri du deuxième et du premier, ainsi de suite.

Le premier choix permet d'éviter de placer le processus fils dans le voisinage direct du nœud cc-NUMA courant où le processus père s'exécute puisque les deux processus peuvent à leurs tours créer leurs propres threads. Le deuxième choix permet de favoriser la sélection d'un cluster logique ayant le minimum de tâches actives afin de favoriser le cas où le parcours descendant trouve un nœud cc-NUMA oisif ou le moins chargé.

3. Création d'une application utilisateur

Dans le monde des systèmes d'exploitation respectant la norme POSIX, le lancement d'une nouvelle application est effectué à travers l'appel système *exec*. Cet appel système transforme le processus appelant en lui donnant un nouvel espace d'adressage virtuel. Par conséquent, la nouvelle application a un faible degré de partage, en termes de structures noyau, avec l'application créatrice. Notre stratégie de placement dans ce cas de figure est la même que celle d'un placement d'un thread à l'exception des choix suivants :

- Absence de parcours ascendant de l'arbre DQDT. La recherche, lors des trois tentatives, commence directement par un parcours descendant partant de la racine de l'arbre.

- Lors de la première tentative, la condition d'éligibilité d'un cluster logique de niveau *i-1* est la suivante : $(T_{i-1} < N_{i-1})$ *AND* $(M_{i-1} > M_{threshold})$.

- Le tri des clusters logiques éligibles au niveau *i* est effectué selon un ordre croissant par rapport à la valeur T_{i-1}, puis par rapport à la valeur A_{i-1}.

Le premier et le troisième choix visent à favoriser la sélection d'un nœud peu chargé en terme de nombre de tâches actives. La condition de sélection mentionnée dans le deuxième choix permet de favoriser un nœud cc-NUMA ayant suffisamment de ressources mémoire. Ce qui permet au noyau d'anticiper les besoins en mémoire physique pour répondre aux différentes allocations de structures de données et de pages physiques nécessaires pour démarrer une nouvelle application. La valeur $M_{threshold}$ est un paramètre ajustable par l'administrateur système. Sa valeur par défaut est égale à la moitié du nombre de pages physiques disponibles pour un nœud.

4.3.4 Équilibrage de charge

Le noyau est amener à équilibrer la charge entre les cores afin d'augmenter les performances moyennes du système en terme de nombre de tâches exécutées (terminées) par unité de temps. Dans un contexte généraliste où le noyau ne dispose d'aucune information préalable concernant les caractéristiques des tâches (p. ex : schéma de communication inter-tâches, durée d'exécution, etc) le noyau vise à tirer profit du parallélisme réel offert par un processeur many-core. Comme nous l'avons vu dans la section 4.3.3, les différentes stratégies de placement ont comme premier objectif de favoriser le placement d'une seule tâche par core. Toutefois, le noyau peut affecter plus d'une tâche à un core.

En effet, à un instant donné, le nombre total de tâches peut excéder le nombre de cores ou lors de la prise de décision les indicateurs de ressources (*M*, *T* ou *U*) n'étaient pas à jour. Les conséquences d'une telle affectation varies selon la nature des tâches affectées au même core. Si au moins deux tâches sont orientées calcul, alors l'impact sur le temps d'exécution de l'ensemble des tâches affectées au même core est plutôt grand. En revanche, si les différentes tâches sont plutôt interactives où une bonne partie de leurs temps d'exécution est passée en attente d'événements (p. ex : entrée/sortie, périodique, instrumentation, etc) alors leur

multiplexage temporel sur le même core peut avoir moins d'impact sur leurs temps d'exécution. Cela montre la nécessité de définir la charge d'un core.

Définir la charge d'un core n'est pas une question évidente. Une première idée est d'utiliser comme indicateur, le nombre de tâches affectées à un core. Ce nombre ne donne pas la charge réelle du core puisque il englobe toutes les tâches, même celles en attente sur des ressources système ou endormies sur un événement et donc ne s'exécuteront pas sur ce core avant un certain temps. Une deuxième idée consiste à utiliser le nombre de tâches prêtes à s'exécuter. Bien que ce nombre donne plus de précision sur la charge réelle d'un core que celui dénombrant toutes les tâches affectées à ce core, il reste insuffisant. En effet, un nombre variable de tâches applicatives peuvent être des tâches de fonds qui se réveillent de temps à autre afin de vérifier une certaine condition puis cèdent le core rapidement. Autrement dit, leurs contribution à la charge d'un core est faible, mais le noyau ne peut pas connaître leurs nombre et distribution par core à l'avance.

Le pourcentage d'utilisation d'un core donne, quant à lui, une indication précise sur la charge d'un core. En général, la fréquence de l'interruption horloge du système (tick) est de 60 Hz. Pour un core cadencé à 600 MHz, cet intervalle est de 10 millions cycles. Nous définissons la charge d'un core par la moyenne mobile exponentielle de son utilisation pendant N ticks où N est un paramètre ajustable (selon la configuration de l'architecture matérielle sous-jacente) et dont sa valeur par défaut est de 4. Les taux d'utilisation des cores d'un nœud cc-NUMA sont utilisées pour déterminer le taux d'utilisation (l'indicateur U) de son cluster logique de niveau 0. Cette dernière valeur est propagée dans la DQDT lors de la mise à jour périodique. Bien que le nombre de tâches prêtes à s'exécuter représente un meilleur indicateur de charge que celui des tâches actives, nous avons choisi le nombre de tâches actives (l'indicateur T) dans la DQDT. Ce choix représente un meilleur compromis entre le coût du maintient de la DQDT à jour et la précision dans la capture de la charge d'un core (le nombre de tâches prête est plutôt volatil).

Notre mécanisme d'équilibrage de charge est dynamique, décentralisé, coopératif et sans verrous. Il est déclenché à l'initiative d'un core trop chargé (initiateur). D'une manière périodique, chaque serveur d'ordonnancement vérifie si son core a besoin de déclencher une opération d'équilibrage de charge. Il s'agit d'une décision locale qui dépend des politiques d'ordonnancement appliquées par le serveur aux tâches qui lui sont affectées. Un exemple d'une telle condition est : $(U == 100)\ AND\ (T > 1)\ AND\ (R > 0)\ AND\ (U_H < 100)$ où R est le nombre de tâches utilisateur à l'état prêt et U_H est le taux d'utilisation moyen de tous les cores (l'indicateur U du nœud racine de l'arbre DQDT). La période d'équilibrage de charge peut être écourtée si la condition d'équilibrage de charge est satisfaite et $(U_2 < 100)$. La sélection d'une tâche à transférer (tâche victime) est également une décision locale selon les politiques d'ordonnancement appliquées par le serveur. Lorsque la condition d'équilibrage de charge est satisfaite et qu'une tâche victime est sélectionnée, le core initiateur consulte la DQDT afin de déterminer un core cible auquel la tâche victime sera transférée. Il s'agit de trouver un core proche oisif ou dont la charge est inférieure à 100 %. Cette consultation de l'arbre DQDT est la même que celle effectuée pour le placement d'un thread à l'exception des choix suivants :

- Lors du parcours descendant de la première tentative, le tri des clusters logiques éligibles au niveau i est effectué selon un ordre croissant par rapport à la valeur T_{i-1}, puis par rapport à la valeur D_{i-1}.

- En cas d'échec de la première tentative, le noyau se contente de rechercher un core proche ayant une charge inférieure à 100 %.

Lorsque le core cible a été déterminé, le core initiateur demande au core cible de récupérer la tâche victime en utilisant le mécanisme d'événements distants (c.f : section 4.2.3). L'idée est de laisser le core oisif, ou moins chargé, payer le coût du transfert de la tâche victime.

La figure 4.10 montre le déroulement réparti d'une opération d'équilibrage de charge effectuée d'une manière coordonnée entre un core initiateur *A* et un core cible *B*. À l'occurrence d'une période d'équilibrage de charge, le serveur d'ordonnancement du core *A* décide qu'il est nécessaire d'effectuer une opération d'équilibrage de charge dont la tâche en cours d'exécution *X* est la tâche victime. Après consultation de la DQDT, le core *B* est désigné en tant que cible du transfert. L'étape de la préparation du transfert consiste à sauvegarder le contexte d'exécution de la tâche *X*.

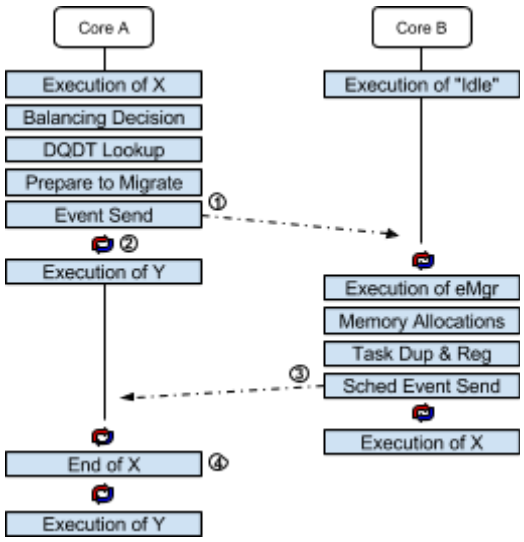


Figure 4.10 : déroulement réparti d'une opération d'équilibrage de charge.
Le core *A* décide de transférer la tâche *X* vers le core *B*.

La tâche *X* se met en attente passive (sleep) après avoir envoyé un événement (1) au core *B* afin de lui demander de réaliser le transfert. Cet événement prend en paramètre l'adresse d'une structure de données indiquant les informations nécessaires afin d'effectuer le transfert de la tâche *X*. En particulier, elle contient un drapeau permettant de positionner la valeur de retour du transfert. L'attente passive de la tâche *X* est un point d'ordonnancement (2) suit auquel la tâche *Y* à l'état prêt est alors élue et exécutée. Du côté du core *B*, la réception de l'événement conduit à l'exécution de la tâche dédiée au traitement d'événements pendants pour le core *B*. En réponse à l'événement envoyé par le core *A*, cette tâche réalise le transfert effectif de la tâche *X* en effectuant localement les allocations mémoire nécessaires (la pile noyau et le descripteur de la tâche). Une fois la duplication de la tâche *X* terminée, elle est enregistrée auprès du serveur d'ordonnancement du core *B* et une valeur de retour est positionnée pour indiquer le bon déroulement du transfert. En fin, une notification de réveil (3) distante (c.f : section 4.2.2) concernant la tâche X est envoyée au serveur d'ordonnancement du core *A*. À partir de ce moment, la tâche *X* poursuit son exécution sur le core *B* dès sa prochaine élection. Du côté du core *A*, la tâche *X* finit par être élue. À la reprise de son exécution elle vérifie d'abord si le transfert s'est bien passé, et dans ce cas là, elle met fin à son exécution (4). En cas d'échec du transfert, elle continue son exécution sur le core *A*.

L'équilibrage de charge à l'intérieur du même nœud cc-NUMA ne présente pas de difficulté particulière de point de vue de la localité des accès mémoire d'une tâche après une migration. En revanche, l'équilibrage de charge inter-nœuds pose plusieurs questions :

- Le noyau doit il migrer toutes les données d'une tâche qui a été déplacée ?

- Une tâche ayant été replacée dans un nouveau nœud cc-NUMA, doit elle rester dans ce nœud tant qu'elle n'a pas fini son exécution ou peut elle retourner à son nœud d'origine ?

- Une tâche ayant été déportée dans un nouveau nœud cc-NUMA, doit elle faire ses futures allocations mémoire sur son nouveau nœud ou sur son nœud d'origine ?

Il est difficile pour les noyaux monolithiques existants tels que Linux ou Solaris de migrer efficacement les données d'une tâche suite à une opération d'équilibrage de charge conduisant à un changement de nœud cc-NUMA. Ainsi, l'équilibrage de charge dans ces noyaux ne prend pas en compte la localité des accès mémoire. Les futures allocations mémoires sont réalisées sur le nouveau nœud cc-NUMA tandis que les données déjà allouées ne subissent aucune migration [80]. Cela cause une dispersion de données sur plusieurs nœuds. Cette dispersion se traduit par une perte de localité concernant la tâche transférée et s'aggrave à chaque migration dans un nouveau nœud. Notre modèle hybride de tâches présenté dans la section 4.5.3 introduit une abstraction nécessaire, et manquante dans les noyaux existants, permettant au noyau de déterminer, avec précision, les pages physiques accédées par une tâche. Par conséquent, il devient possible de migrer les données d'une tâche après la migration de la tâche elle-même. Cette migration peut être effectuée, à la demande, lors de ses prochains accès mémoire. Cela permet de rétablir la localité des accès mémoire d'une tâche après son transfert dans un nouveau nœud. Nous revenons sur ce point dans la section 4.5.3 où notre modèle hybride de tâches est présenté.

4.3.5 Conclusion

La structure de données DQDT définit un mécanisme wait-free basé sur un ensemble d'indicateurs d'usage de ressources et intégrant une représentation hiérarchique multi-niveaux de la notion de voisinage entre ces ressources. Il permet au noyau une prise de décision dynamique, décentralisée et multi-critères concernant l'allocation mémoire, le placement d'une tâche et l'équilibrage de charge entre les cores. Lors d'une prise de décision, les indicateurs de ressources, présents dans chaque nœud/feuille de l'arbre DQDT, permettent au noyau d'orienter sa recherche vers un nœud cc-NUMA cible en évitant ainsi de visiter l'ensemble des nœuds. L'arbre DQDT est lui-même distribué sur les nœuds cc-NUMA tout en respectant les niveaux de voisinages définit par celui-ci. Cela permet d'une part, d'éviter toute centralisation lors des accès distants et d'autre part, de limiter la distance de ces accès à la fois lors d'une prise de décision et lors d'une mise à jours de l'arbre.

Le placement des tâches sur les nœuds cc-NUMA est une fonctionnalité explicite dans le noyau d'ALMOS. Elle est appelée tout au début d'une création de tâche. Le but étant de rendre les services du noyau, concernant la création de tâches, conscients du caractère NUMA de l'architecture matérielle. Pour valider ce mécanisme, nous avons proposé et implémenté différentes stratégies de placement selon la nature de la tâche à placer (thread, processus ou une nouvelle application). En s'appuyant sur la DQDT, le schéma d'ordonnancement distribué et le mécanisme d'événements, nous avons proposé un mécanisme d'équilibrage de charge dynamique, décentralisé, coopératif et sans verrous.

Dans nos différentes stratégies de placement et d'équilibrage de charge, nous n'avons pas fait d'hypothèses particulières sur la nature des applications ou le domaine de déploiement du noyau. En pratique, les différents paramètres d'ajustement tels que la période de mise à jour de l'arbre DQDT, la période de calcul de la moyenne d'utilisation d'un core, la période

d'équilibrage de charge et le choix d'une tâche victime pour une opération d'équilibrage de charge, sont à positionner en fonction du domaine applicatif et des caractéristiques de l'architecture matérielle cible. En fin, l'ensemble d'indicateurs de ressources (*M, T, U*) par cluster logique de l'arbre DQDT, peut être étendu afin d'affiner d'avantage le choix du nœud/core cible ou d'étendre l'usage de la DQDT pour la gestion d'autres types de ressources.

4.4 Localité des accès mémoire aux données

Renforcer la localité des accès mémoire d'une tâche dans une architecture cc-NUMA nécessite de placer conjointement la tâche et ses données dans le but d'augmenter la localité des accès mémoire de la tâche. Nous avons vu dans la section 4.3 comment le noyau d'ALMOS s'appuie sur la DQDT afin de garder une proximité entre les tâches d'une même application. Nous avons vu dans la section 4.2.1 que le noyau d'ALMOS met en place un Memory-Manager par nœud afin de répondre aux allocations mémoire réclamées par les tâches placées dans son nœud. Dans un noyau monolithique en mémoire partagée, le contrôle transparent du placement des données d'une tâche peut se faire à l'aide du mécanisme de mémoire virtuelle. Il s'agit de décider à chaque défaut de page, sur quel nœud une page physique doit être allouée. Dans cette section nous abordons la question de placement dynamique des données d'une application utilisateur. Plus précisément, nous présentons une nouvelle stratégie nommé "Auto-Next-Touch" permettant de migrer les pages physiques d'une application vers le nœud de la tâche qui les utilise. Cette migration est transparente vis-à-vis de l'application utilisateur et elle ne nécessite pas d'intervention de la part du programmeur.

4.4.1 First-Touch : problèmes et inconvénients

Dans la plupart des noyaux monolithiques existants, la politique d'allocation de pages physiques par défaut, est "First-Touch". Quand une tâche accède pour la première fois à une page dans l'espace virtuel de son processus, un défaut de page (une exception matérielle) se déclenche. Selon cette politique et en réponse à ce défaut de page, le noyau alloue une page physique sur le nœud où la tâche ayant provoqué l'exception s'exécute. En cas de pénurie de pages physiques libres localement, le noyau réalise l'allocation mémoire sur le premier nœud ayant une page physique libre se trouvant à proximité.

Bien que cette politique favorise la localité des accès mémoire d'une tâche, elle a l'inconvénient majeur de ne pas tenir compte de la phase séquentielle d'initialisation se trouvant dans la plupart des applications parallèles. Une application parallèle commence souvent par une phase d'initialisation séquentielle où une seule tâche exécute certain nombre d'allocations mémoire. Les structures de données ainsi allouées sont alors initialisées (p. ex : génération des valeurs ou lecture des données depuis le disque). Par conséquent, le noyau alloue, selon la politique "First-Touch" localement toutes les pages physiques nécessaire pour ces données. Une fois la phase d'initialisation terminée, les tâches sont alors créées pour traiter parallèlement ces données. Pour les tâches qui seront placés sur les cores du même nœud que celui du core exécutant la phase d'initialisation, les accès aux données sont locales. En revanche, toutes les autres tâches accéderont d'une manière distante à ces données ce qui génère un trafic distant subissant le facteur NUMA de l'architecture. Plus il y a de tâches exécutées par des cores différents, plus les performances se dégradent car (i) la distance par rapport au nœud ayant les données initiales augmente; et (ii) il y a plus de contention : sur l'interconnect, sur le cache ayant une copie des lignes accédées et sur le contrôleur du banc mémoire. Par conséquent, cette politique ne permet pas le passage à l'échelle.

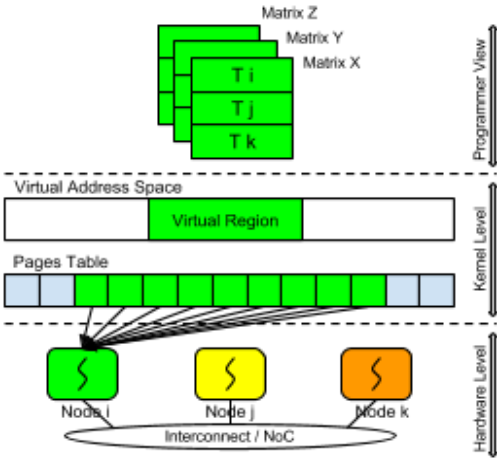


Figure 4.11 : conséquences de la stratégie “First-Touch”. Toutes les données initialisées se retrouvent dans un seul nœud, celui du thread initiateur *i*. Les deux autres threads (*j* et *k*) y accèdent à distance tout au long du traitement parallèle.

La figure 4.11 illustre les conséquences de la stratégie “First-Touch”. Il s’agit de l’exemple d’une application parallèle où le programmeur a décidé de traiter trois structures de données *x*, *y* et *z* par trois threads *i*, *j* et *k*. Ces threads ont été placés respectivement dans trois nœuds cc-NUMA différents. Bien que le programmeur a pris le soin d’assigner à chaque thread une partie différente de chacune de ses structures de données, les threads *j* et *k* accèdent à distance à leurs propres données tout au long de la phase du traitement parallèle. La raison de ces accès distants revient au choix du placement “First-Touch” (fait par le noyau) des pages physiques lors de la phase d'initialisation exécutée alors par le thread initial *i* (p. ex : celui qui exécute la fonction *main* dans une application c/c++). Lors de cette phase d’initialisation séquentielle, toutes les pages physiques ont été allouées localement sur le nœud *i* (en vert). Ce qui cause une contention sur le LLC et par conséquent, sur le contrôleur mémoire de ce nœud.

La stratégie “First-Touch” présente un autre problème lié également à l'existence d’une phase d'initialisation séquentielle. Il s’agit de la dispersion potentielle des allocations mémoire lors de la phase parallèle. En effet, les allocations mémoire faites lors de la phase de l'initialisation peuvent engendrer une pénurie de mémoire physique locale. Ce qui impacte les tâches créées par la suite et placées dans le même nœud que celui de la tâche initiatrice. Dans ce cas, le noyau alloue les pages physiques demandées par ces tâches sur d’autres nœuds. Ce qui impacte négativement leur localité d’accès à la mémoire.

4.4.2 Solutions existantes

Afin de donner aux programmeurs d’applications parallèles plus de contrôle sur le comportement du noyau, Linux propose des politiques d’affinité mémoire via ses APIs NUMA [20]. Il s’agit d’un certain nombre de services systèmes permettant au programmeur d’associer à une région virtuelle, un certain choix concernant la provenance des pages physiques : “First-Touch”, “Bind” et “Interleave”. Ces services ne sont pas normalisés POSIX et par conséquent, ils ne sont pas présents sur d’autres systèmes d’exploitation, ce qui impacte la portabilité des applications. Ces services nécessitent la modification des applications existantes et ils rajoutent plus de difficulté à la charge du programmeur. De plus, les performances des applications programmées en utilisant ces API sont liées à la machine sous-jacente, car le programmeur doit prendre en charge explicitement sa topologie matérielle.

Solaris, quant à lui, propose l'appel système "Next-Touch" [21,73]. Cet appel permet au programmeur d'indiquer au noyau d'une manière explicite, les zones mémoires (adresses virtuelles) qui sont sujet au traitement parallèle. Le noyau parcourt alors la table de pages du processus et marque chaque page physique, correspondante à la zone mémoire en question, pour une migration au prochain accès. Ainsi, lors de cet accès les pages physiques contenant les données à traiter par une tâche vont être migrées vers le nœud où la tâche est placée. Hormis le fait que cet appel système est non standard (i.e : non POSIX) et il n'est pas disponible sur la plus part des systèmes d'exploitation, l'usage de cette approche présente deux inconvénients. Le premier est une certaine complexité d'utilisation, car le programmeur doit considérer les adresses des zones mémoire concernées et faire les appels systèmes nécessaires. Le deuxième est la nécessité de modifier les applications existantes afin d'introduire les appels systèmes nécessaires (analyse de code existant). Toute fois, la stratégie "Next-Touch" ne résout pas le problème potentiel de dispersion des allocations mémoire dans la phase parallèle. Même si les tâches finiront par migrer leurs pages physiques depuis le nœud de la tâche initiatrice, l'ordre selon lequel les migrations les allocations de pages peut être difficilement prédictible ou estimable à l'avance.

4.4.3 Auto-Next-Touch

Nous proposons une nouvelle solution que nous nommons "Auto-Next-Touch". Elle permet d'atteindre les trois objectifs suivants :

- Décharger le programmeur d'indiquer au noyau les zones mémoire qui seront accédées dans la phase parallèle.

- Exécuter les applications parallèles existantes sans modifications préalables et préserver la portabilité des applications.

- Assurer une certaine disponibilité de mémoire physique dans le nœud de la tâche initiatrice pour les tâches d'exécution parallèle.

Notre solution repose sur deux techniques. La première vise à relocaliser, d'une manière transparente à l'application, les pages physiques dans les nœuds des tâches qui les utilisent. Il s'agit de détecter le moment où un processus séquentiel passe en mode multi-tâches. A ce moment, le noyau parcourt l'espace virtuel du processus et pour toute région virtuelle présente et éligible, le noyau consulte la table de pages du processus à la recherche de pages physiques déjà allouées à cette région virtuelle. Pour chaque page physique ainsi trouvée, le noyau marque l'entrée de la table de pages correspondante pour une migration au prochain accès.

Lors de la phase parallèle, le premier accès à une page marquée pour migration (page existante) provoque une exception. En réponse à cette exception, le noyau compare l'identifiant du nœud de la tâche ayant provoquée l'exception et l'identifiant du nœud auquel la page appartient. S'il s'agit du même nœud, le noyau réactive les droits d'accès à la page. Dans le cas contraire, le noyau effectue les actions suivantes : (i) allouer une nouvelle page physique sur le nœud local à la tâche en question; (ii) copier le contenu de la page existante dans la nouvelle page; (iii) remplacer la page existante par la nouvelle dans l'entrée correspondante de la table de pages; et (iv) libérer la page existante (uniquement si la région virtuelle, à laquelle l'accès mémoire a provoqué l'exception, est une région anonyme[9] privée). Dans les deux cas, l'exécution de la tâche est relancée après le traitement de

9 Une région virtuelle anonyme est une région qui ne projette pas le contenue d'un fichier (c.f : *mmap(2)*).

l'exception. Par conséquent, la tâche poursuit son exécution en accédant localement à la page. Les régions virtuelles éligibles pour l'application de la stratégie "Auto-Next-Touch" sont des régions de données privées au processus (la section 4.5.1 présente les différents types de régions virtuelles). Ainsi, une migration d'une page appartenant à de telles régions ne nécessite pas le parcours de l'ensemble de processus ayant potentiellement accès à la page en question pour effectuer une mise à jour coûteuse de leurs tables de pages.

La figure 4.12 illustre les conséquences de la stratégie "Auto-Next-Touch" appliquée par le noyau à la même application traitée dans la figure 4.11 où les conséquences de la stratégie "First-Touch" sont illustrées. Les threads *j* et *k* accèdent localement aux pages physiques contenant leurs parties respectives des structures de données *x*, *y* et *z*. Les pages physiques contenant la partie de données accédée par la tâche *i* ont conservées leur nœud d'origine *i*.

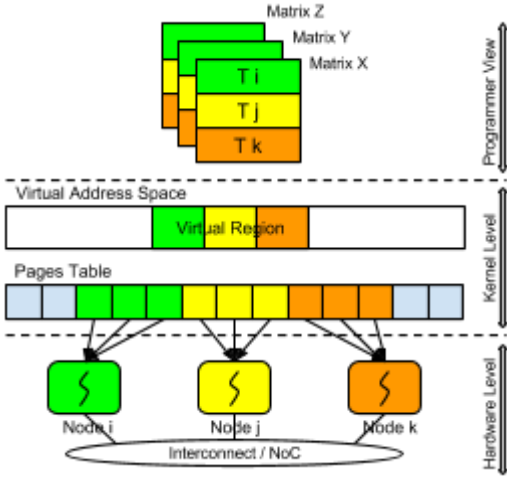


Figure 4.12 : conséquences de la stratégie "Auto-Next-Touch". Les pages contenant une partie des données initialisées ont été migrées par le noyau depuis le nœud *i* vers les deux autres nœuds *(j et k)* où se trouve les tâches qui les accèdent.

La deuxième technique vise à préserver une certaine disponibilité en mémoire physique dans le nœud de la tâche initiatrice. Il s'agit d'appliquer une restriction sur la quantité de mémoire physique maximale R_{max} qu'une tâche peut obtenir sur un nœud. Cette limitation ne s'applique que lorsque la tâche est la seule tâche de son processus (i.e : phase d'initialisation séquentielle). Cette restriction s'exprime comme un pourcentage maximal R_m de la mémoire physique disponible sur un nœud cc-NUMA M_{node}. Lors d'une allocation de page physique suite à un défaut de page concernant un processus séquentiel, le noyau vérifie d'abord le nombre de pages physiques libres dans le nœud local au processus. Si ce nombre est inférieur à $M_{node}\ (1 - R_m)$ alors le noyau réalise l'allocation mémoire depuis un autre nœud. Si aucun nœud n'a été trouvé alors la restriction est abandonnée et l'allocation mémoire est réalisée normalement depuis le nœud local où le processus s'exécute.

4.4.4 Discussion

La restriction imposée par le noyau lors des allocations mémoire d'un processus séquentiel[10], permet également de répartir la contention sur le nœud initial lors des migrations de pages. En réalité, le pourcentage R_m constitue un compromis entre quatre paramètres : (i) la disponibilité mémoire par nœud pour la phase d'exécution parallèle, (ii) le temps d'exécution de la phase d'initialisation séquentielle, (iii) le temps de migration de pages et (iv) la consommation énergétique liée au trafic inter-nœuds. Plus R_m est petit, plus la disponibilité

10 Nous appelons ici processus séquentiel, tout processus ne possédant qu'un seul thread.

mémoire par nœud augmente, plus le temps de migration de pages diminue (les pages à migrer sont réparti entre plusieurs nœuds), plus le temps d'exécution de la phase d'initialisation séquentielle augmente (les données à initialiser se trouvent dans des nœuds de plus en plus loin) et plus le trafic dans l'interconnect augmente (plus de contention et de consommation énergétique).

Bien que notre stratégie permette de remédier aux problèmes induits par la stratégie "First-Touch" concernant les applications parallèles, elle pénalise les applications séquentielles. Cette pénalisation se traduit en dégradation de performances liée à la dispersion de la localité des accès mémoire de l'unique tâche de l'application. L'inconvénient en question peut être évité en demandant explicitement au noyau de désactiver la restriction d'allocations mémoire en mettant la valeur R_m à zéro. Concrètement, l'implémentation de la fonction *fork*, vérifie d'abord la valeur d'une variable d'environnement spécifique (*ALMIX_ANXT_DISABLE* dans le cas d'ALMOS). Si cette variable est positionnée, alors le noyau désactive la stratégie "Auto-Next-Touch" pour le processus cible. Cette désactivation est conservée en cas d'exécution d'une application par la suite (*exec*).

4.4.5 Conclusion

Renforcer la localité des accès mémoire dans une architecture cc-NUMA, c'est faire en sorte que les accès mémoire effectués par une tâche qui s'exécute sur un core restent, en majeure partie locaux, c'est à dire dans le nœud où s'exécute la tâche, ou à proximité. Nous pensons que c'est au système d'exploitation d'effectuer dynamiquement le placement des données d'une application utilisateur et cela pour les raisons que nous avons cités dans la section 2.2. La stratégie "Auto-Next-Touch" s'inscrit dans cet objectif. Elle permet de relocaliser les pages physiques, contenant les données d'une application utilisateur, dans les nœuds des tâches qui y accèdent lors de la phase parallèle de l'exécution de cette application. La migration de pages physiques est transparente vis-à-vis de l'application utilisateur et elle ne nécessite pas d'intervention de la part du programmeur. La restriction des allocations mémoire dans la phase séquentielle exécutée lors du démarrage d'une application parallèle, permet principalement de garantir une certaine disponibilité en mémoire physique dans les nœuds en anticipant les besoins en mémoire lors de la phase parallèle.

Comme nous l'avons mentionné dans la section 4.3.4, l'équilibrage de charge cause une perte de la localité des accès mémoire d'une tâche suite à son transfert d'un nœud à un autre. Notre stratégie "Auto-Next-Touch" constitue une réponse partielle à ce problème. En effet, le noyau peut appliquer la migration automatique de pages, selon la même technique, lorsque l'exécution de la tâche a été transférée dans un autre nœud. Si le processus est séquentiel (mono-tâche), les défauts de pages provoqués par la modification de la table de pages sont majoritairement utiles, c'est-à-dire, ils donnent lieu à une migration de pages et par conséquent, ils permettent au noyau de renforcer la localité des accès mémoire du processus. En revanche, si le processus est multi-tâches, uniquement les défauts de pages, provoqués par la tâche ayant été migrée, sont utiles. Autrement dit, le coût de la migration des données d'une tâche après une opération d'équilibrage de charge est payé par l'ensemble des tâches du même processus et qui ne sont pas concernées par cette migration. Plus il y de tâches subissant d'opérations d'équilibrages de charge plus ce coût est amplifié et pénalisant l'ensemble de tâches et par conséquent, pénalisant les performances de l'application en terme de temps d'exécution. Nous revenons sur ce point dans la section 4.5.3 où nous conjuguons la migration automatique de pages avec notre modèle hybride de tâches afin de proposer une solution plus complète à ce problème.

4.5 Localité des accès mémoire liés aux miss des caches instruction et TLB

Le renforcement de la localité des accès mémoire des tâches ne doit pas concerner uniquement les accès aux données des applications utilisateur et du noyau. Le coût des accès mémoire au code et aux tables de pages subi également le facteur NUMA de l'architecture. Cela est valable pour les tâches d'une application utilisateur comme pour celles du noyau. Dans cette section, nous introduisons deux abstractions absentes dans les noyaux monolithiques existants tels que Linux, Solaris ou BSD. La première est la notion d'un réplica noyau. La deuxième est un nouveau concept de thread que nous nommons processus hybride. Notre motivation principale est d'atteindre les trois objectifs suivants :

- Renforcer la localité des accès mémoire liés aux miss instruction concernant le code du noyau et le code des applications utilisateur.

- Renforcer la localité des accès mémoire liés au traitement des miss TLBs (instruction et données).

- Permettre au noyau de contrôler l'accès aux pages physiques avec une granularité plus fine que celle des processus.

Les accès mémoire liés aux miss TLB peuvent être réalisées par le matériel (Table-Walk) comme dans les architectures de type Intel, AMD, ARM, PowerPC et UltraSPARC; ou par le noyau, comme dans les architectures de type MIPS et SPARC. Dans les deux cas, un miss TLB génère des accès mémoire sollicitant la hiérarchie de caches de l'architecture afin de retrouver l'information de traduction manquante. Par conséquent, quelque soit l'origine des accès mémoire causés par un miss TLB, la localité de ces accès mémoire restent un problème dans une architecture many-core cc-NUMA. Les informations de traduction sont organisées sous forme d'une structure de données en mémoire appelée table de pages. Le fait qu'une table de pages soit directe, comme dans la plupart des architectures matérielles (Intel, AMD, ARM) ou inversée (PowerPC, UltraSPARC, Itanium) n'impacte pas la faisabilité des solutions que nous allons présenter dans cette section. En effet, ces solutions concernent la gestion de l'espace d'adressage virtuel reposent sur des opérations existantes telles que le partage d'une région virtuelle entre processus ou d'avoir une région virtuelle privée par processus. Dans le restant de cette section, nous allons utiliser, à titre d'exemple, une table de pages directe à deux niveaux afin d'illustrer ces solutions.

Le restant de cette section est organisé comme suit. D'abord nous rappelons dans la sous-section 4.5.1, un certain nombre de notions nécessaires afin d'introduire nos solutions. Ensuite, nous présentons dans la sous-section 4.5.2 un premier concept : le réplica noyau. Il s'agit de répliquer le code et la table de pages du noyau dans le but de renforcer la localité des accès mémoire liés aux miss TLBs et instructions des threads de fond noyau. L'existence d'un réplica noyau ne permet pas de renforcer la localité des accès mémoire liés aux miss TLBs et aux instructions des threads d'une application utilisateur. Nous présentons dans la sous-section 4.5.3, notre deuxième concept : le processus hybride. Il s'agit d'un nouveau modèle de threads permettant de répliquer la table de pages et le code d'une application utilisateur dans le but de renforcer la localité des accès mémoire liés aux miss TLBs et aux instructions. Le concept de processus hybride et celui de réplica noyau permettent conjointement d'attendre nos trois objectifs que nous avons mentionnés plus haut. Nous discutons dans la sous-section 4.5.4 d'autres propriétés qui découlent de notre nouveau modèle hybride de threads avant de présenter notre conclusion dans la sous-section 4.5.6.

4.5.1 Introduction

Un processus est l'image d'un programme en cours d'exécution. Dans une architecture supportant la mémoire virtuelle, chaque processus possède son propre espace d'adressage virtuel. Dans cet espace d'adressage, une partie est dédiée pour projeter (mapper) les objets mémoire du noyau du système d'exploitation tandis que l'autre est dédiée pour mapper les objets mémoire de l'application utilisateur.

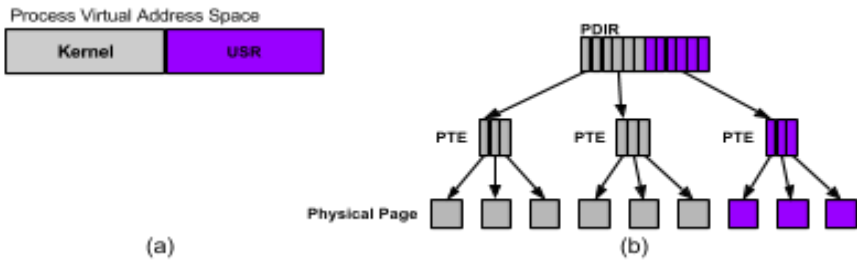


Figure 4.13 : (a) espace d'adressage virtuel d'un processus avec deux parties, une pour le noyau et l'autre pour l'application utilisateur. (b) la table de pages directe de deux niveaux correspondant.

La figure 4.13 (a) illustre un espace d'adressage virtuel d'un processus où la partie dédiée au noyau est en grise tandis que la partie dédiée à l'application utilisateur est en mauve. Chaque page virtuelle dans l'espace d'adressage d'un processus peut être associée (mappé) avec une page de la mémoire physique. La table de pages permet de décrire cette association ou mapping. La figure 4.13 (b) illustre une table de pages directe à deux niveaux. Chaque entrée de la table du premier niveau (PDIR) contient l'adresse physique d'une table de deuxième niveau. Chaque entrée d'une table du deuxième niveau (PTE) contient l'adresse d'une page de la mémoire physique. Dans la figure 4.13 comme pour toutes les figures suivantes, une table de deuxième niveau est confondue par le nom de ses entrées et donc désignée par l'acronyme PTE. La figure 4.13 (b) illustre le mapping de 6 pages physiques pour le noyau et 3 pages physiques pour le processus utilisateur.

La table de pages définissant l'espace virtuel d'un processus est une structure de données noyau. D'une part, elle est allouée et gérée par le noyau et d'autre part, l'accès par logiciel à cette structure se fait uniquement lors de l'exécution d'un code privilégié, celui du noyau. La table de pages elle-même est stockée dans un ensemble de pages physiques. La taille de la table de pages du premier niveau ainsi que celle d'une table de pages de deuxième niveau sont alignées sur un multiple de la taille d'une page. Lors d'un miss TLB d'un core, la table de pages est consultée afin de retrouver la PTE (information de traduction) manquante. Dans une architecture cc-NUMA, si les pages physiques accédées lors de cette consultation sont allouées sur le banc mémoire local au core, ces accès sont traités par le LLC (Last-Level Cache) local du nœud cc-NUMA. Sinon, il s'agit d'un accès distant et c'est au LLC du nœud cible de traiter ces accès. La figure 4.14 illustre l'espace d'adressage virtuel de deux processus *i* et *j*. Le noyau est présent dans les deux processus aux mêmes adresses virtuelles, c'est-à-dire, les deux processus partagent la même partie de leurs espaces d'adressage virtuel où le noyau est mappé.

Dans les noyaux monolithiques existants, la réalisation de ce partage se fait lors de l'opération *fork* où les entrée de la table de pages du premier niveau, relatives au noyau, sont copiées depuis la table du processus père vers la table de pages du premier niveau du

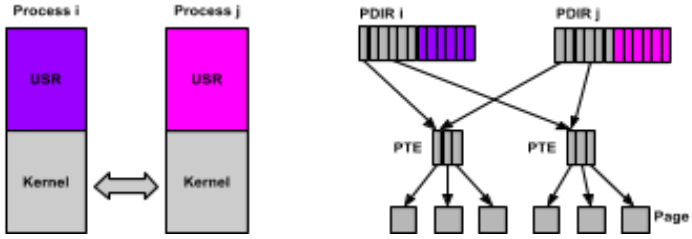


Figure 4.14 : Deux processus *i* et *j* exécutant deux applications utilisateur différentes mais ils ont le même mapping du noyau. Le noyau est présent dans les deux processus aux mêmes adresses virtuelles.

processus fils. Les valeurs de ces entrées sont conservées lors de l'opération *exec* pour lancer une nouvelle application dans la partie utilisateur de l'espace d'adressage virtuel du processus fils. Ainsi, le noyau est présent dans tous les processus actifs grâce à l'héritage successif du mapping du noyau depuis le premier processus utilisateur "init" crée par le noyau à la fin de la phase de démarrage.

Organisation de l'espace d'adressage virtuel

L'espace d'adressage virtuel d'un processus est organisé sous forme d'un ensemble de régions virtuelles. Une région virtuelle est une zone contiguë d'adresses virtuelles dont la taille doit être un multiple de celle de la plus petite page gérée par l'architecture matérielle sous-jacente. Les systèmes d'exploitation respectant la norme POSIX offrent le service *mmap* afin de permettre au programmeur d'une application utilisateur de se faire allouer une région virtuelle dans la partie utilisateur de l'espace d'adressage virtuel du processus exécutant son programme. Une région virtuelle peut être privée à un processus ou partagée entre processus. Une région virtuelle peut projeter le contenu d'un fichier (région nommée) ou simplement de la mémoire (région anonyme). Chaque région virtuelle est associée avec un mode d'accès, ou protection, qui est une combinaison des modes suivants : lecture seule, exécution seule et écriture seule.

Lors de l'opération *fork*, la partie utilisateur de l'espace d'adressage virtuel du processus appelant est dupliquée tandis que lors de l'opération *exec*, cette partie de l'espace d'adressage virtuel du processus appelant est réinitialisée. Suite à l'opération *exec*, la partie utilisateur de l'espace d'adressage virtuel contient, au moins, quatre régions virtuelles : une région pour le code ".text", une région pour les données initialisée ".data", une région pour le tas "heap" et une région pour la pile "stack". Le code utilisateur peut faire appel aux services *mmap* et *munmap* afin, respectivement, d'allouer et libérer d'autres régions virtuelles selon ses besoins.

La figure 4.15 illustre un espace d'adressage virtuel d'un processus à un instant donné. Dans cette figure, les régions présentes dans la partie noyau sont créées lors du démarrage du noyau et sont présentes dans ce processus par héritage de son processus créateur. Par conséquent, le code du noyau est partagé par l'ensemble de processus actifs dans le système. Dans la partie utilisateur, la région ".text + .rodata" est une région nommée dont le contenu est associé avec le fichier binaire (p. ex : format ELF) de l'application exécutée par ce processus. Le mode de protection associé avec cette région est "lecture + exécution". La région ".data" est une région nommée et privée. Elle est associée avec le même fichier binaire

de l'application exécutée avec un mode de protection en "lecture + écriture". Étant donnée que la région en question est privée, une écriture à une adresse virtuelle dans cette région ne modifie pas le contenu du fichier projeté mais elle déclenche, au niveau noyau, une opération de copie concernant la page accédée (mécanisme du Copy-On-Write [121,122]).

Figure 4.15 : Organisation de l'espace d'adressage virtuel d'un processus sous forme d'un ensemble de régions virtuelles.

Les noyaux monolithiques existants tels que Linux, Solaris et BSD, continuent (depuis l'époque de UNIX) à partager le code d'une application (ou d'une bibliothèque dynamique) entre tous les processus qui l'utilisent. En effet, les pages physiques contenant le code d'une application sont accédées en "lecture + exécution" par l'ensemble de processus exécutant cette application. De la même manière, le code d'une bibliothèque dynamique est présent en un seul exemplaire en mémoire mais il est accédé en "lecture + exécution" par l'ensemble de processus des différentes applications qui l'utilisent. La principale motivation pour maintenir une seule copie d'un code est d'économiser l'empreinte mémoire de l'ensemble de processus qui l'utilisent.

Gestion de l'espace d'adressage virtuel (partie utilisateur)

Dans un système d'exploitation monolithique, la gestion de la partie utilisateur de l'espace d'adressage virtuel d'un processus est effectuée par le noyau. Chaque région virtuelle, présente dans la partie utilisateur de l'espace d'adressage virtuel d'un processus, est représentée au niveau noyau par un descripteur de région. Un descripteur de région décrit l'ensemble de caractéristiques d'une région. Ces caractéristiques incluent : l'adresse virtuelle de début; l'adresse virtuelle de fin; la protection associée; la visibilité de la région (privée ou partagée); le cache de pages associé avec cette région (si cette dernière est nommée, ou si elle est anonyme et partagée); et le descripteur du fichier que la région projette (si cette dernière est nommée).

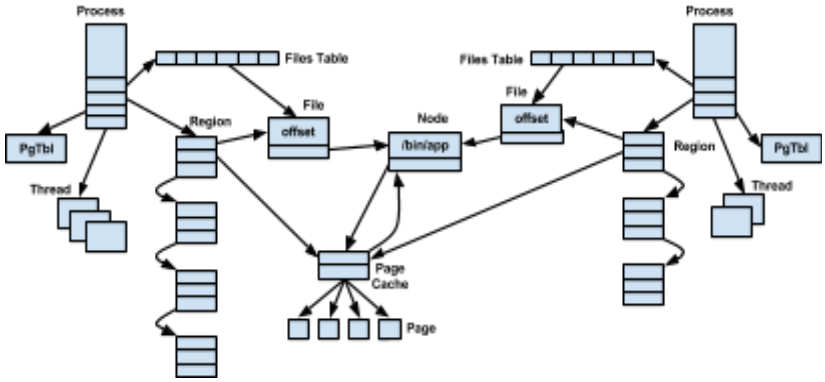


Figure 4.16 : Schéma illustrant les structures de données noyau impliquées dans la gestion de l'espace d'adressage virtuel des processus.

La figure 4.16 montre un schéma illustrant les principales structures de données noyau impliquées dans la définition et la gestion de l'espace d'adressage virtuel des processus dans les noyaux monolithiques existants. Les noms des structures de données peuvent varier d'un noyau à un autre mais le schéma général reste le même. Le noyau maintient une liste de

descripteurs de régions virtuelles par processus. Le noyau veille à ce que un processus (le code du programme utilisateur exécuté par le processus) accède à des régions autorisées. Ainsi, à chaque défaut de page le noyau doit retrouver le descripteur de la région contenant l'adresse virtuelle à l'origine du défaut de page. Si aucune région contenant l'adresse virtuelle en question n'a pas été retrouvée, le noyau génère le signal *SIGSEGV* afin d'informer le processus que son accès est illicite. Le traitement du défaut de de page dépend de la nature (nommée ou anonyme) et la visibilité de la région (privée ou partagée). Une fois la région recherchée a été retrouvée, le noyau exécute une méthode spécifique associée à cette région afin de traiter le défaut de page. Si la région est nommé (p. ex : la région du code ".text"), le cache de pages associée à la région est alors consulté afin de trouver l'image en mémoire de la page absente faisant partie du fichier projeté dans cette région. Une fois la résolution du défaut de page est effectuée, le noyau met à jour la table de pages du processus avant de relancer son exécution.

Notion d'un thread et d'un processus

Dans les systèmes d'exploitation en mémoire partagée, et plus particulièrement ceux à base de noyaux monolithiques tels que Linux, Solaris et BSD; un processus est vu en tant que conteneur de ressources. Parmi les ressources définies par un processus : l'espace d'adressage virtuel, les descripteurs de fichiers ouverts et l'état des signaux. Un thread quant à lui est vu en tant que fil d'exécution à l'intérieur du processus, autrement dit, une entité ordonnançable ayant son propre contexte d'exécution. La figure 4.17 illustre deux threads partageant l'espace d'adressage virtuel de leur processus.

Chaque thread dispose de sa propre pile qui lui est allouée, en pratique[11], en tant que région virtuelle. Dans les systèmes d'exploitation les plus répandus, tels que Windows NT/XP/2000, Linux, et Solaris (à partir de la version 9) les threads sont crées et ordonnancés directement par le noyau, c'est à dire, pour chaque thread utilisateur il existe un descripteur au niveau noyau. Ce descripteur contient, entres autres, le contexte d'exécution du thread. Le but étant de permettre au noyau de répartir les différent threads du même processus sur les cores et de tirer ainsi profit du parallélisme réal d'une architecture multi-cores. Selon cette approche, l'ensemble de descripteurs de threads d'un processus sont attachés au descripteur de leur processus comme il est illustré par la figure 4.16.

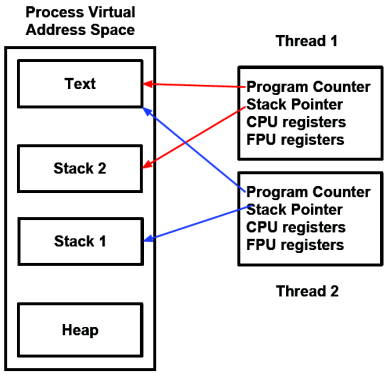


Figure 4.17 : La notion actuelle de threads en tant que fil d'exécutions qui partagent le même espace d'adressage virtuel de leur processus.

11 D'une part, sa taille est importante (de 64 KiB à 10 MiB selon les systèmes) et d'autre part, le noyau peut ainsi tenter d'expanser dynamiquement la pile en cas de besoin.

Observations et conclusion

De ce qui précède, nous en tirons les observations suivantes concernant les noyaux monolithiques existants :

1. Tours les processus partagent les informations de traductions (PTEs) du noyau par héritage lors de l'opération *fork*.

2. Tous les processus partagent le code du noyau, c'est-à-dire, ils accèdent aux même pages physiques qui mappent la région ".text + .rodata" du noyau.

3. Tous les processus exécutant un programme *P* réfèrent aux mêmes pages physiques contenant le code de ce programme *P*.

4. La représentation au niveau noyau d'un thread se limite aux informations permettant son ordonnancement.

5. L'ensemble de threads du même processus partagent le même espace d'adressage virtuel définie par le processus. Par conséquent, ils partagent la même table de pages du processus.

6. L'ensemble de threads du même processus partagent les pages physiques contenant le code du programme exécuté par le processus.

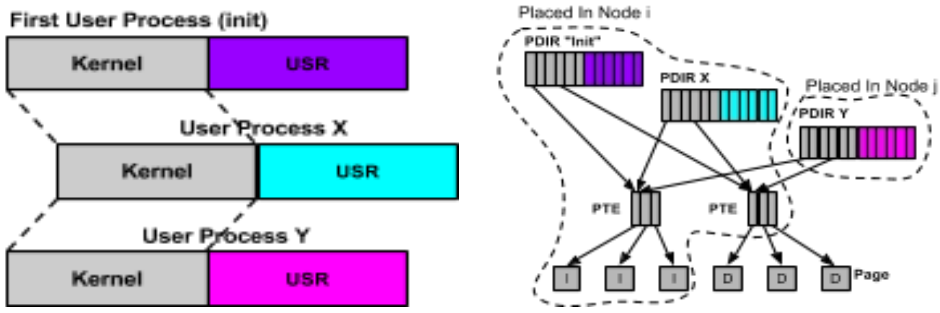
7. L'ensemble de threads de tous les processus partagent les pages physiques contenant le code du noyau.

Les noyaux monolithiques existants, tels que Linux, BSD et Solaris, ne traitent pas la question de la localité des accès mémoire aux codes et tables de pages des applications utilisateur ou du noyau. Dans un many-core ayant plusieurs centaines de cores, la présence d'une seul copie d'un code et d'une seule table de pages par processus peut devenir rapidement un goulot d'étranglement pour une application parallèle à fort degré de parallélisme de threads. D'autre part, le trafic engendré par les accès distants lors des miss instruction et TLBs de l'ensemble de cores exécutant les threads de cet unique processus, augmentent la contention sur l'interconnect et la consommation électrique liée au mouvement de données. En fin, la représentation d'un thread au niveau noyau se limite sur l'aspect ordonnancement et ne permet pas au noyau de localiser efficacement l'ensemble de pages physiques accédées par un thread donné. Comme nous l'avons abordé dans la section 4.3.4, ce manque de connaissance est à l'origine de la dispersion de la localité des accès mémoire d'un thread suite à une opération d'équilibrage de charge impliquant un changement de nœud.

4.5.2 Réplica Noyau

Dans un noyau monolithique existants tels que Linux, BSD et Solaris; le noyau est présent dans chaque processus et il est mappé à l'identique dans l'espace d'adressage virtuel de chaque processus. La figure 4.18 (a) illustre l'héritage successif de ce mapping depuis le premier processus dans le système *"init"* crée à la fin de la phase de démarrage. Dans ces

noyaux, il n'existe qu'une seule copie de code noyau. Bien que chaque processus dispose de son propre espace d'adressage, uniquement la table de page de premier niveau est propre à chaque processus tandis que les pages physiques contenant les PTEs du noyau sont partagées par l'ensemble de processus dans le système. Dans une architecture cc-NUMA, le partages des PTEs du noyau entre tous les processus ainsi que la présence d'une seule copie de code noyau posent problème quand à leurs placement comme il est illustré par la figure 4.18 (b).



(a) (b)

Figure 4.18 : Placement du code et tables de pages du noyau. (a) Le mapping du noyau est hérité successivement lors des opérations *fork*. (b) Le code et les tables de pages du noyau sont locales pour les processus *"init"* et *"X"* mais distants pour le processus *"Y"*.

Dans le but de rendre les accès mémoire au code et aux informations de traduction du noyau, locales à chaque nœud cc-NUMA, nous proposons de les répliquer dans chaque nœud. Selon notre approche, lors de la phase de démarrage, le noyau se réplique dans chaque nœud en créant un processus particulier appelé "Réplica Noyau". Chaque réplica dispose de son propre espace d'adressage virtuel, de sa propre copie du code noyau et de ses propres tables de pages. Dans cette approche, le noyau continue à être mappé à l'identique dans chaque réplica mais ce mapping est partitionné en deux régions virtuelles : locale et globale. Pour un réplica donné, sa région locale réfère à des pages physiques allouées localement sur son nœud. Dans le noyau d'ALMOS, la région locale d'un réplica projette la copie locale du code et des données en lecture seule du noyau (".text" + ".rodata"). Ainsi, à la même adresse virtuelle, appartenant à une région locale, peuvent correspondre différentes adresses physiques et cela en fonction de l'espace d'adressage virtuel utilisé, c'est à dire, en fonction du réplica. La région globale quant à elle, permet d'accéder à l'ensemble de pages physiques partagées contenant les structures de données du noyau (allouées statiquement ou dynamiquement). Une adresse virtuelle dans cette région est mappé avec une seule et unique adresse physique quelque soit le réplica.

La figure 4.19 (a) illustre l'organisation de l'espace d'adressage virtuel de deux réplicas noyau ceux des nœuds *i* et *j*. Le mapping du noyau tels qu'il est défini par le réplica d'un nœud est copié dans tous les processus crées dans ce nœud par héritage successif comme il est illustré également par la figure 4.19 (a). Lorsque le noyau décide de migrer un processus d'un nœud à un autre suite à une opération d'équilibrage de charge, le mapping noyau hérité depuis le réplica du noyau d'origine est remplacé par celui du réplica du nœud cible. La figure 4.19 (b) illustre la mise en ouvre, au niveau des tables de pages, de l'organisation en deux régions (locale et globale) de la partie noyau de l'espace d'adressage virtuel des processus. Ainsi, les tables de pages et le code du noyau sont répliqués dans chaque nœud cc-NUMA. Cette réplication ne présente pas un éventuel surcoût lié au maintient de la cohérence entre les réplicas, car généralement ils ne sont pas modifiés en cours de l'exécution du noyau après son démarrage.

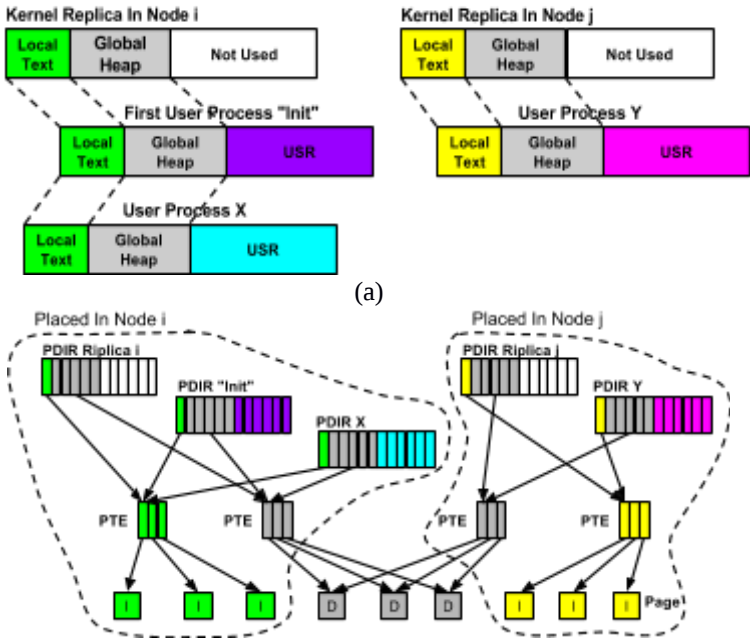


Figure 4.19 : (a) l'organisation de l'espace d'adressage virtuel d'un réplica noyau; (b) les tables de pages ainsi que le code du noyau sont répliqués localement.

Nous pouvons qualifier les miss instructions et TLBs en deux catégories. La première, concerne le noyau puisque les miss en question résultent de l'exécution du code noyau et des accès mémoire faits par ce code aux différentes structures de données noyau ou utilisateur. La deuxième catégorie concerne les applications utilisateurs puisque les miss en question résultent de l'exécution du code d'une application utilisateur et des accès mémoire faits par ce code aux structures de données utilisateur. Nous nommons les miss de la première catégorie miss noyau et les miss de la deuxième catégorie miss utilisateur. La réplication du code et tables de pages du noyau permet de renforcer uniquement la localité des accès mémoire liés aux miss noyau.

Il existe deux sources pour les miss noyau, la première est l'exécution d'un thread de fond noyau tandis que la deuxième est l'exécution, en mode noyau, d'un thread utilisateur (suite à un appel système ou un événement matériel). Les threads noyau, placé dans un nœud cc-NUMA donné, sont attachés au réplica noyau de ce nœud et ils partagent l'espace d'adressage virtuel de leur réplica. Par conséquent, les accès mémoire liés aux miss noyau générés par ces threads sont locaux. En revanche, les accès mémoire liés aux miss noyau générés par l'exécution d'un thread utilisateur en mode noyau sont locaux uniquement si le thread est placé dans le même nœud que celui de son processus, c'est à dire, dans le même nœud que celui du réplica noyau depuis lequel le processus du thread en question a hérité le mapping du noyau. Ceci est vrai notamment pour les architectures matérielles dont les miss TLB sont traités par le matériel (table walk) comme c'est le cas dans les architectures de type Intel, AMD et ARM.

Dans ces architectures, la MMU d'un core dispose d'un registre qui indique l'adresse mémoire physique de la table de page définissant l'espace d'adressage virtuel du code exécuté. Lors du chargement d'un thread sur un core, c'est l'adresse de la table de pages de son processus qui est chargée dans ce registre. Ainsi, le core qui exécute un thread, réfère systématiquement à la table de pages de son processus lors des miss TLBs. Par conséquent, les miss noyau générés par un thread utilisateur en mode noyau sont distants pour tout thread placé dans un nœud différent de celui de son processus. Concernant les architectures dont les

miss TLB sont traités par le noyau, il existe une plus grande flexibilité. Ainsi, le noyau peut consulter la table de pages du réplica du nœud où se trouve le thread ayant causé le miss TLB et par conséquent, les accès mémoire liés aux miss noyau deviennent des accès locaux.

Nous revenons sur la question de la localité des miss noyau générés par un thread utilisateur en mode noyau dans la section 4.5.3 où nous présentons le concept de processus hybride. Conjugué avec la mise en place d'un réplica noyau par nœud cc-NUMA, le concept de processus hybride permet de rendre les accès mémoire, liés aux miss noyau, locaux quelque soit le placement du thread utilisateur.

4.5.3 Processus Hybride

Dans les noyaux monolithiques existants tels que Linux, BSD et Solaris; il existe deux abstractions permettant de décrire le parallélisme d'une application, à savoir, le processus et le thread. Nous pensons que l'abstraction de thread telle qu'elle est définie dans ces noyaux n'est intrinsèquement pas adaptée pour les architectures de type many-core cc-NUMA :

1. Les threads partagent le code et la table de pages de leur processus. Par conséquent, le trafic généré par les accès mémoire liés aux miss instruction et TLBs lors de l'exécution d'une application à fort parallélisme de threads, sont dans leur écrasante majorité des accès distants pouvant : (i) provoquer un goulot d'étranglement dans leur traitement; (ii) accroître la contention sur l'interconnect de communication; et (iii) augmenter la consommation électrique liée au mouvement des données.

2. La représentation d'un thread au niveau noyau est limitée, essentiellement, à son aspect d'ordonnancement sans aucun moyen permettant au noyau de faire un lien entre un thread et l'ensemble de pages physiques qu'il utilise. Ceci constitue un handicape pour une approche de gestion transparente, au niveau noyau, de la localité des accès mémoire des threads.

L'abstraction de processus quant à elle, est suffisante pour répondre aux deux limitations que nous venons de soulever. En effet, l'abstraction de processus définit un espace d'adressage virtuel propre à chaque processus. Bien que ce ne soit pas fait par les noyaux monolithiques existants, le noyau pourrait renforcer la localité des accès mémoire liés aux miss instruction et TLBs en allouant localement l'ensemble de pages physiques constituant le code et la table de pages du processus et de migrer ces pages lors d'un changement de nœud cc-NUMA (p. ex : une opération d'équilibrage de charge). Le noyau peut effectivement faire le lien entre un processus et l'ensemble de pages physiques qu'il utilise grâce à la table de pages du processus (p. ex : la stratégie d'affinité mémoire Auto-Next-Touch que nous avons présentée dans la section 4.4.3). Malheureusement, décrire une application parallèle en tant qu'ensemble de processus coopérants est une tâche relativement lourde, car elle nécessite une gestion explicite par le programmeur des moyens de communication inter-processus. C'est essentiellement pour cette raison que la notion de thread a été introduite.

Le concept de processus hybride, que nous avons mis en place dans le système d'exploitation ALMOS, introduit une nouvelle abstraction intermédiaire entre la notion de processus et la notion de thread. Nous définissons un processus hybride en tant que fil d'exécution ayant son propre espace d'adressage virtuel. De point de vue du programmeur d'une application parallèle, les processus hybrides sont des threads respectant le standard PThtread. En particulier, la communication inter-processus hybrides rattachés au même processus peut être effectuée directement en mémoire partagée via des variables globales allouées statiquement ou dynamiquement. De point de vue du noyau, le lancement d'une nouvelle application via

l'appel système *exec*, donne lieu à la création du premier processus hybride. Ce premier processus hybride est attaché au nouveau processus et il constitue le premier fil d'exécution du processus.

L'appel système réalisant la fonction *pthread_create* déclenche, au niveau noyau, la création d'un nouveau processus hybride qui partage, tout comme un thread classique, la table des fichiers ouverts, le masque des signaux du processus, etc. La seule différence réside dans le fait qu'un processus hybride dispose de son propre espace d'adressage virtuel dont la partie noyau est organisée comme nous l'avons présenté dans la section 4.5.2 et illustré par la figure 4.19 (a). La partie utilisateur de l'espace d'adressage virtuel d'un processus hybride, est elle-même partitionnée logiquement par le noyau en trois parties : privée, locale et globale. Dans le restant de cette section nous considérons uniquement la gestion et l'organisation de la partie utilisateur de l'espace d'adressage virtuel d'un processus hybride. Ainsi, sauf indication contraire, l'espace d'adressage virtuel d'un processus (normal ou hybride) désigne la partie utilisateur de cet espace d'adressage.

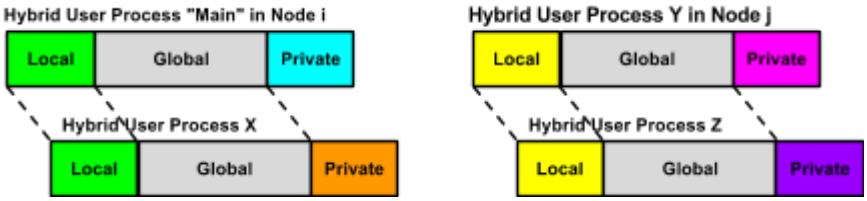


Figure 4.20 : La partie locale est partagée entre l'ensemble de processus hybrides qui sont placés dans le même nœud. La partie globale est partagée par l'ensemble de processus hybrides du même processus. La partie privée est expliquée par son nom.

Le partitionnement de l'espace d'adressage virtuel est réalisé à des adresses virtuelles fixes et identiques pour tous les processus hybrides crées et attachés au même processus. Le noyau gère les trois parties de l'espace d'adressage virtuel d'un processus hybride d'une manière transparente au programmeur de l'application. La figure 4.20 illustre ce partitionnement où pour chaque partie le noyau attache un niveau de visibilité différent vis-à-vis l'ensemble de processus hybride du même processus.

La partie globale est, par construction, visible par l'ensemble des processus hybrides du même processus, c'est à dire, le noyau mappe à l'identique les adresses virtuelles de cette partie dans l'espace d'adressage virtuel de chaque processus hybride. C'est grâce à cette visibilité globale que le noyau assure l'illusion d'un unique espace d'adressage virtuel partagé entre les fils d'exécution d'une application parallèle conformément au standard PThread. Concrètement, la section ".data" du programme, lancé par l'appel système *exec*, est mappé dans la partie globale. Le tas du processus évolue également dans cette partie tout comme les projections demandées par le programme via l'appel système *mmap*.

La partie locale n'est visible que par les processus hybrides placés dans le même nœud cc-NUMA. Le noyau attache à ce niveau de visibilité, local à un nœud, un mécanisme de réplication à la demande du code de l'application. Ainsi, le code de l'application utilisateur (section ".text" + ".rodata" dans le cas d'ALMOS) est mappé dans cette partie et il est répliqué à fur et à mesure que les fils d'exécution (représenté par les processus hybrides) réclament des pages physiques de code. Cette réplication à la demande s'appuie sur le mécanisme de résolution des défauts de pages. Ainsi, lorsque un processus hybride subit un défaut de page appartenant à la région locale, le cache de pages du fichier binaire de l'application est consulté. Si la page recherchée est présente en mémoire mais sur un nœud cc-NUMA différent de celui du processus hybride subissant le défaut de page, alors le noyau

duplique cette page en utilisant une nouvelle allouée sur le nœud local au processus hybride demandeur. En résultat de ce défaut de page, le noyau met à jour la table de pages du processus hybride en utilisant la nouvelle copie avant de résumer l'exécution de ce processus. Si la page est absente du cache, le noyau alimente le cache de pages en utilisant une page allouée localement sur le même nœud que celui du processus hybride, avant de l'utiliser pour mettre à jour la table de pages du processus en question.

Comme son nom l'indique, la partie privée de l'espace d'adressage virtuel d'un processus hybride n'est visible que par les instructions exécutées par ce fil d'exécution. Typiquement, le noyau mappe la pile du processus hybride dans cette partie privée de son espace d'adressage virtuel. D'une part, la pile peut ainsi évoluer dynamiquement sans un surcoût de gestion, car elle ne peut pas se chevaucher avec les autres régions virtuelles du tas ou avec les autres piles des processus hybride du même processus. D'autre part, la taille de la pile d'un processus hybride peut être importante (centaines de mégaoctets) quelque soit le nombre de ces processus par application et cela même sur une architecture 32 bits où l'espace d'adressage virtuel est limité à 4 Go. Nous revenons, plus loin, sur l'usage potentiel de cette région privée par processus hybride et les conséquences qui en découlent.

Notre conception du processus hybride permet au noyau de renforcer la localité des accès mémoire liés aux miss TLBs et instructions. La figure 4.21 illustre comment le noyau peut réaliser l'organisation en trois parties (privée, locale et globale) des quatre processus hybrides placés dans les nœuds *i* et *j* et illustrés dans la figure précédente 4.20. Chaque processus hybride dispose de sa propre table de pages de premier niveau que le noyau alloue localement

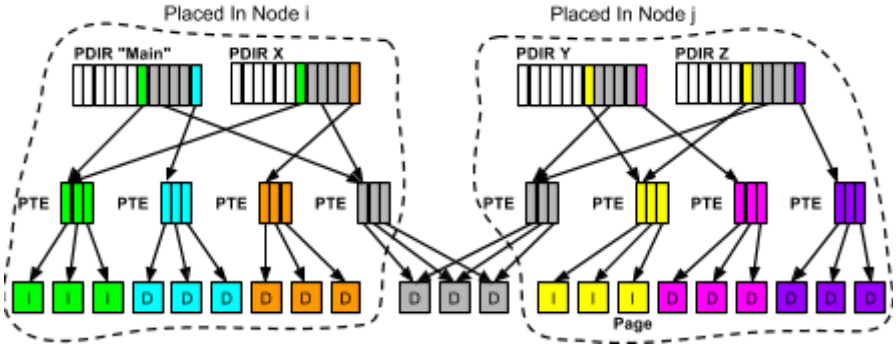


Figure 4.21 : L'organisation des tables de pages des quatre processus hybrides de la figure 4.20.

sur le même nœud cc-NUMA où le processus est placés. Le noyau réplique les PTEs de la partie globale (en gris) et maintient leur cohérence. Ces PTEs pointent sur les pages physiques de données partagées globalement entre l'ensemble de processus hybrides placés dans le même nœud et appartenant à la même application. Les PTEs de la partie locale (en vert pour le nœud *i* et en jaune pour le nœud *j*) pointent sur les pages physiques contenant les réplicas du code de l'application. Le noyau les partage entre l'ensemble des processus hybrides de même application qui sont placés dans le même nœud. Le noyau peut ne pas maintenir la cohérence entre les PTEs des différentes parties locales de différents nœuds, s'il n'autorise pas le code de l'application à modifier les droits d'accès à ses propres pages de code. Généralement, c'est effectivement le cas puisque la section de code n'a pas été mappée explicitement par le code de l'application. Par conséquent, le service système *mprotect* (qui permet à un processus de modifier lui-même les droits d'accès à certaines régions de son espace d'adressage virtuel) sera donc restreint s'il porte sur une région de type "local".

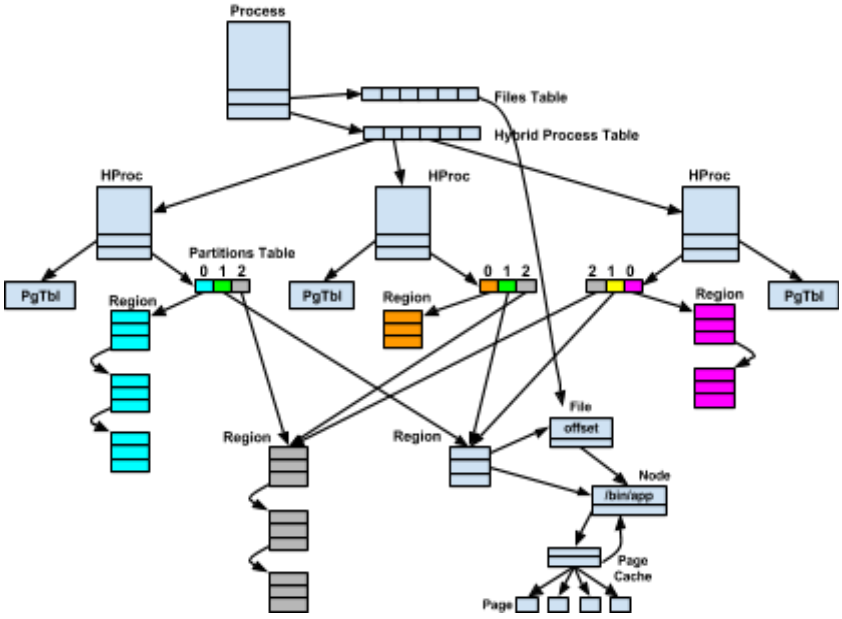


Figure 4.22 : Les principales structures de données qu'un noyau monolithique peut mettre en place afin d'implémenter notre concept de processus hybrides.

Du point de vue implémentation du noyau, la figure 4.22 illustre une nouvelle organisation des structures de données liées à la gestion de l'espace d'adressage virtuel d'un processus dans un noyau monolithiques intégrant l'abstraction des processus hybrides. En comparaison avec le schéma décrivant la gestion de l'espace d'adressage virtuel d'un processus dans les noyaux monolithiques existants illustrée par la figure 4.16, nous pouvons noter les trois points suivants :

- La table de threads du processus est remplacée par une table de processus hybrides. Le processus continue à être un conteneur de ressources partagées (p. ex : la table des fichiers ouverts).

- La table de page n'existe plus en un seul exemplaire attaché au descripteur de processus. Chaque processus hybride dispose de sa propre table de pages.

- L'espace d'adressage virtuel du processus n'est plus décrit par une seule liste de régions virtuelles. Chaque processus hybride dispose de sa propre table de partitionnement qui reflète l'organisation en trois parties (privée, locale et globale) de son propre espace d'adressage virtuel. Pour chaque processus hybride, la première entrée dans cette table de partitionnement pointe sur la liste des régions virtuelles mappées dans sa partie privée. La deuxième et troisième entrées dans cette table pointent, respectivement, sur la liste des régions virtuelles de la partie locale et sur la liste des régions virtuelles de la partie globale.

4.5.4 Discussion

Revenons sur la question de la localité des miss instruction et des miss TLB générés par un thread utilisateur s'exécutant en mode noyau (appelés pour faire court : miss noyau). Selon le modèle existant de threads, chaque thread réfère à l'unique table de pages du processus. La partie noyau de cette table de pages dispose du même mapping que le réplica noyau du nœud cc-NUMA où le processus est placé. Par conséquent, les accès mémoire liés aux miss noyau sont distants pour tout thread placé dans un nœud différent de celui de son processus (i.e :

celui du premier thread de son processus). Notre concept de processus hybrides résout cette difficulté. En effet, chaque processus hybride dispose de sa propre table de pages. Ainsi, lors de la création (ou placement) d'un processus hybride sur un core d'un nœud cc-NUMA, le noyau copie les informations de traduction du réplica noyau du nœud cible dans la partie noyau de la table de pages du processus hybride. Ainsi, à tout moment la partie noyau de la table de pages d'un processus hybride réfère à la copie locale de PTEs noyau et aux pages physiques contenant la copie locale du code noyau.

Le concept de processus hybride répond aux deux objectifs que nous avons rappelés au début de cette section, à savoir : (i) renforcer de la localité des accès mémoire liés aux miss instruction et TLBs et (ii) doter le noyau d'une abstraction adéquate aux architectures cc-NUMA lui permettant de contrôler finement les accès aux pages physiques avec une granularité de fil d'exécution. En revenant sur le problème de perte de la localité des accès mémoire d'une tâche suite à une opération d'équilibrage de charge (c.f : section 4.3.4), l'abstraction de processus hybride permet au noyau de migrer les données de la tâche dans son nouveau nœud cc-NUMA. En effet, l'application de la stratégie de migration Auto-Next-Touch (c.f : section 4.4.3) sur les deux parties globale et privée de l'espace d'adressage virtuel du processus hybride permet de migrer progressivement les pages physiques lors du prochain accès. Ainsi, le noyau est capable de migrer la tâche et ses données tout en réduisant le coût de cette migration puisque : (i) le marquage des pages à migrer ne concerne que cette tâche; et (ii) uniquement les pages accédées par la tâche après son transfert seront migrées.

D'autre part, il est important de noter que le partitionnement et la gestion de l'espace d'adressage virtuel des processus hybrides sont assurés par le noyau sans intervention du programmeur. Il s'agit d'une solution au niveau noyau et par conséquent, totalement compatible avec le standard PThread. Cependant, le fait d'avoir d'une part, la facilité de programmation des threads classiques et d'autre part, la protection assurée par la présence d'un espace d'adressage virtuel par fil d'exécution doté d'une partie privée, font des processus hybrides un cadre plus riche concernant le développement d'exécutifs (run-time), de bibliothèques, voire même, de paradigmes de programmation pour le mode utilisateur. En effet, en introduisant un drapeau supplémentaire dans la fonction *mmap*, le noyau peut autoriser un processus hybride à mapper dynamiquement des régions virtuelles dans la partie privée de son espace d'adressage virtuel. Cela ouvre la porte à la mise en place, entres autres, d'un tas privé par processus hybride. Ainsi, le programmeur voulant tirer profit du nouveau modèle de threads, c'est à dire les processus hybrides, peut optimiser le traitement intermédiaire de données en utilisant des régions mémoire privées.

Les avantages d'un tel usage de la partie privée : (i) garantir le confinement ou l'isolation (p. ex : applications de cryptographie ou manipulation d'information sensible) et (ii) minimiser le coût des défauts de pages puisque leur résolution ne rentre en aucune concurrence avec celles des autres parties quelque soit le nombre de processus hybrides. D'autre part, le noyau peut utiliser la partie privée du processus hybride, pour projeter dans une page en lecture seule pour le code utilisateur, un certain nombre d'informations telles que : l'identité du core sur lequel il s'exécute, l'identité du nœud cc-NUMA dans lequel il est placé, le nombre de cycles restant avant l'expiration de son quantum en cours, la charge actuelle du core, etc. Ces informations, accessibles sans appel système et maintenues à jour par le noyau, peuvent être très utiles pour les exécutifs utilisateur afin d'optimiser leurs fonctionnement (p. ex : l'allocation dynamique de mémoire dans la libC, la gestion des threads dans le run-time OpenMP, etc). En revanche, notre concept de processus hybrides présente deux inconvénients par rapport à celui des threads classiques.

Le premier est le coût des défauts de pages. En effet, dans le modèle existant de threads, le coût de défaut de page concernant une page de code ou de données est payé par un seul et unique thread, celui qui y accède pour la première fois. Les accès suivants à cette page ne génèrent pas de défauts de pages quelque soit le thread à l'origine de ces accès, car la table de pages est partagée entre l'ensemble de threads du même processus. Dans le cas des processus hybrides chaque processus hybride dispose de sa propre table de pages et par conséquent, chaque processus hybride subit un défaut de page lors de son premier accès à une page de code ou de données. Cet inconvénient se traduit par une augmentation du temps d'exécution de la phase d'initialisation d'un calcul parallèle.

Le deuxième inconvénient potentiel concerne les performances des TLBs ou du cache de données de premier niveau (L1) d'un core quand il est indexé en adresses virtuelles. Les performances de ces caches sont négativement impactées, dans le cas d'une concurrence entre processus hybrides sur le même core. En effet, chaque processus hybride dispose de son propre espace d'adressage virtuel ce qui implique l'éviction de toute entrée, non marqué globale, de la TLB ainsi que l'éviction de toutes les lignes de caches associées au ASID (Adress Space ID) du processus hybride dans le cas d'un cache L1. Dans le modèle existant de threads, la concurrence entre les threads appartenant au même processus sur un core ne présente pas cet inconvénient. Nous pensons qu'un noyau pour many-core doit tirer profit du parallélisme réel de l'architecture et par conséquent, il doit minimiser la concurrence de tâches sur un core, en plaçant les processus hybrides de la même application sur des cors différents.

Enfin, le fait d'interdire les communications entre processus hybrides en passant par la pile peut être considéré comme un inconvénient. Précisons cependant que le standard PThread n'oblige en aucun cas une implémentation à garantir un tel partage ou une telle forme de communication.

4.5.5 Conclusion

Les noyaux monolithiques existants tel que Linux, BSD et Solaris ne traitent pas la question de la localité des accès mémoire liés aux miss instruction et aux miss TLBs dans les architectures many-core cc-NUMA. D'autre part nous pensons que l'abstraction d'un fil d'exécution telle qu'elle est définie dans ces noyaux, n'est pas suffisante dans le contexte d'architectures many-cores cc-NUMA. Nous avons proposé et implémenté dans notre système d'exploitation ALMOS, deux concepts qui sont le réplica noyau et le processus hybride. Grâce à ces deux concepts, le noyau est capable de : (i) répliquer son code et sa table de pages dans chaque nœud cc-NUMA; (ii) renforcer la localité des accès mémoire liés aux miss TLBs lors de l'exécution d'une application parallèle; (iii) répliquer à la demande le code de l'application dans chaque nœud et par conséquence, renforcer la localité des accès mémoire liés aux miss instruction; et (iv) contrôler avec la granularité d'un fil d'exécution, les accès mémoire aux pages physiques.

Enfin, notre nouveau concept de processus hybride est compatible avec le standard PThread ce qui permet d'exécuter des applications existantes sans modification. Notre organisation de l'espace d'adressage d'un processus hybride permet aux programmeurs d'applications et d'environnements d'exécution parallèle, d'exploiter la facilité de programmation et de communication des threads, et de bénéficier des mécanismes de protection et d'isolation des processus.

4.6 Conclusions

Nous avons présenté dans ce chapitre les principes généraux du prototype du système d'exploitation ALMOS permettant de renforcer la localité des accès mémoire des tâches dans le contexte des architectures many-cores cc-NUMA. Les caractéristiques essentielles de l'approche proposée sont les suivantes : (i) une organisation distribuée du noyaux monolithique; (ii) une mise en œuvre répartie des services systèmes tout en prenant en compte le caractère NUMA de l'architecture, (iii) une prise en compte de la localité des accès mémoire au niveau noyau d'une manière transparente aux programmeurs d'application parallèles; et (iv) une redéfinition de la notion de fil d'exécution et de sa représentation dans les noyaux monolithiques.

En effet, la gestion des ressources (cores et mémoires physiques) dans le noyau d'ALMOS est décentralisée reflétant le caractère distribué[12] d'une architecture cc-NUMA. Les décisions dynamiques relatives à la gestion de ces ressources sont prises d'une manière décentralisée et multi-critères prenant en compte à la fois la localité et la disponibilité des ressources. La notion de réplicas noyau permet au noyau de répliquer son code et ses tables de pages dans chaque nœud de l'architecture. La notion de processus hybride permet au noyau de répliquer, à la demande, le code d'une application et de contrôler les accès aux pages physiques par une granularité de fil d'exécution. Bien que ALMOS dispose d'un noyau distribué en mémoire partagée cohérente, il offre un environnement d'exécution compatible POSIX. En particulier, les applications parallèles existantes et qui reposent sur la notion de thread et les primitives de synchronisation en mémoire partagée, peuvent être exécutées sans aucune modification préalable. ALMOS constitue donc un prototype d'un système d'exploitation où la prise en charge des aspects NUMA de l'architecture matérielle est réalisée d'une manière transparente aux applications utilisateur.

L'implémentation de nos solutions et la réalisation d'un système d'exploitation capable d'exécuter des applications POSIX standard ont nécessité la conception et le développement d'autres organes d'un noyau tels que : le sous-système gestionnaire de fichiers avec ses mécanismes d'abstraction et de caches (méta-données et données), l'abstraction et la gestion des périphériques, la couche d'abstraction du matériel (HAL), la procédure d'initialisation et de démarrage, etc. D'autres développements et d'intégrations ont été nécessaires afin d'assurer au noyau une interface avec les applications utilisateur tels que : le développement d'une bibliothèque implémentant une bonne partie du standard PThread et l'intégration d'une bibliothèque C. Cette infrastructure a permis de réaliser les expérimentations présentées dans le chapitre 5.

12 Une architecture cc-NUMA peut être vue comme un système distribué fortement couplé.

Chapitre 5

Évaluations expérimentales

Dans ce chapitre, nous essayons d'évaluer expérimentalement l'efficacité de différents mécanismes permettant le renforcement de la localité des accès mémoire des tâches utilisateur effectué par le noyau d'ALMOS. Nous analysons en particulier le passage à l'échelle des applications hautement multi-threads sur une architecture cc-NUMA many-core intégrée sur puce. Nous commençons par présenter dans la section 5.1, les questions auxquelles les expérimentations doivent fournir des réponses. Ensuite, dans la section 5.2 nous présentons le dispositif d'expérimentation retenu. Enfin, nous présentons les expérimentations menées ainsi que l'analyse de leurs résultats dans la section 5.3 avant de terminer ce chapitre en présentant nos conclusions dans la section 5.4.

5.1 Questions investiguées

- Quel est l'impact du schéma d'ordonnancement réparti client-serveur sur le temps de réveil des threads ?

- Quel est l'impact de la réalisation répartie de l'appel système *fork* sur le temps d'exécution de ce service système ?

- Quel est le coût (en nombre de cycles) de la mise à jour de l'infrastructure de prise de décision décentralisée et multi-critères, la DQDT ?

- Quel est l'impact de la stratégie d'affinité mémoire automatique Auto-Next-Touch (en terme de scalabilité et de trafic distant) sur l'exécution des application multi-threads sur un processeur many-core cc-NUMA ?

- Quel est l'impact du modèle de processus hybrides (en terme de scalabilité et de trafic distant) sur l'exécution des applications fortement multi-threads sur un processeur many-core cc-NUMA ?

- Quelle est la scalabilité des applications parallèles utilisées dans les différentes évaluations lorsqu'elles sont exécutées sur une machine multi-cores existante exécutant Linux ?

5.2 Dispositif d'expérimentation

Notre dispositif d'expérimentation est constitué de deux éléments. Le premier est l'architecture matérielle. Nous utilisons le prototype virtuel du processeur many-core cc-NUMA TSAR pour exécuter le système d'exploitation ALMOS. Le deuxième élément concerne le choix des applications parallèles de référence. Le prototype virtuel de TSAR est présenté dans la section 5.2.1, tandis que les applications parallèles de référence que nous avons retenues sont présentées dans la section 5.2.2.

5.2.1 TSAR (Tera-Scale ARchitecture)

5.2.1.1 Présentation

TSAR est l'architecture d'un processeur many-core cc-NUMA homogène pouvant intégrer jusqu'à 4096 cores [8]. Cette architecture est un résultat d'un projet de recherche européen MEDEA+ [123,124] (2008) dont les principaux partenaires industriels sont BULL, Philips et THALES, et dont les partenaires académiques sont le LIP6 et le CEA-Leti. La figure 5.1 illustre un aperçu global de l'architecture TSAR. Il s'agit d'un ensemble de clusters interconnectés par le NoC (Network-On-Chip) DSPIN (Distributed, Scalable, Predictable, Integrated Network) [125]. Chaque core dispose de ses propres caches L1 indexés en adresses physiques (données et instructions séparés) et d'une MMU (Memory Management Unit). La cohérence des caches de premier niveau de tous les cores ainsi que des TLBs est assurée par un protocole matériel nommé DHCCP (Distributed Hybrid Cache Coherence Protocol). Une description complète de cette architecture est disponible sur le site du projet TSAR [9].

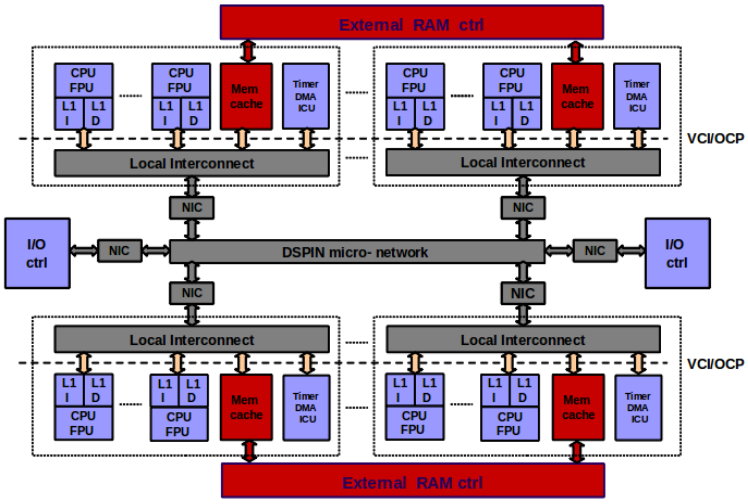


Figure 5.1 : un aperçu global de l'architecture TSAR. Source : revue finale du projet TSAR, Alain Greiner, mai 2011.

5.2.1.2 Motivations

L'architecture TSAR est prototypée virtuellement en SystemC [126] en utilisant la bibliothèque de composants SoCLib [127]. La modélisation de cette architecture est précise au cycle et au bit près. Nous l'avons retenu comme support matériel dans le but de réaliser ALMOS et de mener nos expérimentations, pour les trois raisons suivantes :

1. Il n'existe pas encore industriellement de processeur many-core cc-NUMA ayant plusieurs centaines de cores. Une telle architecture est pourtant nécessaire pour analyser le passage à l'échelle des solutions proposées.

2. L'architecture many-core TSAR dispose de plusieurs points en commun avec les architectures industrielles proposées par Tilera comme Tile64 [13] et Tile-Gx [18] tels que : (i) l'usage d'un NoC sous forme d'un mesh ayant la même stratégie de routage et d'acheminement des paquets; (ii) le choix d'une stratégie Write-Through pour tous les caches L1; et (iii) la taille relativement petite des caches du premier et deuxième niveau.

3. Il existe un simulateur pour TSAR ayant une précision au niveau CABA (Cycle Accurate Bit Accurate) permettant l'exécution d'une pile logicielle complète (système d'exploitation et applications) dès le signal de démarrage (reset) du processeur TSAR.

L'utilisation d'un simulateur précis au niveau CABA a principalement deux avantages. Le premier, c'est la précision des résultats obtenus puisque le simulateur prend en compte de manière fidèle les différentes latences liées au fonctionnent des composants matériels et aux contentions sur ceux-ci. Le deuxième, c'est d'avoir une possibilité d'observation non intrusive ni pour le matériel émulé ni pour le logicielle exécuté. L'inconvénient majeur, c'est la vitesse de simulation qui est très lente allant de 250 Kcycles simulés par seconde pour un TSAR à 4 cores, à 400 cycles par seconde pour un TSAR à 1024 cores. Cela rend la tâche du développement logiciel plus difficile et limite le choix des applications et la taille de leurs jeu de données lors d'une étude expérimentale.

5.2.1.3 Paramètres et configurations

La figure 5.2 illustre l'architecture clusterisée 2D du processeur many-core TSAR. La configuration de référence de TSAR que nous avons utilisée dans toutes les expérimentations présentées dans ce chapitre est récapitulée dans le tableau 5.1.

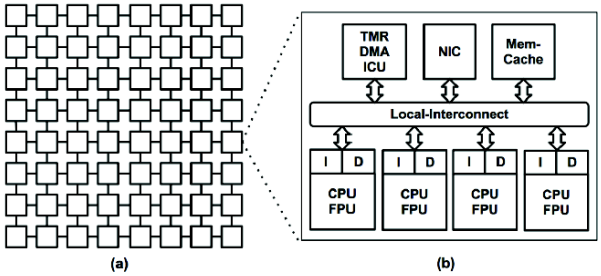


Figure 5.2 : (a) l'architecture clusterisée de TSAR avec un NoC en 2D; (b) un cluster de TSAR est constitué de 4 cores, un interconnect local, un NIC (Network-Interface) vers le NoC, un Memory-Cache (L2), un multi-timers, un contrôleur DMA et une ICU (Interrupt Controler Unit).

Le nombre de clusters utilisés est variable selon les expérimentations. Ainsi, pour évaluer l'exécution d'une application sur *N* cores, un mesh de *X* * *Y* clusters est utilisé où les valeurs de *N*, *X* et *Y* sont les suivantes : 4 cores (1 * 1 cluster), 16 cores (2 * 2 clusters), 64 cores (4 * 4 clusters), 256 cores (8 * 8 clusters), 512 cores (8 * 16 clusters) et 1024 cores (16 * 16 clusters). Le prototype virtuel du processeur many-core TSAR intègre dans l'un de ses clusters (dit cluster E/S) les quatre ressources matérielles suivantes :

- Un contrôleur disque à capacité DMA (Direct Memory Access) offrant une abstraction LBA (Logical Block Array) avec une taille de bloc (secteur) configurée à 4096 octets. Ce contrôleur accède à une image de disque formaté en utilisant le système de fichiers Ext2.

- Une ROM (Read Only Memory) contenant le noyau d'ALMOS, un bootloader et un fichier binaire décrivant l'ensemble des ressources matérielles présentes lors du démarrage.

- Un contrôleur de terminal d'entrée/sortie texte.

- Un contrôleur de sortie vidéo (frame-buffer).

Ces trois ressources matérielles permettent l'exécution des différentes phases de démarrage. En effet, suite au signal "reset", le bootloader est exécuté afin de charger le noyau dans la RAM avant de lui donner la main. Le noyau d'ALMOS identifie l'ensemble de ressources matérielles présentes (grâce au fichier binaire descriptif de ressources) et charge dynamiquement les pilotes adéquats. Enfin, après l'initialisation des différents sous-systèmes du noyau, en particulier le VFS (Virtual File System) avec ses couches sous-jacentes (Ext2, DevFS, SysFS), le noyau exécute le premier processus utilisateur "/bin/init", qui à son tour lance le shell d'ALMOS "/bin/sh". Les programmes d'évaluation peuvent alors être lancés en mode interactif sur le terminal ou en utilisant un fichier de configuration "/etc/shrc".

Parameter	Value
Cache Line Size	64 Bytes
Instruction L1	64 sets, 4 ways, 16 KiB
Data L1	64 sets, 4 ways, 16 KiB
Instruction TLB	64 entries
Data TLB	64 entries
MMU	2 levels, Hardware Table-Walk
Page Size	2 sizes: 2 MiB, 4 KiB
MemCache (L2)	Unified, 256 sets, 16 ways, 256 KiB
Core	32 Bits, Type: MIPS ISS, ISA: MIPS32

Tableau 5.1 : la configuration de référence de l'architecture TSAR utilisée dans toutes les expérimentations.

5.2.2 Les applications parallèles utilisées

Nous avons retenu les huit applications parallèles suivantes : FFT, LU, Radix, Ocean, FMM, Histogram, EPfilter et Tachyon. Ces applications relèvent du domaine du HPC (High Performance Computing) et du domaine du traitement d'images. Elles sont toutes écrites en C et expriment le parallélisme de threads en utilisant le standard PThreads.

5.2.2.1 Présentation

FFT implémente une version à une dimension de la transformation de Fourrier en six étapes, décrite par Bailey et al. [128]. LU implémente une version dite "contiguë" d'une factorisation de matrice dense en produit de deux matrices triangulaires (basse et haute), décrite par Woo et al. [129]. Radix implémente un tris de nombres entiers basé sur l'utilisation d'un arbre radix selon la méthode décrite par Blelloch et al. [130]. Ocean implémente une version dite "contiguë" d'une simulation de mouvements à large échelle d'océan, décrite par Woo et al. [131]. FMM implémente une version parallèle d'une méthode adaptative, rapide et multi-pôles afin de simuler l'interaction d'un système de corps (le problème dit "N-Body"); décrite par Singh et al. [132]. Ces cinq premières applications font partie de la suite parallèle d'évaluations SPLASH-2 [133].

Histogram fait partie du projet Phoenix de Standford [112], le programme génère un histogramme en fréquence de valeurs des pixels selon les couleurs de base (rouge, vert et bleu). EPfilter est une application industrielle de filtrage d'images médicales et elle nous a été fournie par Philips, partenaire du projet TSAR. Elle consiste à appliquer un large filtre de convolution de 201x35 pixels sur une image médicale en entrée de 2 octets par pixel. Les synchronisations inter-threads sont globales par barrière. Enfin, Tachyon [134] est un système de ray-tracing fortement parallèle et qui fait partie de la suite d'évaluation SPEC MPI2007. Il fonctionne également en multi-threads selon le standard PThread.

5.2.2.2 Motivations

Nous avons retenu les applications ci-dessus pour les raisons suivantes :

- Nous avons besoin d'applications fortement parallèles en mémoire partagée, existantes et dont le degré du parallélisme peut être varié au lancement. Cela nous permet d'évacuer la question de la qualité des applications utilisées en cas d'anomalie dans les résultats.

- Nous avons besoin d'applications fortement communicantes afin d'évaluer l'efficacité du concept de processus hybride pour le passage à l'échelle de telles applications.

- Nous avons besoin de différents schémas de communications en mémoire partagée entre les threads d'une même application. Deux schémas de communication sont souhaités. Le premier est un schéma étendu où chaque thread communique avec tous les autres threads par des lectures et/ou par des écritures. Le deuxième est un schéma plus limités où chaque thread communique avec un nombre restreint d'autres threads. Cela nous permet d'évaluer l'impact de nos solutions concernant le renforcement de la localité des accès mémoire dans différentes conditions de stresse et de contention sur le système mémoire (L1, TLBs, L2) et sur le NoC du processeur many-core TSAR.

Finalement nous avons besoin d'applications dont le temps d'exécution reste relativement petit (au plus quelques milliards de cycles) puisque nous sommes limités par les temps de simulation du prototype virtuel de TSAR.

5.2.2.3 Paramètres et configurations

Le tableau 5.2 récapitule les paramètres d'entrée de l'ensemble des applications utilisées dans nos expérimentations. Les abréviations indiquées par ce tableau seront utilisées dans le restant de ce chapitre. Pour chaque application, tout paramètre non indiqué dans ce tableau vaut sa valeur par défaut prédéfinie dans l'application elle-même. Les applications en questions ont été compilées en utilisant une chaîne de compilation croisée GCC (version 4.4.3) et un mode d'optimisation -O3.

Abbreviation	Comment
FFT-M18	FFT with 262144 complex double
LU-N1024	LU contiguous with an input matrix of 1024 x 1024
Radix-N24	Radix sort applied on 16777216 (2^24) kyes
Ocean-N514	Ocean contiguous with grid size of 514 x 514
FMM-i16384	FMM with 16384 particules, 2 clusters, Plummer Model
Histo-8	Histogram with 8.7 Mpix input image
EP1024	EPfilter with 1 Mpix input image
EP2048	EPfilter with 4 Mpix input image
Tach-2048	Tachyon with scene (2 balls) resolution of 2048x2048

Tableau 5.2 : les paramètres d'entrée de référence de l'ensemble des applications utilisées dans nos expérimentations.

5.3 Expérimentations

Dans cette section, nous allons investiguer chaque question à part en présentant d'abord notre méthodologie d'expérimentation; ensuite, les résultats obtenus et l'analyse de ces résultats.

5.3.1 Ordonnanceur réparti

Le but de cette expérimentation est d'évaluer l'impact du schéma d'ordonnancement réparti client-serveur sur le temps de réveil des threads.

5.3.1.1 Méthodologie

Nous allons comparer les performances obtenues par l'utilisation de deux versions du noyau d'ALMOS. La première (nommée "baseline") implémente une approche d'ordonnancement similaire à celle du noyau Linux puisqu'elle dispose d'un ordonnanceur par core protégé par verrou à attente active (spinlock). La deuxième version du noyau (nommée "sched-server") implémente un serveur d'ordonnancement par core (présenté dans la section 4.2.2). La politique d'ordonnancement mise en œuvre par les ordonnanceurs de ces deux versions du noyau est exactement la même. Elle consiste à gérer trois niveaux de priorités (migration, noyau, utilisateur) en appliquant la stratégie round-robin à chaque niveau. Enfin, les deux versions du noyau implémentent le même modèle de threads existant dans les noyaux monolithiques d'aujourd'hui avec une gestion de l'espace d'adressage virtuel des processus similaire à celle de Linux.

L'application d'évaluation retenue pour cette expérimentation est un microbench écrit en C et reposant sur l'API des threads POSIX (Pthreads). Il consiste à créer un certain nombre de threads N (un paramètre de la ligne de commandes) qui se synchronisent par barrière. L'implémentation de la barrière de synchronisation (*pthread_barrier_t*) dans la bibliothèque système (Libpthread) fait appel au noyau afin de réaliser ce service. Ainsi, tout thread qui rejoint la barrière est mis en attente passive par le noyau (via la primitive d'ordonnancement *sched_sleep*) à l'exception du dernier arrivant. À l'arrivée de ce dernier, le noyau réveille l'ensemble des N-1 threads (en appelant N-1 fois la primitive d'ordonnancement *sched_wakeup*).

Deux mesures sont réalisée : (i) Le nombre total de cycles de la phase parallèle de ce microbench qui consiste à effectuer 32 synchronisations globales par barrière; et (ii) Le coût moyen en nombre de cycles de l'opération de réveil des N-1 threads réalisée par le noyau. La différence entre ces deux mesures réside dans le fait que la première est une mesure plus globale incluant le coût de fonctionnement des différentes primitives d'ordonnancement ainsi que l'impact de ces primitives sur le working-set des threads; alors que la deuxième mesure cible uniquement le coût de la seule primitive d'ordonnancement *sched_wakeup* en fonction de la distance inter-nœuds et du nombre de threads à réveiller.

5.3.1.2 Résultats et analyses

La figure 5.3 montre l'évaluation du coût de réveil des threads par le noyau. Les résultats montrent que l'utilisation d'un serveur d'ordonnancement par core permet de diviser le coût de cette opération par un facteur de 5.6 pour une configuration ayant 256 threads sur 256 cores. Ces résultats indiquent clairement un meilleur passage à l'échelle en nombre de cores de l'opération de réveil réalisé selon l'approche de l'ordonnancement client-serveur contre celle basée sur un accès partagé protégé par verrou.

La figure 5.4 montre l'évaluation du gain en temps d'exécution de la phase parallèle du même microbench lors de l'utilisation de la version du noyau "sched-server" par rapport à la version

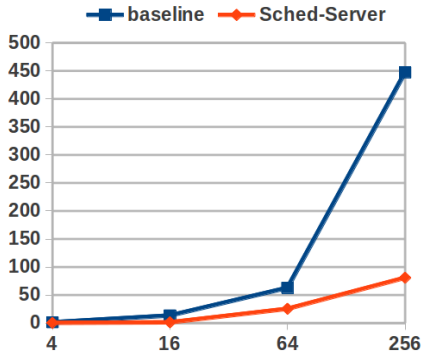


Figure 5.3 : le coût de réveil en milliers de cycles de N-1 threads où N est le nombre de threads (axe X) avec un thread par core.

“baseline”. Ce gain est effectif dès 4 cores et peut aller jusqu’à 67 % pour 256 cores.

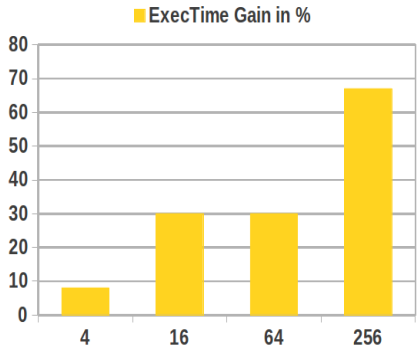


Figure 5.4 : Le gain en temps d’exécution de la phase parallèle. L’axe X indique le nombre de cores avec un thread par core, tandis que l’axe Y indique le pourcentage du gain.

Nous nous intéressons à présent au cas où le nombre de threads est plus grand que le nombre de cores : le même microbench est exécuté, mais en lançant 4 threads par core. La figure 5.5 montre l’évaluation du coût de l’opération réveil de 4N-1 threads en fonction du nombre de cores N et de la version du noyau utilisé. Les résultats de cette figure confirment le meilleur passage à l’échelle en nombre de cores et en nombre de threads de l’opération de réveil dans la version “sched-server” comparé à celle de la version “baseline”. Le coût de cette opération est devisé par un facteur 4.2 quand il s’agit de réveiller 1023 threads sur 256 cores (64 clusters du many-core TSAR).

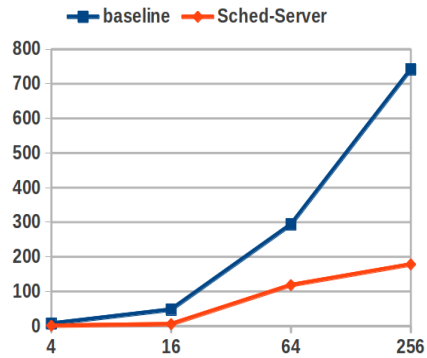


Figure 5.5 : le coût de réveil en milliers de cycles de 4N-1 threads où N est le nombre de cores indiqué par l’axe X.

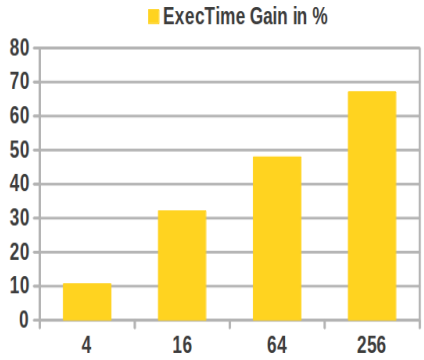


Figure 5.6 : Le gain en temps d'exécution de la phase parallèle. L'axe X indique le nombre de cores avec un thread par core, tandis que l'axe Y indique le pourcentage du gain.

La figure 5.6 montre l'évaluation du gain en temps d'exécution de la phase parallèle du microbench lors de l'utilisation de la version du noyau "sched-server" comparé avec à la version "baseline", dans le cas où l'ordonnanceur de chaque core gère 4 threads. Les résultats de cette figure confirme le gain en temps d'exécution dès 4 cores (1 cluster du many-core TSAR).

5.3.1.3 Conclusion

Les différentes évaluations montrent que la mise en place d'un serveur d'ordonnancement par core fonctionnant présente un avantage effectif comparé à un ordonnanceur par core à accès partagé protégé par verrou. Ainsi, le schéma d'ordonnancement réparti client-serveur permet un meilleur passage à l'échelle pour la primitive *sched_wakeup* ce qui doit améliorer en conséquence les performances des différents moyens de synchronisation par attente passive du noyau, tels que les barrières, les sémaphores, les variables de conditions, etc. Enfin, le gain en temps d'exécution observé, même pour un faible nombre de cores (environ 30% pour 16 cores), indique la pertinence de la mise en place d'un serveur d'ordonnancement par core même pour une configuration relativement petite.

5.3.2 Appels systèmes répartis

Le but de cette expérimentation est d'évaluer l'impact de la réalisation répartie, en utilisant le mécanisme des événements, de l'appel système *fork* sur le temps d'exécution de cet appel.

5.3.2.1 Méthodologie

Nous allons comparer dans cette expérimentation les performances obtenues par l'utilisation de deux versions du noyau d'ALMOS qui diffèrent uniquement par leurs mises en œuvre de la même procédure de création d'un processus fils en réponse à l'appel système *fork*. Dans la première version, après que la DQDT a été consultée et qu'un core cible a été déterminé, la création du processus fils par clonage se fait à distance. Selon cette méthode, le code réalisant le service *fork* s'exécute entièrement en local sur le core exécutant l'appel système (le core exécutant le processus père). Toutes les allocations de mémoire et les initialisations nécessaires à la création et au clonage du nouveau processus sont effectuées à distance en utilisant l'allocateur mémoire du nœud cible. Cette première version est référencée par la suite par "Remote".

Dans la deuxième version, après que la DQDT a été consultée et qu'un core cible a été déterminé, le noyau invoque à distance (en utilisant le mécanisme des événements) la fonction réalisant la création du processus fils par clonage sur le core cible. Toutes les allocations de mémoire et initialisations nécessaires à la création et au clonage du nouveau

processus sont effectuées en local en utilisant l'allocateur mémoire du nœud cible. Ce schéma réparti de la création d'une tâche a été présenté dans la section 4.2.3. Cette deuxième version est référencée par la suite par "Distributed". Ces deux versions du noyau implémentent le même modèle de threads existant dans les noyaux monolithiques d'aujourd'hui avec une gestion de l'espace d'adressage virtuel des processus similaire à celle de Linux.

L'application d'évaluation retenue pour cette expérimentation est un microbench écrit en C et respectant la norme POSIX. Il consiste à dupliquer un processus père pour créer 512 processus fils en utilisant le service *fork*. La configuration de TSAR utilisée est de 128 clusters (512 cores). Le temps (en nombre de cycles) de la réalisation de l'appel système *fork* est mesuré pour chaque duplication. Le temps moyen d'une duplication est ensuite calculé en fonction de la distance de Manhattan séparant le cluster de coordonnées (0,0) où le processus père s'exécute du cores cible où le processus fils a été placé. Étant donné la taille du mesh (8x16) de TSAR utilisé dans cette expérimentation, cette distance varie donc entre 0 et 22.

5.3.2.2 Résultats et analyses

La figure 5.7 montre l'évaluation du coût en nombre de cycles d'une duplication en fonction de la distance par rapport au cluster du processus père invoquant l'appel système *fork*. Les résultats montrent que la réalisation répartie de l'appel système *fork* permet d'améliorer le temps d'exécution de ce service particulièrement lorsque la distance entre le nœud source et le nœud cible augmente. D'autre part, le temps d'exécution de l'appel système *fork* dans la version "Remote" augmente de 103 % quand le processus fils est placé dans le cluster le plus loin (distance 22) comparé au cas d'un processus placé localement. Cette même augmentation n'est que de 14 % dans le cas de la version "Distributed". Cela montre que la réalisation répartie de cet appel système passe mieux à l'échelle.

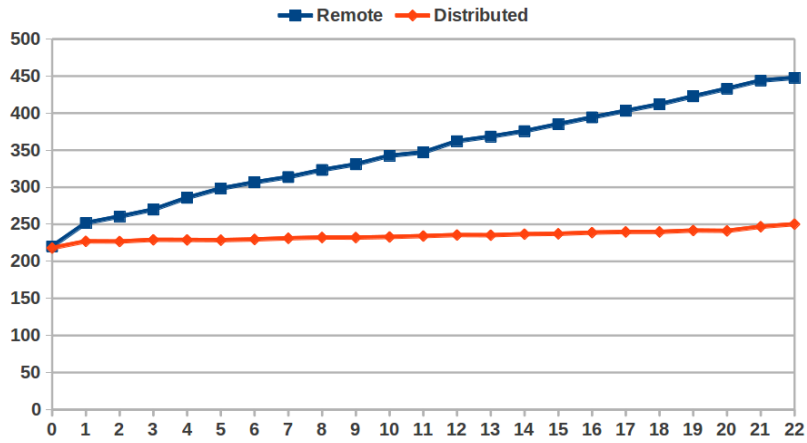


Figure 5.7 : l'évaluation du coût d'une duplication de processus. L'axe X indique la distance de Manhattan. L'axe Y indique le nombre de cycles x1000.

5.3.2.3 Conclusion

Les résultats de cette expérimentation montrent un avantage en terme de temps d'exécution et de passage à l'échelle d'une réalisation répartie de l'appel système *fork*. Cette avantage est lié au fait que le code réalisant la création et le clonage du nouveau processus s'exécute près des données sur lesquelles il opère. Par conséquent, la réalisation répartie peut être utilisée dans le but de renforcer la localité des accès mémoire lors de la réalisation d'un service système opérant sur des données accédées à distance.

Enfin, cette expérimentation n'a pas été conduite dans un contexte plus dynamique où différentes applications s'exécutent sur différents cores. En effet, le microbench s'exécute

seul et les processus fils dupliqués se synchronisent aussitôt par attente passive sur une barrière globale. Les éventuelles contentions sur les nœuds cibles des processus fils liées aux accès mémoire issus de l'exécution des tâches (locales ou distantes) de différentes applications ne sont pas prises en compte. Dans ce cas de figure, les accès distants aux données manipulées par l'appel système *fork* dans la version "Remotre" devraient être impactés d'avantage par ces contentions qu'une réalisation répartie dans la version "Distributed".

5.3.3 Mise à jour de la DQDT

Le but de cette expérimentation est d'évaluer le coût de la mise à jour de l'infrastructure de prise de décision décentralisée et multi-critères, la DQDT (Distributed Quaternary Decision Tree). Présentée dans la section 4.3.

5.3.3.1 Méthodologie

Comme nous l'avons présenté dans la section 4.3, la mise à jour de l'arbre DQDT est réalisée par deux opérations différentes en fonction des indicateurs de ressources à mettre à jour. La première concerne les indicateurs M (disponibilité en mémoire physique) et U (pourcentage de l'utilisation des cores). Elle est réalisée périodiquement par autant de tâches qu'il y de nœuds cc-NUMA, à savoir, des clusters logiques de niveau 0 (feuilles de l'arbre DQDT). La deuxième opération de mise à jour concerne les indicateurs T (nombre de threads actifs utilisateur). Elle est réalisée immédiatement suite à une décision de placement, migration ou terminaison d'un thread. Elle consiste à effectuer une incrémentation (ou décrémentation) atomique des indicateurs T faisant partie du chemin entre une feuille la racine de l'arbre.

Étant donné la nature de ces deux types d'opérations de mise à jour (périodique dans le 1er cas, et immédiate dans le second) nous allons évaluer chacune de ces opérations séparément. La premier évaluation consiste à mesurer le temps (en nombre de cycles) de l'opération de mise à jour périodique exécutée par chacune des threads de mise à jour. Cette mesure est répétée 10 fois et le temps moyen de chaque tâche est alors retenu. La somme de l'ensemble de ces résultats permet de déterminer le surcoût global lié à cette opération de mise à jour collective par rapport à un intervalle du temps (en nombre de cycle) exprimant la capacité de calcul de l'ensemble des cores.

La deuxième évaluation consiste à mesurer le coût (en nombre de cycles) de l'opération de mise à jour immédiate en fonction de deux paramètres : le nombre de cores susceptible d'exécuter simultanément une opération de ce type et la distance les séparent du nœud racine de l'arbre DQDT. Pour cela, un microbench écrit en C et faisant appel à l'API PThreads est exécuté par le noyau d'ALMOS sur une configuration de TSAR à 512 cores (128 clusters). La phase séquentielle de ce microbench consiste à lancer un certain nombre de threads. La phase parallèle consiste à ce que chaque thread crée à son tour un autre thread et il se synchronise sur sa terminaison. Afin d'augmenter la probabilité d'avoir plusieurs mises à jour immédiate simultanées, deux techniques sont utilisées. La première consiste à synchroniser l'ensemble de threads crées lors de la phase parallèle par une barrière globale au début de la phase parallèle. La deuxième consiste à contrôler explicitement le placement de l'ensemble de threads selon le scénario pire cas suivant : 256 threads placés dans les 128 clusters avec 2 threads par cluster, mais sur deux cores différents.

L'arbre DQDT examiné dans cette expérimentation dispose d'un hauteur de 4. Lors d'une opération de mise à jour immédiate, 2 additions atomiques par niveau sont effectuées. L'indicateur U contient d'un seul champ, le pourcentage d'utilisation. L'indicateur M contient deux champs : (i) le nombre estimé de pages physiques disponibles, et (ii) un tableau de 10

cases où chaque case indique le nombre estimé de blocs contigus de 2^i pages physiques disponibles (i étant l'indice de la case). Le choix de cette implémentation de l'indicateur M est lié au fonctionnement de l'allocateur de pages physiques du Memory-Manager (algorithme Buddy [115,116]). L'opération de mise à jour périodique faite par une tâche doit donc accumuler les estimations de 4 x (1+1+10) = 48 compteurs pour un cluster logique d'un niveau donné et propager la valeur de (1+1+10) = 12 compteurs au cluster logique père du niveau suivant.

5.3.3.2 Résultats et Analyses

La figure 5.8 montre l'évaluation du coût en nombre de cycles de l'opération de mise à jour périodique en utilisant une configuration de TSAR à 1024 cores (256 clusters).

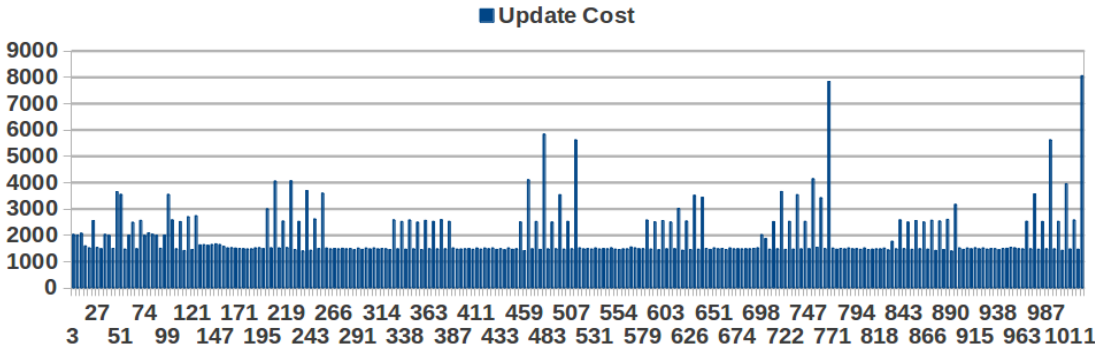


Figure 5.8 : Le coût par core de l'opération de mise à jour périodique. L'axe X indique les cores exécutant une tâche de mise à jour. L'axe Y indique le nombre de cycles.

Uniquement 256 cores sont impliqué dans l'opération de mise à jour périodique puisqu'il y a un thread de mise à jour par cluster. La valeur minimum est de 1400 cycles, tandis que la valeur maximum est de 8100 cycles. Le nombre de cycles total de cette opération de mise à jour collective est de 500700 cycles. Pour une période de mise à jour de 1 million de cycles et pour 1024 cores; cela représente un coût relatif de : (500700 / (1024 x10^6)) = 0.05 %. Dans le cas où le processeur many-core TSAR est cadencé à 500 MHz, cet intervalle de 1 million de cycles représente 2 ms. Une opération de mise à jour chaque 2 ms est donc tout à fait envisageable.

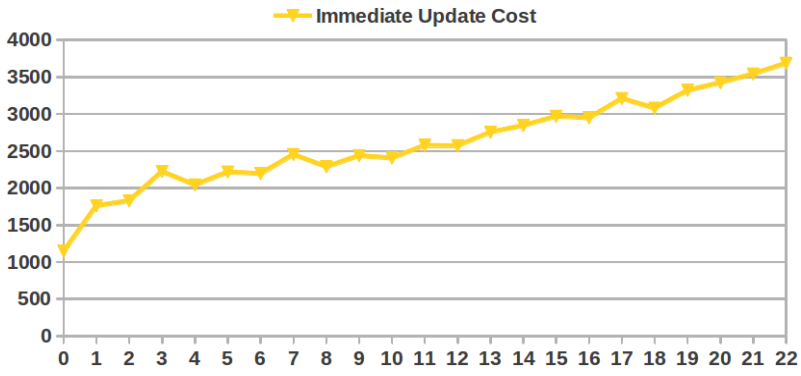


Figure 5.9 : Le coût moyen d'une opération de mise à jour immédiate. L'axe X indique la distance de Manhattan. L'axe Y indique le nombre de cycles.

La figure 5.9 montre l'évaluation du coût moyen de l'opération de mise à jour immédiate exécutée par 256 threads en fonction de la distance séparant le cluster où l'opération de mise à jour est exécuté, du cluster où le nœud racine de l'arbre est placé. Le mesh de TSAR utilisé est de 8x16 clusters. Le nœud racine est placé dans le cluster de coordonnées (7,15). La

distance de Manhattan par rapport au nœud racine varie donc entre 0 et 22 inclus. Les résultats de cette figure montrent que l'éloignement par rapport au nœud racine de l'arbre augmente le coût de l'opération de mise à jour immédiate, ce qui est un résultat attendu. D'autre part, les résultats montrent également que l'impact de l'augmentation en nombre de threads sur le coût de l'opération est limité selon le scénario pire cas pour un processeur TSAR à 512 cores où 256 threads créent en parallèle 256 autres threads en les plaçant sur des cores libres.

5.3.3.3 Conclusion

Les différents résultats obtenus dans cette expérimentation permettent de déterminer le coût des deux opérations de mise à jour de l'arbre DQDT (périodique et immédiate) sur deux grosses configurations du many-core TSAR (respectivement 1024 et 512 cores). Le coût permanent payé par le quart des cores, afin de maintenir les estimations U et M, constitue un coût acceptable (5/10000 de la puissance de calcul totale). D'autre part, le coût payé à chaque création ou destruction d'un thread pour mettre à jour immédiatement les indicateurs T concernés, est faiblement impacté par le nombre de threads susceptibles d'exécuter simultanément ce type d'opération de mise à jour, et ne pose donc pas de problème de passage à l'échelle.

5.3.4 Stratégie Auto-Next-Touch

Le but de cette expérimentation est d'évaluer l'impact en terme de temps d'exécution, de trafic distant et de scalabilité de la stratégie d'affinité mémoire automatique Auto-Next-Touch.

5.3.4.1 Méthodologie

Nous allons comparer les performances de la stratégie Auto-Next-Touch décrite dans la section 4.4.3 avec deux autres stratégies : First-Touch et Interleave. La première est utilisée dans les noyaux monolithiques existants tels que Linux. La deuxième consiste à distribuer l'allocation des pages physiques, demandées par l'application lors des défauts de pages, sur l'ensemble des nœuds cc-NUMA. Les applications utilisées dans cette expérimentation sont : FFT-M18, LU-N1024, FMM-i16384 et Ocean-N514. Ces applications ont été exécutées sans modification particulière et sans l'ajout d'appels aux services non POSIX. Lors de cette expérimentation, nous nous intéressons uniquement à la phase de l'exécution parallèle de ces applications et le nombre de threads de traitement est égal au nombre de cores (association 1-à-1).

Mise en place des trois stratégies

Pour cette expérimentation, nous utilisons une version d'ALMOS dont la notion de threads et la gestion de l'espace virtuel de processus sont tout à fait similaires à celles de Linux. Cette version du noyau implémente les trois stratégies. La stratégie Auto-Next-Touch y est appliquée par défaut, mais ce choix par défaut peut être désactivé et l'une des deux autres stratégies (First-Touch ou Interleave) peut être activée dynamiquement (avec une granularité de thread) à deux moments précis : lors de la création d'un thread et lors de la bifurcation d'un processus (appel système fork). Sans indication explicite lors de ces deux appels systèmes, la stratégie du thread créateur est héritée par le nouveau thread et elle est conservée lors de la transformation d'un processus (appel système exec). Pour pouvoir appliquer une stratégie différente à chaque lancement du même programme, la commande *exec* du shell d'ALMOS interprète la valeur d'une variable d'environnement : *ALMIX_MEMORY_AFFINITY*. Selon la valeur de cette variable, le shell informe le noyau du choix de la stratégie à appliquer au nouveau processus.

Observation du trafic distant

Pour pouvoir évaluer l'impact des trois stratégies d'affinité mémoire en terme de trafic inter-nœuds, nous avons réalisé un module d'instrumentation matériel au niveau de chaque cluster et l'avons intégré au prototype virtuel de TSAR comme il est illustré par la figure 5.10.

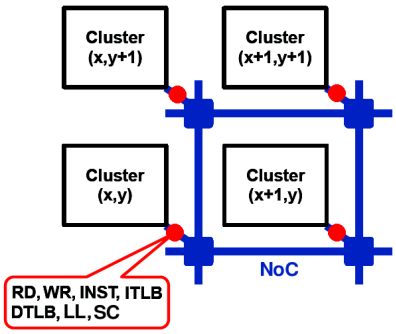


Figure 5.10 : Le module d'instrumentation (points rouges) est attaché à chaque cluster, il permet de compter les commandes distantes reçues par son cluster.

Ce module permet de compter, d'une manière non intrusive, le nombre de commandes distantes reçues par le cluster auquel il est attaché. Cela nous permet de construire un histogramme de toutes les commandes distantes et quantifier ainsi leur distribution sur l'ensemble des clusters. Les commandes distantes reçues par un cluster sont de 7 types : (i) lecture d'une ligne de cache suite à un miss du cache L1 data (RD); (ii) lecture d'une ligne de cache suite à un miss du cache L1 instruction (INST); (iii) lecture d'une ligne de cache suite à un miss TLB data (DTLB); (iv) lecture d'une ligne de cache suite à un miss TLB instruction (ITLB); (v) écriture d'un à quatre mots de 32 bits (WR); (vi) lecture d'un mot 32 bits appelée "linked-load" (LL); et (vii) écriture d'un mot 32 bits appelée "store conditional" (SC). Les commandes LL et SC implémentent des opérations de type "lecture-puis-écriture" atomique et sont générées par un core suite à l'exécution d'une opération de synchronisation inter-tâches.

D'autre part, afin de quantifier le trafic distant, nous considérons une estimation de la quantité de données échangées inter-clusters durant l'exécution de la phase parallèle de chaque application. Cette estimation est calculée comme suit (*N* étant le nombre de clusters) :

$$Q = 64 * \bigsqcup_{0}^{N-1}(RD,\ INST,\ DTLB,\ ITLB) + 4 * \bigsqcup_{0}^{N-1}(WR,\ LL,\ SC)$$

Le premier coefficient 64, correspond à la taille d'une ligne de cache en octets; tandis que le deuxième 4, suppose que les commandes WR, LL et SC transfèrent 4 octets (1 mot de 32 bits).

Enfin, notre évaluation expérimentale de l'impact de la stratégie Auto-Next-Touch se déroule selon les étapes suivantes : (i) évaluer le speedup des quatre applications sélectionnées pour cette expérimentation en fonction des trois stratégies d'affinité mémoire (First-Touch, Interleave et Auto-Next-Touch) mises en œuvre par le noyau; (ii) caractériser le trafic inter-cluster en analysant conjointement le speedup obtenu et la distribution correspondante des commandes distantes; et (iii) évaluer la quantité du trafic distant inter-clusters obtenus en fonction de la stratégie d'affinité mémoire.

5.3.4.2 Résultats et analyses

La figure 5.11 présente le speedup obtenu par une exécution sur un TSAR à 64 cores par rapport à une exécution sur une architecture à 1 seul core. Elle montre l'impact de chacune des trois stratégies First-Touch, Interleave et Auto-Next-Touch sur la scalabilité des quatre applications parallèles.

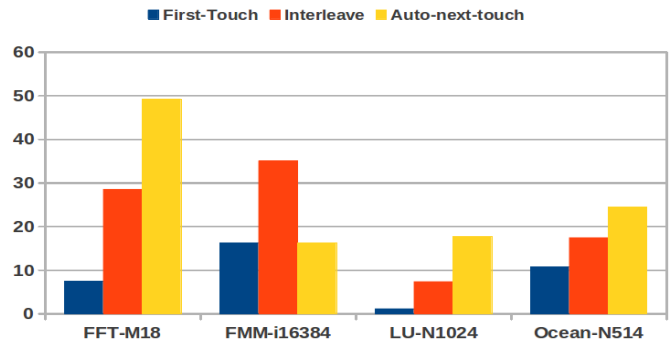


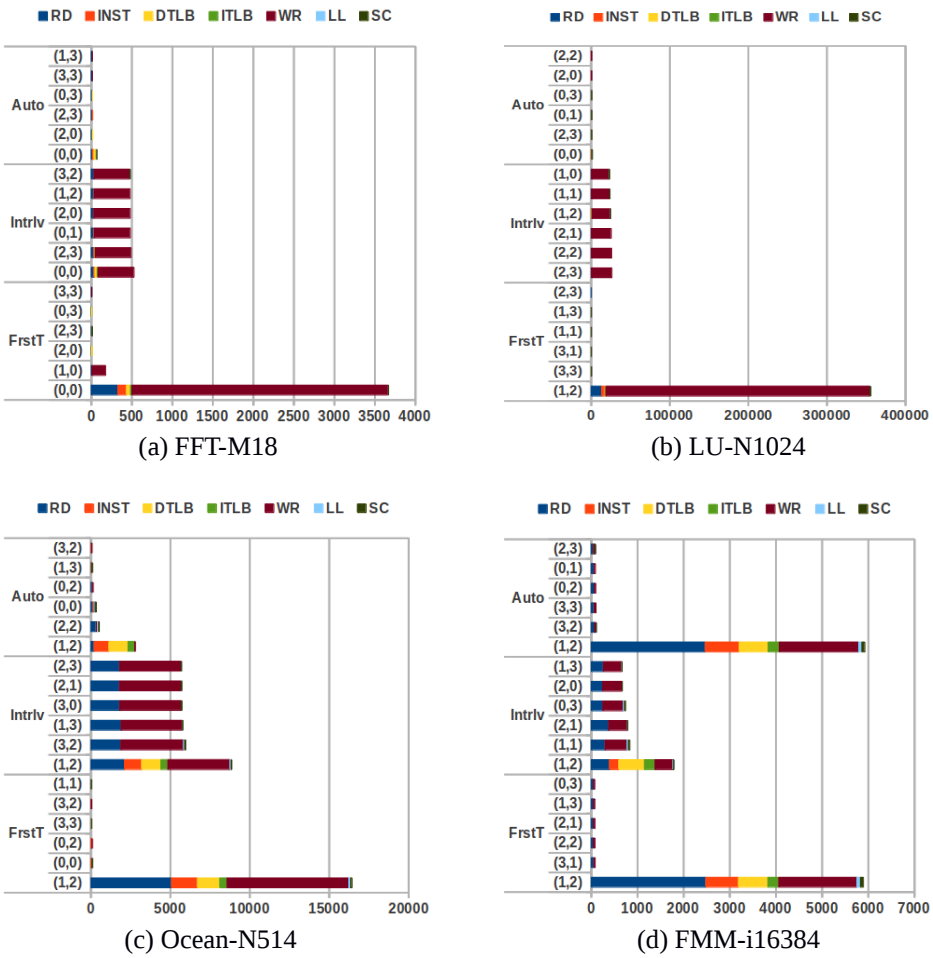
Figure 5.11 : Le speedup, sur l'axe Y, obtenu de l'exécution séparée des quatre applications sur TSAR à 64 cores pour les trois stratégies First-Touch, Interleave et Auto-Next-Touch.

Les résultats de la figure 5.11 montrent que pour les trois applications FFT-M18, LU-N1024 et Ocean-N514; Auto-Next-Touch permet d'avoir un speedup de 2.26 à 15.5 fois supérieur par rapport à First-Touch et de 1.4 à 2.4 fois supérieur par rapport à Interleave. En revanche, pour l'application FMM-i16384, Auto-Next-Touch ne présente pas d'amélioration de speedup par rapport à celui de First-Touch, alors que Interleave donne un speedup 2 fois supérieur par rapport aux deux autres stratégies.

Pour mieux comprendre les résultats de la figure 5.11, nous nous intéressons à la distribution du trafic distant inter-clusters pour chaque application examinée. La figure 5.12 montre la distribution des différents types de commandes distantes reçues par chaque cluster lors de l'exécution séparée des quatre applications sur 64 cores (16 clusters) en fonction de la stratégie d'affinité mémoire. Les clusters sont triés par ordre décroissant par rapport au nombre de commandes reçues. Seuls les six premiers clusters sont représentés, tandis que les clusters non représentés ont des valeurs très proches de celle du dernier cluster représenté.

En analysant conjointement les résultats des deux figures 5.11 et 5.12, nous pouvons en déduire les points suivants :

1. La mauvaise performance en terme de speedup de la stratégie First-Touch est effectivement due à un mauvais choix de placement de données effectué par le noyau lors de l'exécution de la phase d'initialisation d'une application parallèle. En effet, les pages physiques constituant la quasi-majorité des données à traiter sont allouées sur le cluster du thread initiateur. Par conséquent, la stratégie First-Touch cause un goulot d'étranglement lié au grand nombre de commandes distantes visant le même cluster.



(a) FFT-M18 (b) LU-N1024

(c) Ocean-N514 (d) FMM-i16384

Figure 5.12 : Une comparaison entre les distributions de commandes distantes sur les clusters en fonction des trois stratégies d'affinité mémoire. L'axe X donne le nombre de commandes distantes x1000. L'axe Y indique les coordonnées des clusters.

2. En distribuant les allocations de pages physiques sur l'ensemble de clusters, la stratégie Interleave permet d'améliorer le speedup par rapport à First-Touch, car cette distribution élimine le goulot d'étranglement dans le traitement des commandes distantes relatives aux données. Toutefois, le trafic distant inter-clusters reste important.

3. La stratégie Auto-Next-Touch permet de renforcer la localité des accès mémoire aux données grâce à la migration des pages physiques accédées au début de la phase parallèle de chaque application. Ce renforcement de localité se traduit par l'augmentation de speedup constatée plus haut par rapport aux deux autres stratégies. Le cas de l'application FMM-i16384 montre que la migration de pages n'a pas été effectuée et que dans ce cas là, la distribution de commandes distantes et le speedup sont similaires à ceux obtenus en appliquant la stratégie First-Touch.

Analysons plus précisément l'application FMM-i16386. Le seul scénario possible qui peut expliquer pourquoi la migration des pages physiques n'a pas eu lieu est le suivant : un ou plusieurs threads placés dans le cluster contenant les données initiales, accèdent les premiers à l'ensemble des pages physiques constituant ces données. Ainsi, le premier accès à ces données fait par un thread placé dans un autre cluster, ne génère plus de défaut de page. Par conséquent, il n'y a pas de migration de pages vers les autres clusters. Pour vérifier cette hypothèse, nous avons donc modifié le code de l'application FMM. Cette modification se

résume en deux actions : (i) déplacer une fonction (création de boites de particules) appelée par un seul thread dans la phase parallèle vers la phase séquentielle; et (ii) rajouter dans la phase parallèle deux boucles *for* d'une ligne pour que chaque thread touche un sous-ensemble d'objets en mémoire (particules et boites) manipulés dans cette phase. Notre modification du code n'est donc pas d'ordre structurel ou algorithmique et elle n'ajoute aucune fonction au code d'origine. Il s'agit simplement d'ajouter un patern d'accès aux données permettant au noyau d'effectuer la migration de pages comme prévue dans la stratégie Auto-Next-Touch.

La figure 5.13 montre le speedup obtenu lors de l'exécution successive de l'application FMM2-i16384 modifiée confirmant que notre modification permet au noyau de doubler le speedup de l'application en appliquant la stratégie Auto-Next-Touch.

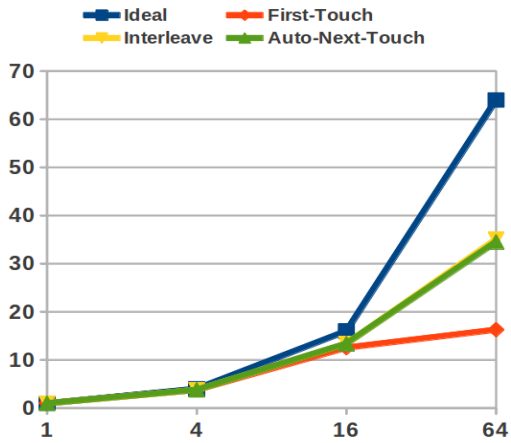


Figure 5.13 : Le speedup de l'application FMM2-i16384 modifiée en fonction de la stratégie d'affinité mémoire mise en œuvre. L'axe X indique le nombre de cores.

La figure 5.14 montre une comparaison entre les distributions de commandes distantes lors de l'exécution parallèle de l'application FMM2-i16384 modifiée en fonction de la stratégie d'affinité mémoire appliquée par le noyau.

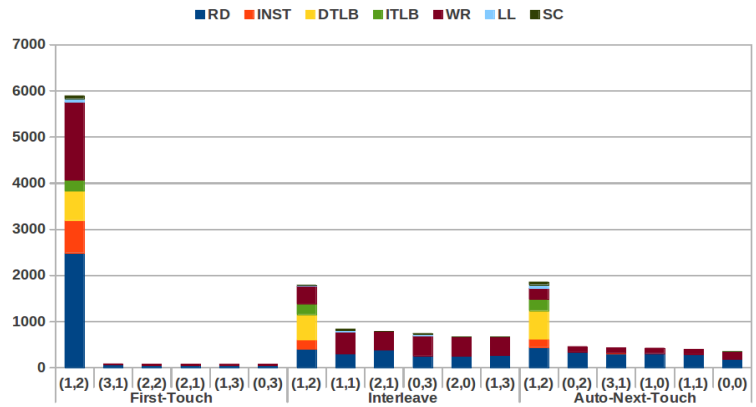


Figure 5.14 : La distribution des commandes distantes sur les clusters en fonction de la stratégies d'affinité mémoire lors de l'exécution de FMM2-i16384. L'axe Y indique le nombre de commandes distantes x1000. L'axe X indique les coordonnées des 6 clusters les plus sollicités

Les résultats de la figure 5.14 montrent que le patern d'accès aux données que nous avons introduit déclenche effectivement la migration de pages physiques par le noyau selon la stratégie Auto-Next-Touch. En corrélant le speedup montré dans la figure 5.13 et la distribution du trafic inter-clusters montrée dans la figure 5.14, nous pouvons déduire que les valeurs proches de speedup obtenues en appliquant les deux stratégies Auto-Next-Touch et

Interleave sont essentiellement dues à la sérialisation restante du traitement d'une partie des commandes distantes par le contrôleur de cache L2 du cluster de coordonnée (1,2) où le premier thread de l'application a été placé.

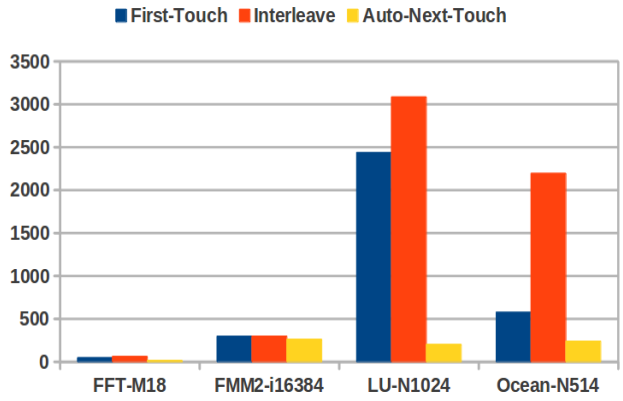


Figure 5.15 : L'estimation de la quantité du trafic distant inter-clusters généré par l'exécution parallèle des quatre applications sur TSAR à 64 cores en fonction de la stratégie d'affinité mémoire. L'axe Y indique la valeur de cette estimation en MiB.

Concernant le trafic distant inter-clusters généré lors de l'exécution parallèle d'une application en fonction de la stratégie d'affinité mémoire, la figure 5.15 montre une estimation de ce trafic en mégaoctets lors de l'exécution des mêmes applications sur TSAR à 64 cores. Les résultats montrent que la stratégie Auto-Next-Touch permet une réduction de trafic distant allant jusqu'à 15 fois par rapport à Interleave et jusqu'à 12 fois par rapport à First-Touch.

5.3.4.3 Conclusion

Les différents résultats de cette expérimentation montrent que les performances obtenues en appliquant la stratégie Auto-Next-Touch dépassent celles obtenues en appliquant les deux autres stratégies First-Touch et Interleave. Auto-Next-Touch donne une meilleure scalabilité, un meilleur temps d'exécution parallèle et réduit le trafic distant inter-nœuds. En réduisant le trafic inter-nœuds, la stratégie Auto-Next-Touch devrait permettre de réduire également la consommation électrique liée aux transferts de données.

Cependant, pour que la migration de pages physiques selon la stratégie Auto-Next-Touch puisse fonctionner d'une manière effective, il est indispensable que chaque thread accède à sa propre partie des données à traiter avant de commencer le traitement parallèle. En particulier, quand le traitement parallèle nécessite un accès à l'ensemble des données à traiter. Toutefois, l'accès par chaque thread à sa propre partie des données à traiter peut se faire avec la granularité de page et l'introduction d'un tel patern d'accès ne nécessite pas d'appel système ni de modifications d'ordre algorithmique.

5.3.5 Processus hybrides

Le but de cette expérimentation est d'évaluer l'impact en terme de scalabilité et de trafic distant du modèle de processus hybride sur les applications fortement multi-threads s'exécutant sur un processeur many-core cc-NUMA.

5.3.5.1 Méthodologie

Afin d'évaluer l'impact du modèle de processus hybride, nous allons étudier les performances obtenues en utilisant deux versions du noyau monolithique distribué d'ALMOS. La première, implémente la notion actuelle de threads en tant que fils d'exécution

à l'intérieur du même processus et partageant son espace d'adressage virtuel. Cette première version dispose donc du même modèle de threads que dans la plupart des noyaux d'aujourd'hui tels que Linux, Solaris, BSD et Windows NT/XP/2000. La deuxième version du noyau implémente le modèle de processus hybride que nous avons présenté dans la section 4.5.3. Les deux versions du noyau ont la même ABI (Application Binary Interface), c'est-à-dire, les mêmes appels systèmes. Par conséquent, l'utilisation de l'une ou l'autre version du noyau est transparente pour les applications utilisateur (les applications ont été compilées en utilisant les bibliothèques d'ALMOS).

Dans cette expérimentation, nous utilisons les trois applications suivantes : FFT-M18, Radix-N24 et EP1024/EP2048. Ces trois applications ont été exécutées sans aucune modification. Nous nous intéressons uniquement à la phase d'exécution parallèle de ces applications où le nombre de threads de traitement est égal au nombre de cores. Nous utilisons également le prototype virtuel du processeur many-core TSAR incluant le module d'instrumentation non intrusive du trafic distant inter-clusters que nous avons présenté dans la section précédente. Ceci nous permet d'étudier l'impact du modèle de thread employé sur la distribution des commandes distantes et d'estimer la quantité du trafic distant généré par ces commandes.

Enfin, notre évaluation expérimentale de l'impact du modèle de processus hybride se déroule selon la démarche suivante : (i) évaluer le speedup des trois applications sélectionnées pour cette expérimentation en utilisant le modèle de threads existant; (ii) caractériser le trafic inter-cluster en analysant conjointement le speedup obtenu et la distribution correspondante des commandes distantes; (iii) évaluer le speedup des mêmes applications en utilisant le modèle de processus hybrides; (iv) comparer la distribution correspondante des commandes distantes générées par la mise en ouvre au niveau du noyau de chacun des deux modèles de threads; et (v) évaluer la quantité de trafic distant inter-clusters en fonction du modèle de threads utilisé.

5.3.5.2 Résultats et Analyses

La figure 5.16 montre l'évaluation du speedup pour les trois applications fortement multi-threads, jusqu'à 512 cores et en utilisant la première version du noyau d'ALMOS appelée *"baseline"* implémentant le modèle classique de threads.

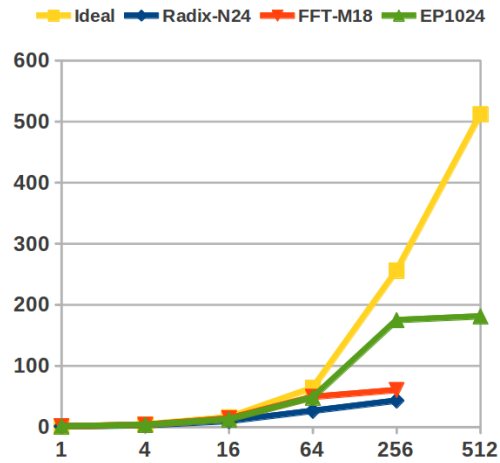
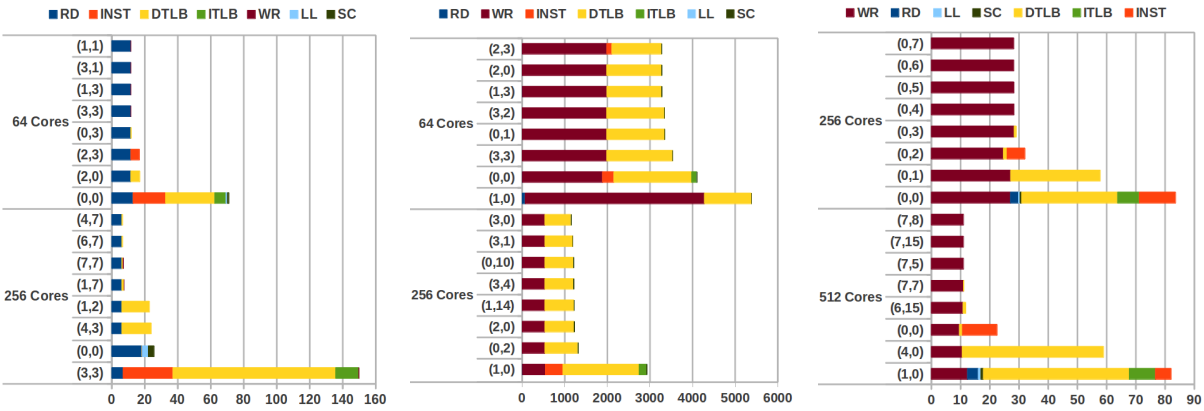


Figure 5.16 : Le speedup des trois applications obtenu en utilisant le modèle existant de threads. L'axe X indique le nombre de cores.

Les résultats de la figure 5.16 montrent une limitation de speedup pour l'application EP1024 au delà de 256 cores et une limitation de speedup au delà de 64 cores pour les deux applications FFT-M18 et Radix-N24. Pour mieux comprendre ces résultats, nous nous intéressons à la distribution des commandes distantes sur l'ensemble des clusters de TSAR.

Le but est de vérifier si la limitation de speedup détectée dans la figure 5.16 provient d'un mauvais placement de données.



(a) FFT-M18 (b) Radix-N24 (c) EP1024

Figure 5.17 : Une comparaison entre les distributions de commandes distantes correspondante aux deux derniers points de scalabilité pour chacune des trois applications. L'axe X indique le nombre de commandes distantes x1000. L'axe Y indique les coordonnées des huit clusters retenus pour chaque point de scalabilité.

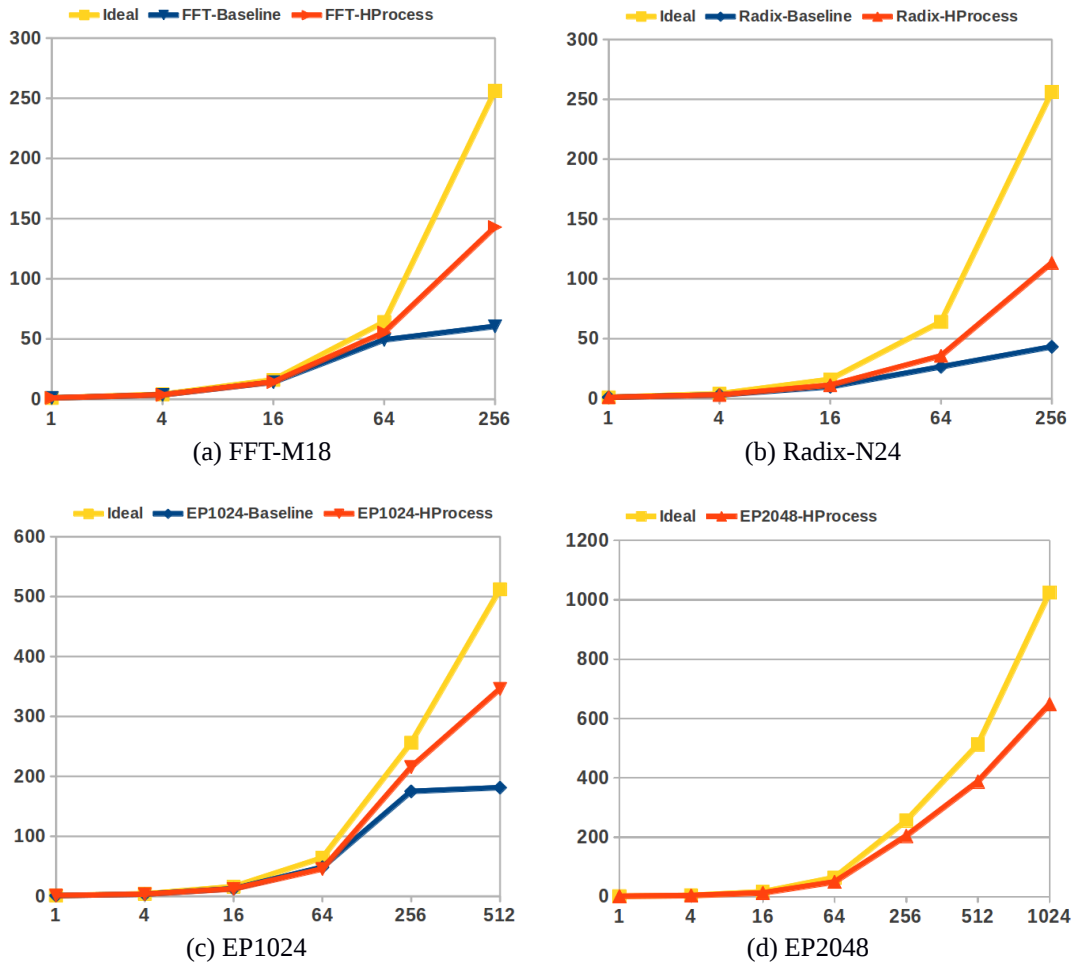
La figure 5.17 montre l'évolution de la distribution des commandes distantes pour 64, 256 et 512 cores. Les clusters ont été triés par ordre décroissant en nombre de commandes reçues. Seuls les huit premiers clusters ont été représentés tandis que les clusters non présentés de la figure 5.17 ont tous des valeurs similaires à celles du dernier cluster représenté.

Dans le suite de cette section, nous classons les commandes distantes reçues par chaque cluster en deux catégories : *"utiles"* et *"surcoût"*. Les commandes de la première catégorie "utiles" sont celles qui servent à accéder aux données de l'application et à effectuer les communications et synchronisation inter-threads. Il s'agit des commandes suivantes : RD, WR, LL et SC. Les commandes de la deuxième catégorie "surcoût" sont celles issues par les caches du premier niveau suite à des miss instruction et TLBs. Il s'agit des commandes suivantes : INST, DTLB et ITLB. La logique de cette classification est expliquée ci-dessous.

La figure 5.17 montre que la distribution des commandes *"utiles"* est en majeure partie uniforme sur les différents clusters. La part de ces commandes reçues par chaque cluster diminue globalement quand le nombre de cores augmente. En revanche, la distribution des commandes de la deuxième catégorie *"surcoût"* montre une concentration sur un ou deux clusters, même si les commandes *"surcoût"* sont globalement mieux réparties dans le cas de l'application Radix-N24. Par conséquent, cette limitation du speedup n'est pas liée à un mauvais placement de pages physiques de données, mais au goulot d'étranglement dans le traitement des commandes d'accès aux instructions et aux informations de traduction de l'espace d'adressage virtuel des processus.

Les commandes de *"surcoût"* sont intrinsèquement liées au modèle du threads existant selon lequel les threads partagent le même espace d'adressage virtuel de leur processus. Ceci conduit systématiquement à référer à l'unique copie du code et à l'unique table de pages du processus par l'ensemble de threads. Plus il y a de threads, plus le trafic distant généré par les accès mémoire liés aux miss instruction et TLBs augmente; ce qui à son tour accroît la contention sur le cache L2 de chaque cluster, c'est-à-dire, sur le LLC (Last Level Cache) de chaque nœud cc-NUMA de l'architecture.

La figure 5.18 montre une comparaison de speedup obtenu en exécutant les mêmes applications, et sans aucune modification, sur les deux versions du noyau d'ALMOS : "*Baseline*" implémentant le modèle existant de threads et "*HProcess*" implémentant le modèle de processus hybrides. Les résultats de la figure 5.18 montrent que le modèle de processus hybrides permet d'éliminer la limitation de speedup pour les trois applications FFT-M18, Radix-N24 et EP1024. Ainsi, grâce au modèle du processus hybride, les deux premières applications passent à l'échelle jusqu'à 256 cores, tandis que l'application EP1024 passe à l'échelle jusqu'à 512 cores, voire même, jusqu'à 1024 cores pour EP2048 (la même application EPfilter mais avec une image à traiter 4 fois plus grande que celle d'EP1024).

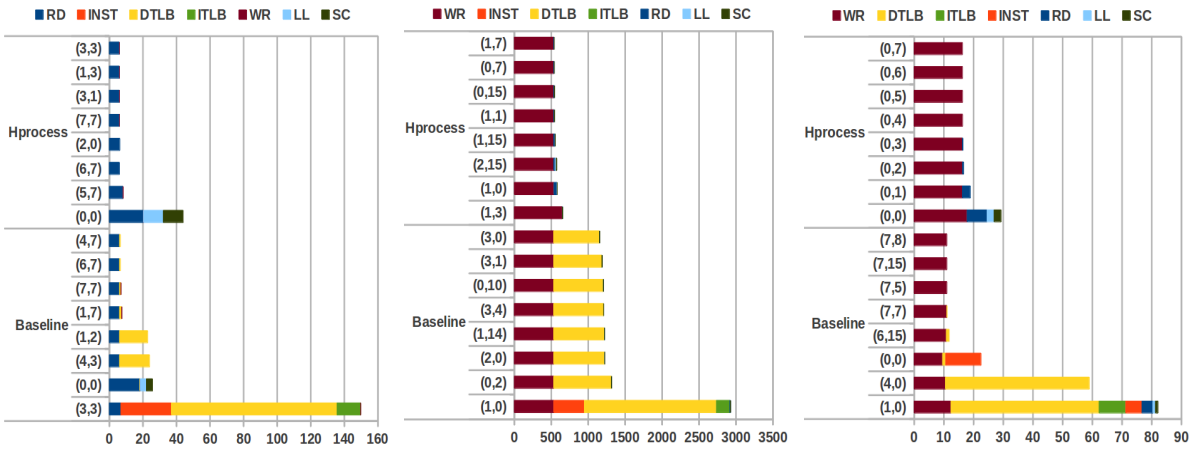


(a) FFT-M18 (b) Radix-N24

(c) EP1024 (d) EP2048

Figure 5.18 : Une comparaison de speedup en fonction de la version utilisée du noyau d'ALMOS : baseline (le modèle existant de threads) et HProcess (le modèle de processus hybrides). L'axe X indique le nombre de cores. L'axe Y indique le speedup obtenu.

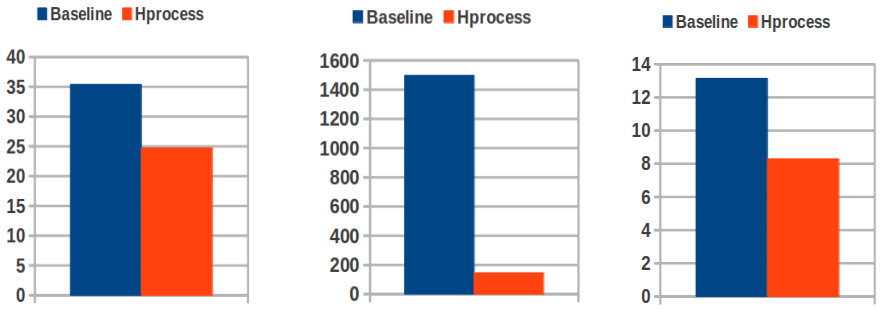
La figure 5.19 montre une comparaison entre les distributions de commandes distantes en fonction de la version de noyau utilisée ("*Baseline*" vs "*HProcess*") : 256 cores pour FFT-M18 et Radix-N24; et 512 cores pour EP1024. Les résultats de cette figure montrent que la version "*HProcess*" du noyau d'ALMOS permet un meilleur renforcement de la localité des accès mémoire des threads puisque les commandes distantes de "surcoût" sont devenues locales.



(a) FFT-M18 (b) Radix-N24 (c) EP1024

Figure 5.19 : Une comparaison entre les distributions de commandes distantes en fonction de la version du noyau d'ALMOS. Le nombre de cores correspondant est 256 dans les deux cas (a) et (b); et 512 cores pour (c). L'axe X indique le nombre de commandes distantes x1000. L'axe Y indique les coordonnées des clusters retenus.

La figure 5.20 montre une comparaison en terme d'estimation de trafic distant inter-clusters en fonction du modèle de threads utilisé. Les résultats de cette figure montrent que le modèle de processus hybrides permet une réduction de trafic distant allant de x1.43 à x10.36 par rapport au modèle existant de threads.



(a) FFT-M18 (b) Radix-N24 (c) EP1024

Figure 5.20 : L'estimation de la quantité du trafic distant inter-clusters généré par l'exécution parallèle des trois applications en fonction du modèle de threads proposé par les deux versions du noyau d'ALMOS. Le nombre de cores correspondant est de 256 dans les deux cas (a) et (b); et de 512 cores pour (c). L'axe Y indique la valeur de cette estimation en MiB.

5.3.5.3 Conclusion

Les différents résultats obtenus dans cette expérimentation montrent que d'une part, le problème de la localité des accès mémoire liés aux miss TLBs et aux miss instructions est exacerbé par l'augmentation en nombre de cores exécutant les threads du même processus; et d'autre part, le modèle de processus hybride constitue une solution à ce problème tout en gardant la compatibilité avec le modèle actuel de threads. Le modèle de processus hybride permet le passage à l'échelle des trois applications examinées et la réduction du trafic distant inter-nœuds. En réduisant le trafic inter-nœuds, le modèle de processus hybride devrait permettre de réduire également la consommation énergétique liée à ce trafic.

5.3.6 Comparaison de scalabilité

Le but de cette expérimentation est d'une part, d'évaluer la scalabilité des applications parallèles utilisées dans nos différentes expérimentations sur une cible multi-cores existante

exécutant Linux; et d'autre part, de comparer cette scalabilité à celle obtenue sur la cible TSAR exécutant ALMOS.

5.3.6.1 Méthodologie

Les applications utilisées sont les suivantes : FFT-M18, LU-N1024, Radix-N24, Ocean-N514, FMM2-i16384, EP1024, Tach-2048 et Histo-8. L'évaluation consiste à mesurer le speedup obtenu en exécutant ces mêmes applications sur deux cibles cc-NUMA à 64 cores. La première cible est le prototype virtuel du processeur many-core TSAR exécutant le système d'exploitation ALMOS. La deuxième cible est une machine ayant 4 processeurs Opteron Interlagos (6282 SE) d'AMD (soit 64 cores) exécutant le système d'exploitation Scientifique Linux (la version du noyau est 2.6.32-358.6.2.el6.x86_64). Les applications ont été compilées en utilisant GCC 4.4.6 (x86_64-redhat-linux) pour la cible Linux/AMD et cross-compilées en utilisant GCC 4.4.3 (mipsel-unknown-elf) pour la cible ALMOS/TSAR. Le mode d'optimisation utilisé est -O3 dans les deux cas.

L'architecture matérielle de la deuxième cible est illustrée par la figure 5.21. Elle est constituée de 4 processeurs ayant chacun 16 cores. Chaque processeur contient deux nœuds cc-NUMA de 8 cores et les 4 processeurs sont interconnectés par des liens Hyper-transport-3.1 sous forme d'une cross-bar. Chaque nœud consiste de 4 modules de calcul dont chacun dispose de 2 cores. Les deux cores par module partagent un seul cache L1 d'instructions (64 KiB) et une seule unité de calcul flottant, mais chaque core dispose de son propre cache L1 de données (16 KiB) et de sa propre unité de calcul entier. Chaque module dispose de son propre cache L2 unifié (2 MiB), tandis que chaque nœud dispose de son propre cache L3 (12 MiB). La taille d'une ligne de cache est de 64 octets. De point de vue système d'exploitation, Linux détecte et gère 8 nœuds cc-NUMA de 8 cores chacun.

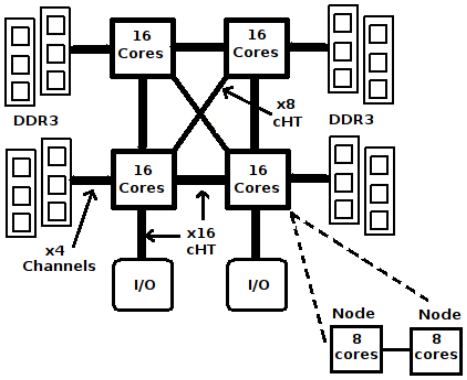


Figure 5.21 : Architecture cc-NUMA à 64 cores constituée de l'interconnexion de 4 processeurs Opteron Interlagos d'AMD.

Afin d'exécuter les applications évaluées en utilisant un thread par core, les threads sont assignés sur les cores d'une manière explicite (le 1er thread sur le core 0, le 2ème thread sur le core 1, ainsi de suite). Sous Linux, la fonction *pthread_setaffinity_np* est utilisée (qui se traduit en l'appel système *sched_setaffinity*); tandis que sous ALMOS, c'est la fonction *pthread_attr_setcpuid_np* qui est utilisée. La différence entre ces deux fonctions réside dans le fait que sous Linux, l'assignation d'un nouveau thread sur un core se fait après la création du thread, nécessitant ainsi une migration. Sous ALMOS, le noyau est informé par cette assignation au moment de la création du thread (le placement explicite est un attribut interprété par l'appel système réalisant la création du thread) ce qui permet au noyau d'ALMOS de créer et lancer le nouveau thread directement sur le core cible. D'autre part, ces deux fonctions désactivent toute migration du thread après son assignation. Mise à part cette différence dans la manière d'assigner un thread sur un core (localisée dans la boucle de

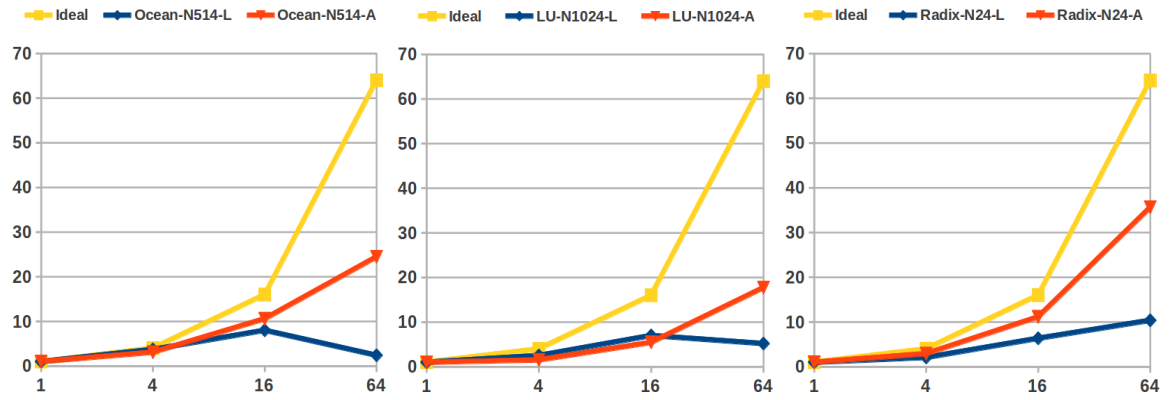
lancement des threads) les 8 applications évaluées ont été exécutées sans changement sur les deux cibles. Uniquement le temps d'exécution (en nombre de cycles) de la phase parallèle de ces applications est utilisé afin de calculer le speedup.

5.3.5.3 Résultats et analyses

La figure 5.22 montre l'évaluation de scalabilité des 8 applications parallèles sur les deux cibles Linux/AMD et ALMOS/TSAR en utilisant 1, 4, 16 et 64 threads. Les résultats montrent que la cible ALMOS/TSAR donne systématiquement une bien meilleure scalabilité pour chacune des applications examinées. À l'exception de deux applications, le speedup mesuré sur la cible Linux/AMD est relativement médiocre. Étant donné que ces mêmes applications donnent un bon speedup sur la cible ALMOS/TSAR, la qualité de ces applications ne peut pas être la cause de ce résultat.

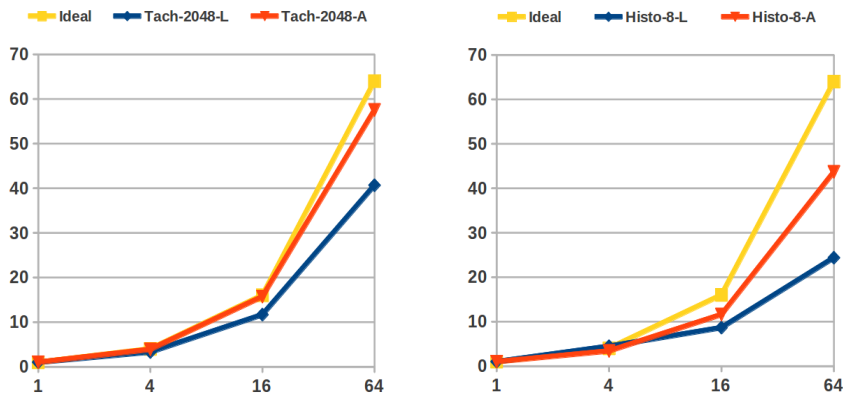
5.3.5.3 Conclusion

Les résultats de cette expérimentation montrent que la cible ALMOS/TSAR fournit systématiquement une bien meilleure scalabilité pour les 8 applications évaluées que la cible Linux/AMD. De point de vue programmeur, l'abstraction de thread en tant qu'une unité de traitement parallèle et la granularité mémoire d'une ligne de cache et d'une page sont suffisantes pour exploiter les performances offertes par la cible ALMOS/TSAR dans le cas de ces applications. Concernant la cible Linux/AMD, il reste très probablement une marge pour améliorer les performances obtenues mais cela doit passer nécessairement par plusieurs phases d'instrumentation, de modification et de validation de code avec deux inconvénients majeurs : augmenter la complexité pour le programmeur, et rendre le code de ces applications dépendant de la cible (services systèmes propres à Linux, topologie matérielle exposée au programmeur et propre à la machine sous-jacente d'AMD).



(a) FFT-M18 (b) LU-N1024 (c) Radix-N24

(d) Ocean-N514 (e) FMM2-i16384 (f) EP1024



(g) Tach-2048 (h) Histo-8

Figure 5.22 : Évaluation de speedup des 8 applications parallèles exécutées sur les deux cibles Linux/AMD et ALMOS/TSAR en utilisant 1, 4, 16 et 64 threads.

5.4 Conclusions

Les résultats des différentes expérimentations présentées dans ce chapitre montrent que les différents mécanismes implémentés dans le système ALMOS renforcent la localité des accès mémoire des tâches et améliorent les performances du système.

La mise en place d'un schéma d'ordonnancement distribué client-serveur par core permet effectivement d'améliorer les performances des primitives d'ordonnancements et de mieux passer à l'échelle la primitive "réveiller une tâche". Étant donné le rôle central de l'ordonnanceur dans la réalisation des différents services de synchronisation à attente passive, (tels que les barrières, les sémaphores, les variables de conditions, etc) ces résultats doivent améliorer également leurs performances.

En utilisant le mécanisme des événements, répartir le service système *fork*, permet de réduire le temps de réalisation de ce service tout en lui assurant une meilleur scalabilité. L'utilisation du mécanisme des événements pour transférer l'exécution d'un code dans le but de le rapprocher des données sur lesquelles il opère, nous semble une technique prometteuse pour les noyaux monolithiques ciblant les architectures many-cores cc-NUMA.

Concernant le coût de la mise à jour de l'infrastructure de prise de décision décentralisée et multi-critères, la DQDT; les résultats ont montré que le coût de l'opération de mise à jour périodique des indicateurs M et U est limité et acceptable, tandis que le coût de l'opération de mise à jour immédiate des indicateurs T est faiblement impacté par l'augmentation du nombre de threads susceptibles d'effectuer simultanément cette opération.

Quant à la stratégie de migration automatique des pages Auto-Next-Touch, les résultats ont montré que les performances obtenues par cette stratégie dépasse celles obtenues en appliquant les deux autres stratégies généralement utilisées : Interleave et First-Touch. La stratégie Auto-Next-Touch permet effectivement au noyau de renforcer la localité des accès mémoire des threads d'une application utilisateur.

Les résultats obtenus dans ce chapitre ont montré également l'impact négatif du trafic distant, lié aux miss TLBs et instructions, sur la scalabilité des applications multi-threads. Le modèle de processus hybride a permis d'éliminer ce trafic distant en renforçant la localité de ces accès mémoire. Ainsi, le modèle de processus hybride a permis de passer à l'échelle les

applications évaluées d'une manière transparente, sans changement préalable du code de ces applications.

Enfin, l'évaluation de la scalabilité des 8 mêmes applications non modifiées (FFT, LU, Radix, FMM, Ocean, EPfilter, Histogram et Tachyon) sur les deux cibles ALMOS/TSAR et Linux/AMD a montré que les limitations du speedup au-delà de 16 cores observées sur la cible Linux/AMD ne sont pas une fatalité et ne doivent pas être interprétées comme un indicateur de la fin du paradigme de programmation parallèle (multi-threads) en mémoire partagée cohérente garantie par matériel. L'excellente scalabilité observée sur la cible ALMOS/TSAR montre que ces limitations ont des causes intrinsèques liées aux technologies actuelles représentées par la cible Linux/AMD. Nous pensons que ces limitations sont liées à la fois au matériel (mauvaise scalabilité des protocoles de cohérence des caches) et au système d'exploitation (mauvaise prise en compte du caractère NUMA des architectures many-cores). Le couple ALMOS/TSAR prouve qu'il est possible de faire autrement.

Chapitre 6

Conclusions et perspectives

Dans les architectures many-cores à mémoire partagée cohérente garantie par le matériel, la question de la localité du trafic généré par les miss de caches L1 (data, instruction et TLB) est primordiale à la fois pour passer à l'échelle et pour réduire la consommation électrique (énergie consommée par bit transféré). Dans cette thèse, nous avons proposé de mettre la question de la localité de ce trafic au centre de l'évolution des noyaux monolithiques. En particulier, nous avons proposé de : (i) restructurer d'une manière distribuée la gestion et l'allocation des ressources mémoires et cores; (ii) introduire des mécanismes de coordinations et de prise de décision multi-critères scalables; (iii) redéfinir le concept de thread et l'organisation de l'espace d'adressage virtuel des processus; et (iv) gérer la localité des accès mémoire au niveau noyau d'une manière transparente aux applications utilisateur. Ainsi, nous avons conçu, réalisé et évalué expérimentalement ALMOS (Advanced Locality Management Operating System), un système d'exploitation expérimental à base de noyau monolithique distribué permettant de renforcer, d'une manière transparente aux applications utilisateur, la localité des accès mémoire des threads.

6.1 Réponses apportées

Dans le chapitre 2, nous avons défini les cinq questions auxquelles nos travaux de thèse apportent des réponses. Ces questions concernent trois problématiques : (i) le renforcement de la localité des accès mémoire; (ii) l'architecture scalable de l'ordonnanceur; et (iii) la gestion coordonnée des ressources cores et mémoires.

6.1.1 Renforcement de la localité des accès mémoire

Nos principales contributions sont les suivantes :

- Une stratégie d'affinité mémoire automatique, nommée *Auto-Next-Touch*, permettant de remédier aux inconvénients de la stratégie *First-Touch* appliquée par défaut par les noyaux monolithiques existants (c.f : section 4.4).

- Deux concepts : le réplica noyau et le processus hybride (c.f : section 4.5). Le premier permet à un noyau monolithique de répliquer son code et ses informations de traduction d'une manière portable et sans assistance matérielle particulière. Le deuxième est une redéfinition de la notion de thread telle qu'elle existe dans les noyaux monolithiques. Le concept de processus hybride permet de doter un noyau monolithique d'une abstraction nécessaire afin de contrôler finement et d'une manière transparente les accès mémoire avec la granularité de thread. En combinant l'affinité mémoire *Auto-Next-Touch*, le concept de réplica noyau et le concept de processus hybride, un noyau monolithique peut : (i) renforcer la localité des accès mémoire des threads liés aux miss des caches data, instructions et TLBs; et (ii) restaurer la localité des accès mémoire d'un thread après une décision de migration impliquant un changement de nœud cc-NUMA.

Nous avons démontré expérimentalement (c.f : section 5.3.4) que la stratégie *Auto-Next-Touch* permet de renforcer la localité des accès mémoire aux données d'une application parallèle et

d'éliminer la limitation de scalabilité causée par la stratégie *First-Touch* en faisant passer à l'échelle les 4 applications évaluées sur un processeur TSAR à 64 cores. Comparée avec une autre stratégie existante *Interleave* sur le processeur TSAR à 64 cores, la stratégie *Auto-Next-Touch* donne une meilleure scalabilité (x1.4 à x2.4) et une meilleure réduction de trafic distant (jusqu'à 15 fois). Cependant, cette stratégie d'affinité mémoire ne concerne que les accès aux données d'une application parallèle. Par conséquent, elle ne traite pas la localité des accès mémoire liés aux miss des caches instructions et TLBs.

Nous avons démontré expérimentalement (c.f : section 5.3.5) que : (i) le modèle classique de threads des noyaux monolithiques ne permet pas de renforcer la localité des accès mémoire des threads liés aux miss des caches instructions et TLBs; (ii) le trafic distant généré par ces accès mémoire cause une limitation de scalabilité sur le processeur TSAR au-delà de 64 cores pour deux applications et au-delà de 256 pour une troisième; (iii) le modèle de processus hybrides permet de renforcer la localité des accès mémoire des threads liés aux miss des caches data, instruction et TLBs; (iii) le modèle de processus hybrides permet de passer à l'échelle deux applications jusqu'à 256 cores et une troisième jusqu'à 1024 cores du processeur TSAR; et (iv) le modèle de processus hybrides permet de réduire le trafic distant des trois applications évaluées de x1.43 à x10.36 par rapport au modèle existant de threads. Cependant, dans le temps qui nous est imparti, nous n'avons pas pu étudier expérimentalement la restauration de la localité des accès mémoire après une décision de migration impliquant un changement de nœud cc-NUMA. Nous reviendrons sur ce point dans la section 6.2.1.

6.1.2 Architecture scalable de l'ordonnanceur

Sur ce point, notre contribution est la conception d'un schéma d'ordonnancement pour les noyaux monolithiques qui soit scalable et adapté aux processeurs many-cores à mémoire partagée cohérente garantie par matériel. Nous proposons (c.f : section 4.2.2) un schéma d'ordonnancement distribué client-serveur qui permet de passer à l'échelle les primitives d'ordonnancement sur un processeur many-cores cc-NUMA. Contrairement au schéma d'ordonnancement partagé des noyaux monolithiques existants, l'état de l'ordonnanceur d'un core selon notre proposition n'est pas accessible depuis les tâches qui s'exécutent sur les autres cores. Ceci élimine le besoin de protection par verrou et réduit le trafic distant liés à l'exécution de la primitive *"réveiller une tâche"*.

Nous avons démontré expérimentalement (c.f : section 5.3.1) que la mise en place d'un serveur d'ordonnancement par core permet de diviser le coût de l'opération "réveiller une tâche" par un facteur 5.6 et donne clairement une bien meilleur scalabilité sur le processeur TSAR à 256 cores. L'évaluation du coût de fonctionnement des différents primitives d'ordonnancement indique un gain effectif en temps d'exécution du microbench utilisé dès 4 cores et peut aller jusqu'à 67% sur 256 cores du processeur TSAR. Étant donné que les techiques de synchronisation inter-tâches à attente passive (les barrières, les sémaphores, les variables de conditions, etc) reposent entièrement sur les primitives d'ordonnancement, le schéma d'ordonnancement distribué client-serveur doit améliorer leurs performances.

6.1.3 Gestion coordonnée des ressources cores et mémoires

Notre dernière contribution porte sur : (i) la conception d'un mécanisme décentralisé passant à l'échelle permettant d'une part, d'unifier la représentation de la localité pour tous les services et sous-systèmes du noyau et d'autre part, d'effectuer une prise de décision multi-critères en prenant en compte à la fois la localité et la disponibilité de la ressource recherchée; et (ii) le renforcement de la localité des accès mémoire lors de la réalisation des services systèmes tels que la création d'un

processus, ou la création et la migration d'une tâche. Ainsi, nous proposons une organisation distribuée permettant de décentraliser la gestion des ressources cores et mémoires et de renforcer la localité des accès mémoire lors de cette gestion. En particulier, elle comporte :

- Une infrastructure distribuée, nommée *DQDT (Distributed Quaternary Decision Tree)* (c.f : section 4.3) définissant un mécanisme wait-free basé sur un ensemble d'indicateur d'usage de ressources et intégrant une représentation hiérarchique multi-niveaux de la notion de voisinage entre ces ressources. Il permet au noyau une prise de décision dynamique, décentralisée et multi-critères concernant l'allocation mémoire, le placement d'une tâche et l'équilibrage de charge.

- Une méthodologie permettant de répartir la réalisation des services systèmes afin de renforcer la localité de leurs accès mémoire. En particulier, la réalisation répartie de la création d'une tâche (processus/thread) et celle de la migration de tâche ont été présentées dans les sections 4.2.3 et 4.3.4 respectivement.

Nous avons démontré expérimentalement que le coût de la mise à jour périodique des différents indicateurs d'usage de ressources de la *DQDT* est négligeable (c.f : section 5.3.3). En particulier, le coût de la mise à jour périodique de la DQDT pour un processeur TSAR à 1024 cores ne représente que 0.05% de sa puissance de calcul. D'autre part, nous avons démontré expérimentalement que l'opération de mise à jour immédiate du nombre de tâches actives utilisateur (l'indicateur *T*) ne pose pas de problème de passage à l'échelle.

Nous avons démontré expérimentalement l'intérêt de répartir l'exécution d'un service système en terme de scalabilité et de réduction de coût (c.f : section 5.3.2). En particulier, la réalisation répartie de l'appel système *fork* permet de passer à l'échelle ce service sur le processeur TSAR à 512 cores. Dans cette configuration, la création d'un nouveau processus fils placé sur un core à distance maximale du core où le processus père s'exécute n'augmente le coût de l'opération que de 14% par rapport à une création locale.

Enfin, au-delà des réponses apportées aux questions que nous avons soulevées au début de ce manuscrit, nos travaux permettent de donner un argument tangible concernant l'avenir des processeurs many-cores à mémoire partagée cohérente garantie par le matériel (dont le processeur TSAR n'est qu'un exemple). Il s'agit d'un enjeu très important, car le paradigme de programmation en mémoire partagée est, de loin, le plus utilisé dans l'industrie logicielle. Avoir des solutions permettant de prendre en charge le problème difficile de la localité des accès mémoire sans impacter la portabilité des applications existantes est primordial pour encourager une transition rapide vers l'adoption des architectures many-cores que ça soit dans le domaine de l'embarqué (p. ex : infrastructure de télécommunication, traitement d'images), le domaine du HPC (p. ex : simulations numériques des phénomènes physiques) et dans le domaine des serveurs haut-de-gamme (p. ex : serveurs web, systèmes de réservation, cloud computing).

6.2 Perspectives

6.2.1 Processus hybrides et l'équilibrage de charge

Nous avons implémenté dans le noyau d'ALMOS la politique d'équilibrage de charge et la réalisation répartie de la migration de threads, présentées toutes les deux dans la section 4.3.4.

Actuellement, cette migration est opérationnelle, mais pas pour les processus hybrides. Par manque de temps, nous avons fait le choix pragmatique de différer l'évaluation de l'opération de migration pour les processus hybrides. Nous pensons qu'il est intéressant d'étudier expérimentalement le coût d'une opération d'équilibrage de charge avec et sans migration de données et d'étudier son impact sur les performances en prenant en compte le temps moyen de l'exécution d'une tâche, la distance du core cible par rapport au core initial et la disponibilité mémoire du nœud cc-NUMA cible.

6.2.2 DQDT et stratégies de prise de décision

Nous avons implémenté dans le noyau d'ALMOS l'ensemble des politiques d'allocation de ressources que nous avons proposées dans la section 4.3 afin de démontrer comment l'infrastructure DQDT peut être utilisée. Ces politiques concernent l'allocation mémoire, le placement de tâches et l'équilibrage de charge ; et sont opérationnelles dans le noyau d'ALMOS. D'une manière plus générale, l'introduction de l'infrastructure de décision DQDT ouvre une opportunité pour l'étude de différentes politiques d'allocation, de réservation de ressources et d'équilibrage de charge que le noyau peut mettre en place par classe ou domaine d'applications afin de tirer un meilleur profit des architectures many-cores. Bien que nous avions proposé la DQDT dans le contexte de nos travaux sur la conception d'un noyau monolithique pour many-cores, nous pensons qu'elle peut être réutilisée dans d'autres contextes dans lesquels, il y a un besoin de représenter la disponibilité des ressources physiquement distribuées afin de réaliser une prise de décision décentralisée quant à l'allocation et la gestion de ces ressources.

6.2.3 Possibilités offertes par le concept de processus hybride

Nous avons discuté, dans la section 4.5.4, les nouvelles opportunités offertes par le concept de de processus hybrides, tels que : (i) la migration progressive des pages physiques d'un processus hybride après son changement de nœud cc-NUMA; (ii) la garantie de protection avec une granularité de thread tout en facilitant la communication inter-threads via la mémoire partagée; (iii) la minimisation du coût des défauts de pages pour la partie privée de l'espace virtuel d'un processus hybride; (iv) la gestion dynamique et sans surcoût de la taille de la pile d'un thread; et (v) la communication à moindre coût (en mémoire partagée) entre le noyau et les run-times utilisateur avec la granularité de thread. Nous pensons que l'introduction du concept de processus hybride représente un cadre riche et prometteur pour poursuivre nos travaux sur l'évolution des systèmes d'exploitation à base de noyaux monolithiques.

6.2.3.1 Étendre le contrôle du noyau

Le contrôle des accès mémoire avec la granularité de thread, rendu possible grâce au modèle de processus hybride, permet à un noyau monolithique d'étendre son contrôle sur le matériel en contrôlant le protocole de cohérence de caches. En particulier, le noyau peut décharger le matériel de la responsabilité de maintenir la cohérence de caches pour les lignes d'une page physique associée à une partie locale (lecture seule) ou à une partie privée (lecture/écriture) de l'espace d'adressage virtuel d'un processus hybride. Cela permet de : (i) contrôler la politique d'écriture (write-through/write-back) du cache L1 de chaque core exécutant la même application multi-threads, selon la nature des lignes (cohérentes/non-cohérentes); (ii) mieux utiliser la capacité de caches (p. ex : un cache L2 n'a plus besoin de garder l'inclusivité pour les lignes non-cohérente); et (iii) améliorer les performances du système mémoire de l'architecture matérielle en réduisant le coût des commandes concernant les lignes non-cohérente. Nous avons définit un protocole, nommé ODCCP (On-Demande Cache Coherence Protocol), permettant au noyau d'ALMOS de contrôler le protocole de cohérence du processeur TSAR. Le volet matériel de ce protocole a été réalisé par

Clément Devigne (stage de fin d'études de Master SESI à l'UPMC). Nous allons poursuivre nos travaux de recherche afin de finaliser l'étude expérimentale du gain apporté par une telle approche.

6.2.3.2 Paradigme de programmation PGAS

Comme nous l'avons présenté dans la section 4.5.3, l'espace d'adressage virtuel d'un processus hybride est partitionné en trois parties : privée, locale et globale. Avoir une partie privée présente un certain nombre de propriétés importantes tels que : (i) l'isolation, (ii) le renfoncement de la localité des accès mémoire, (iii) la réduction de coût des défauts de pages, et (iv) l'absence des opérations coûteuses pour maintenir la cohérence des TLBs (TLB shootdown). À ces propriétés s'ajoute la possibilité de décharger le matériel du maintien de la cohérence des lignes de caches appartenant aux pages physiques associées à cette partie privée. Cela rend l'utilisation de cette partie privée attractive de point de vue performance. Par conséquent, il est souhaitable que les programmeurs, les compilateurs et/ou les run-times des langages de programmation maximisent l'utilisation de la partie privée et minimisent l'utilisation de la partie globale prévue essentiellement pour simplifier les communications et les synchronisations inter-processus hybride.

Un moyen pour atteindre cet objectif est l'utilisation d'un paradigme de programmation rendant explicite l'existence, dans l'espace d'adressage virtuel, de deux parties : partagée et privée. Un langage de programmation suivant ce paradigme permet à la fois au programmeur d'exprimer la visibilité de ses structures de données allouées statiquement ou dynamiquement et au compilateur/run-time de les associer à la bonne partie de l'espace d'adressage virtuel selon leurs visibilités. En particulier, le paradigme PGAS (Partitionned Global Adress Space) [12] utilisé dans le domaine du HPC répond parfaitement à ce besoin. Dans ce cas, le concept de processus hybrides peut être vu comme un support natif apporté par le noyau du système d'exploitation pour de tels paradigmes. Nous pensons que cette approche est prometteuse à la fois pour simplifier la conception d'un compilateur/run-time de langage PGAS comme l'UPC (Unified Parallel C) et pour tirer les meilleures performances d'un processeur many-core à mémoire partagée cohérente.

6.2.4 Passage à l'échelle d'autres sous-systèmes du noyau

Parmi les perspectives que nous voyons pour nos travaux de thèse, il faut étudier la scalabilité du sous-système gestionnaire de fichiers. Dans les systèmes d'exploitation de type UNIX, l'abstraction principale offerte aux utilisateurs est le fichier. Plusieurs ressources systèmes sont vues et accédées par les applications utilisateur en utilisant l'interface d'accès aux fichiers. L'abstraction de fichier offerte aux applications utilisateur peut représenter : un fichier régulier sur une partition de disque local (p. ex : Ext2), un fichier régulier en mémoire (p. ex : TmpFS, RAMFS), un fichier régulier existant sur le disque d'une machine lointaine (p. ex : NFS), un fichier spécial désignant un périphérique matériel (p. ex : DevFS) ou encore, un fichier régulier désignant une structure de données du noyau (p. ex : SysFS). Pour pouvoir gérer cette diversité un système de fichiers abstrait nommé VFS (Virtual File System) est utilisé dans les noyaux monolithiques. Nous avons introduit dans le noyau d'ALMOS, un VFS dont l'implémentation suit une approche similaire à ce qui existe dans d'autres noyaux monolithiques tels que Linux. Le but de ce VFS est d'abstraire et d'accéder, d'une manière uniforme, aux 4 systèmes de fichiers sous-jacents : Ext2, FAT32, DevFS et SysFS, mais nous ne l'avons pas pensé pour être scalable.

Bien que des travaux aient été proposés visant à améliorer la scalabilité du VFS en utilisant des techniques de type RCU (Read-Copy-Update) pour augmenter le parallélisme d'accès au cache de méta-données [109] (2004) et au cache de données [68] (2006), nous pensons que cela n'est pas suffisant pour passer à l'échelle sur un processeur many-core, car les problèmes de la localité des

accès mémoire et la contention sur le cache LLC n'ont pas été pris en compte. Nous pensons qu'il est nécessaire d'investiguer une approche répartie pour concevoir un VFS passant à l'échelle et que le noyau d'ALMOS, avec ses infrastructures distribuées et sa gestion décentralisée des ressources, représente un cadre adapté pour introduire et expérimenter une telle approche.

6.2.5 Dissémination technique et scientifique

Nos différentes contributions présentées dans cette thèse sont disponibles et accessibles sur le site web du projet ALMOS [7]. Le projet ALMOS est activement utilisé dans les projets de recherche suivants : (i) TSAR [9] - un projet européen (MEDEA+) visant à concevoir un processeur many-core scalable à mémoire partagée cohérente; (ii) SHARP [10] - un projet européen (MEDEA+) visant à construire une machine HPC intégrant le support nécessaire à plusieurs modèles de programmation parallèle; et (iii) TSUNAMY [11] - un projet national (ANR) visant à concevoir une solution mixte logicielle/matérielle pour renforcer la sécurité des données traitées par un processeur many-core. Sur le plan de l'enseignement, le projet ALMOS constitue le matériel pédagogique de deux modules d'enseignement du Master SESI à l'UPMC. Cela montre que le projet ALMOS répond bien à un besoin existant à la fois dans la recherche et dans l'enseignement.

Aujourd'hui, les statistiques d'accès au site du projet ALMOS indiquent que l'effort de dissémination que nous avons initié à travers nos publications [2,3,4,5,6] et l'ouverture au public de l'accès au système d'exploitation ALMOS sous licence open-source, donne des résultats prometteurs. En effet, après 14 mois, le site web du projet a été accédé depuis 24 pays dont les 3 premiers de la liste sont respectivement : la France (12 régions), les États-Unis (9 états), la Chine (4 villes). Cela donne une indication encourageante quant à la visibilité du projet ALMOS. Nous allons continuer nos efforts de valorisation à travers d'avantage de publications et la recherche de nouvelles opportunités de coopération.

Bibliographie

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. 2000. Clock rate versus IPC: the end of the road for conventional microarchitectures. In Proceedings of the 27th annual international symposium on Computer architecture (ISCA'00). ACM, New York, NY, USA, 248-259.

[2] Ghassan Almaless and Franck Wajsburt. Does shared-memory, highly multi-threaded, single-application scale on many-cores? In Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism, Berkeley, California, USA, Jun 2012.

[3] Almaless, G.; Wajsburt, F., "On the scalability of image and signal processing parallel applications on emerging cc-NUMA many-cores," Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on , vol., no., pp.1,8, 23-25 Oct. 2012.

[4] Ghassan Almaless. ALMOS : un système d'exploitation pour manycores en mémoire partagée cohérente. In Proceedings of the 8th French Conference on Operating Systems (CFSE), the French chapter of ACM-SIGOPS, GDR ARP, Saint-Malo, France, 2011.

[5] Ghassan Almaless and Franck Wajsburt. On the Impact of Many-Cores Small Caches on the Scalability of Shared-Memory Highly Multi-Threaded Single-Applications". In Poster Session of the 3rd Asia-Pacific Workshop on Systems, Seoul, South Korea, 2012.

[6] Ghassan Almaless and Franck Wajsburt. ALMOS: Advanced Locality Management Operating System for cc-NUMA Many-Cores”. In Proceedings of the 5th national seminar of GDR SoC-SIP, Lyon, France, 2011.

[7] Advanced Locality Management Operating System. Project Home Page. web: www.almos.fr [Accessed 08 Sep 2013].

[8] Alain Greiner. Tsar: a scalable, shared memory, many-cores architecture with global cache coherence. In 9th International Forum on Embedded MPSoC and Multicore (MPSoC'09), Savannah, Georgia, USA, 2009. IEEE Press.

[9] Tera-Scale ARchitecture. Project Home Page (LIP6). web: https://www-asim.lip6.fr/trac/tsar [Accessed 08 Sep 2013].

[10] Scalable Heterogeneous ARchitecture for Processing. MEDEA+ European project #CA109. web: http://www.catrene.org/web/projects/projects_call456.php [Accessed 08 Sep 2013].

[11] TSUNAMY. Project Home Page. web: https://www-asim.lip6.fr/trac/tsunamy [Accessed 22 Nov 2013].

[12] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and performance using partitioned global address space languages. In Proceedings of the 2007 international workshop on Parallel symbolic computation (PASCO '07). ACM, New York, NY, USA, 24-32.

[13] Bell, S.; Edwards, B.; Amann, J.; Conlin, R.; Joyce, K.; Leung, V.; MacKay, J.; Reif, M.; Liewei Bao; Brown, J.; Mattina, M.; Chyi-Chang Miao; Ramey, C.; Wentzlaff, D.; Anderson, W.; Berger, E.; Fairbanks, N.; Khan, D.; Montenegro, F.; Stickney, J.; Zook, J., "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International , vol., no., pp.88,598, 3-7 Feb. 2008.

[14] Tilera Announces the World's First 100-core Processor. Tilera Press Releases, SAN JOSE, Calif. - Oct. 26, 2009. Web: http://www.tilera.com/about_tilera/press-releases/tilera-announces-worlds-first-100-core-processor [Accessed 08 Sep 2013].

[15] James Charles, Preet Jassi, Narayan S. Ananth, Abbas Sadat, and Alexandra Fedorova. 2009. Evaluation of the Intel\&\#174; Core\&\#153; i7 Turbo Boost feature. In Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '09). IEEE Computer Society, Washington, DC, USA, 188-197.

[16] Conway, P.; Kalyanasundharam, N.; Donley, G.; Lepak, K.; Hughes, B., "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," Micro, IEEE , vol.30, no.2, pp.16,29, March-April 2010.

[17] George Chrysos. 2012. Knights Corner, Intel’s first Many Integrated Core (MIC) Architecture Product. In Proceedings of Hot Chips: A Symposium on High Performance Chips (HC24), Flint Center, Cupertino, CA, USA.

[18] TILE-Gx Processor Family, Tilera Corp. web: http://www.tilera.com/products/processors /TILE-Gx_Family [Accessed 08 Sep 2013].

[19] Tilera to Double Processor Cores Every Two Years. Tilera Press Releases. SANTA CLARA and SAN FRANCISCO Calif - Jun. 22, 2010. web: http://www.tilera.com/about_tilera/ press-releases/tilera-double-processor-cores-every-two-years [Accessed 08 Sep 2013].

[20] Andi Kleen. An NUMA API for Linux. SUSE Labs, Aug 2004. web: http://www.halobates.de/numaapi3.pdf [Accessed 08 Sep 2013].

[21] Memory and Thread Placement Optimization Developer's Guide. Oracle, Part No: E35301-01, Oct 2012.

[22] Ronald Christopher Unrau. 1993. Scalable Memory Management Through Hierarchical Symmetric Multiprocessing. Ph.D. Dissertation. University of Toronto, Toronto, Ont., Canada, Canada. UMI Order No. GAXNN-82787.

[23] G.S. Almasi and A. Gottlieb. 1989. Highly Parallel Computing. Benjamin-Cummings Publ. Co., Inc., Redwood City, CA, USA.

[24] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, A. Norton, and j. Weiss, "The IBM Research Parallel Processor Prototype (RP3) : Introduction and Architecture", IEEE, Proceedings, ICPP 1985 (Aug. 1985) pp764-771.

[25] BBN Advanced Computers. The Uniform System approach to programming the Butterfly parallel processor. Rep. 6149, Version 2, BBN Advanced Computers, Cambridge. Mass., June 1986. pp. 3-5.

[26] Zvonko G. Vranesic, Michael Stumm, David M. Lewis, and Ron White. 1991. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. Computer 24, 1 (January 1991), 72-79.

[27] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. 1995. Hierarchical clustering: a structure for scalable multiprocessor operating system design. J. Supercomput. 9, 1-2 (March 1995), 105-134.

[28] Benjamin Gamsa, Orran Krieger, and Michael Stumm. 1994. Optimizing IPC Performance for Shared-Memory Multiprocessors. In Proceedings of the 1994 International Conference on Parallel Processing - Volume 01 (ICPP '94), Vol. 1. IEEE Computer Society, Washington, DC, USA, 208-211.

[29] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. 1999. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In Proceedings of the third symposium on Operating systems design and implementation (OSDI '99). USENIX Association, Berkeley, CA, USA, 87-100.

[30] Ben Gamsa. Tornado: Maximizing Locality and Concurrency in a Shared-Memory Multiprocessor Operating System, Ph.D. thesis, 1999.

[31] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. 2006. K42: building a complete operating system. In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06). ACM, New York, NY, USA, 133-145.

[32] Parsons, E.; Gamsa, B.; Krieger, O.; Stumm, M., "(De-)clustering objects for multiprocessor system software," Object-Orientation in Operating Systems, 1995., Fourth International Workshop on , vol., no., pp.72,81, 14-15 Aug 1995.

[33] Jonathan Appavoo. 2005. Clustered Objects. Ph.D. Dissertation. University of Toronto, Toronto, Ont., Canada, Canada.

[34] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. 2007. Experience distributing objects in an SMMP OS. ACM Trans. Comput. Syst. 25, 3, Article 6 (August 2007).

[35] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. 1992. The Stanford Dash Multiprocessor. Computer 25, 3 (March 1992), 63-79.

[36] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 1994. The Stanford FLASH multiprocessor. SIGARCH Comput. Archit. News 22, 2 (April 1994), 302-313.

[37] A. Eiriksson, J. Keen, A. Silbey, S. Venkataraman, and M. Woodacre. 1997. Origin System Design Methodology and Experience: lM-gate ASICs and Beyond. In Proceedings of the 42nd IEEE International Computer Conference (COMPCON '97). IEEE Computer Society, Washington, DC, USA, 157-.

[38] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, P. McHardy, S. Srbljic, M. Stumm, Z. Vranesic, and Z. Zilic. 2000. The NUMAchine Multiprocessor. In Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing (ICPP '00). IEEE Computer Society, Washington, DC, USA, 487-.

[39] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler activations: effective kernel support for the user-level management of parallelism. In Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP '91). ACM, New York, NY, USA, 95-109.

[40] Shekhar Borkar. 2007. Thousand core chips: a technology perspective. In Proceedings of the 44th annual Design Automation Conference (DAC '07). ACM, New York, NY, USA, 746-749.

[41] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. 2010. The 48-core SCC Processor: the Programmer's View. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11.

[42] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, 29-44.

[43] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-level interprocess communication for shared memory multiprocessors. ACM Trans. Comput. Syst. 9, 2 (May 1991), 175-198.

[44] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. 1991. First-class user-level threads. SIGOPS Oper. Syst. Rev. 25, 5 (September 1991), 110-121.

[45] David Wentzlaff and Anant Agarwal. 2009. Factored operating systems (fos): the case for a scalable operating system for multicores. SIGOPS Oper. Syst. Rev. 43, 2 (April 2009), 76-85.

[46] David Wentzlaff, Charles Gruenwald, III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An operating system for multicore and clouds: mechanisms and implementation. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). ACM, New York, NY, USA, 3-14.

[47] Lamia Youseff, Nathan Beckmann, Harshad Kasture, Charles Gruenwald, David Wentzlaff, and Anant Agarwal. 2012. The case for elastic operating system services in fos. In Proceedings of the 49th Annual Design Automation Conference (DAC '12). ACM, New York, NY, USA, 265-270.
[48] Wentzlaff, D., Gruenwald III., C., Belay, A., Kasture, H., Youseff, L., Miller, J.E., Modzelewski, K., Agarwal, A.: Fleets: Scalable services in a factored operating system. Technical Report (TR-2011-012) MIT-CSAIL.

[49] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. 2011. A case for scaling applications to many-core with OS clustering. In Proceedings of the sixth conference on Computer systems (EuroSys '11). ACM, New York, NY, USA, 61-76.

[50] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: running commodity operating systems on scalable multiprocessors. ACM Trans. Comput. Syst. 15, 4 (November 1997), 412-447.

[51] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. 1999. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP '99). ACM, New York, NY, USA, 154-169.

[52] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. 2008. Corey: an operating system for many cores. In Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08). USENIX Association, Berkeley, CA, USA, 43-57.

[53] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: space-time partitioning in a manycore client OS. In Proceedings of the First USENIX conference on Hot topics in parallelism (HotPar'09). USENIX Association, Berkeley, CA, USA, 10-10.

[54] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. 2013. Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In Proceedings of the 50th Annual Design Automation Conference (DAC '13). ACM, New York, NY, USA, , Article 76 , 10 pages.

[55] Pierre Guironnet de Massas. Étude de méthodes et mécanismes pour un accès transparent et efficace aux données dans un système multiprocesseur sur puce, Ph.D. thesis, TIMA, Polytechnic Institute of Grenoble, France, ISBN 978-2-84813-145-0, 2009.

[56] Bryan Cantrill and Jeff Bonwick. 2008. Real-world concurrency. Commun. ACM 51, 11 (November 2008), 34-39.

[57] Yan Cui; Yu Chen; Yuanchun Shi; Qingbo Wu, "Scalability comparison of commodity operating systems on multi-cores," Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on , vol., no., pp.117,118, 28-30 March 2010.

[58] John H. Baldwin. 2002. Locking in the multithreaded FreeBSD kernel. In Proceedings of the BSD Conference 2002 on BSD Conference (BSDC'02). USENIX Association, Berkeley, USA, 4-4.

[59] C. Lameter. Effective synchronization on Linux/NUMA systems. Proceedings of the May 2005 Gelato Federation Meeting. web: http://www.lameter.com/gelato2005.pdf [Accessed 08 Sep 2013].

[60] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In Ottawa Linux Symposium (July 2001). web: http://www.rdrop.com/users/paulmck/RCU/rclock_OLS.2001.05.01c.pdf [Accessed 08 Sep 2013].

[61] Paul E. McKenney and Jonathan Walpole. 2008. Introducing technology into the Linux kernel: a case study. SIGOPS Oper. Syst. Rev. 42, 5 (July 2008), 4-17.

[62] Arcangeli, A., Cao, M., McKenney, P. E., and Sarma, D. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track) (June 2003), USENIX Association, pp. 297--310.

[63] Josh Aas. 2005. Understanding the Linux 2.6.8.1 CPU Scheduler. Silicon Graphics Technical Report. web: http://joshaas.net/linux [Accessed 08 Sep 2013].

[64] Martin J. Bligh, Matt Dobson, Darren Hart and Gerrit Huizenga. Linux on NUMA Systems. In Proceedings of the Linux Symposium 2004, pages 89-102, Ottawa, Ontario, July 2004.

[65] Ray Bryant and John Hawkes. Linux Scalability for Large NUMA Systems. In Proceedings of the Linux Symposium 2003, pages 83-96, Ottawa, Ontario, July 2003.

[66] Ray Bryant, Jesse Barnes, John Hawkes, Jeremy Higdon, and Jack Steiner. Scaling Linux to the extreme: From 64 to 512 processors. In Proceedings of the Linux Symposium 2004, pages 133-148, Ottawa, Ontario, July 2004.

[67] C. Gough, S. Siddha, and K. Chen. Kernel scalability--expanding the horizon beyond fine grain locks. In Proceedings of the Linux Symposium 2007, pages 153-165, Ottawa, Ontario, June 2007.

[68] Nick Piggin. A Lockless Pagecache in Linux -- Introduction, Progress, Performance. In Proceedings of the Linux Symposium 2006, pages 241-254, Ottawa, Ontario, July 2007.

[69] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of Linux scalability to many cores. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, Berkeley, CA, USA, 1-8.

[70] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII). ACM, New York, NY, USA, 199-210.

[71] Schwan, K.; Jones, A.K., "Special Feature: Specifying Resource Allocaton for the Cm* Multiprocessor," Software, IEEE , vol.3, no.3, pp.60,70, May 1986.

[72] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. 1977. Cm*: a modular, multi-microprocessor. In Proceedings of the June 13-16, 1977, national computer conference (AFIPS '77). ACM, New York, NY, USA, 637-644.

[73] Henrik Löf and Sverker Holmgren. 2005. affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system. In Proceedings of the 19th annual international conference on Supercomputing (ICS '05). ACM, New York, NY, USA, 387-392.

[74] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. 2010. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. J. Parallel Distrib. Comput. 70, 12 (December 2010), 1204-1219.

[75] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware monitors for dynamic page migration. J. Parallel Distrib. Comput. 68, 9 (September 2008), 1186-1200.

[76] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. 2000. Extending OpenMP for NUMA machines. In Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM) (Supercomputing '00). IEEE Computer Society, Washington, DC, USA, , Article 48.

[77] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. In Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing (ICPP '00). IEEE Computer Society, Washington, DC, USA, 95-.

[78] Jaydeep Marathe and Frank Mueller. 2006. Hardware profile-guided automatic page placement for ccNUMA systems. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06). ACM, New York, NY, USA, 90-99.

[79] Juan A. Lorenzo-Castillo, Juan C. Pichel, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro. 2013. A flexible and dynamic page migration infrastructure based on hardware counters. J. Supercomput. 65, 2 (August 2013), 930-948.

[80] Blagodurov, S. and Fedorova, A. 2011. User-level scheduling on NUMA multicore systems under Linux 2011, pages 81-91, Ottawa, Ontario, June 2011.

[81] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2013. ADAPT: A framework for coscheduling multithreaded programs. ACM Trans. Archit. Code Optim. 9, 4, Article 45 (January 2013), 24 pages.

[82] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07). IEEE Computer Society, Washington, DC, USA, 13-24.

[83] Broquedis, F.; Aumage, O.; Goglin, B.; Thibault, S.; Wacrenier, P.-A.; Namyst, R., "Structuring the execution of OpenMP applications for multicore architectures," Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on , vol., no., pp.1,10, 19-23 April 2010.

[84] McCurdy, C.; Vetter, J., "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on , vol., no., pp.87,96, 28-30 March 2010.

[85] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. Concurr. Comput. : Pract. Exper. 22, 6 (April 2010), 685-701.

[86] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: a memory profiler for NUMA multicore systems. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, Berkeley, CA, USA, 5-5.

[87] J. Levon. 2009. OProfile manual. web: http://oprofile.sourceforge.net/doc/index.html [Accessed 08 Sep 2013].

[88] A. Cox and R. Fowler. 1989. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum. In Proceedings of the twelfth ACM symposium on Operating systems principles (SOSP '89). ACM, New York, NY, USA, 32-44.

[89] C. Scheurich and M. Dubois. 1989. Dynamic Page Migration in Multiprocessors with Distributed Global Memory. IEEE Trans. Comput. 38, 8 (August 1989), 1154-1163.

[90] M. A. Holliday. 1989. Reference history, page size, and migration daemons in local/remote architectures. In Proceedings of the third international conference on Architectural support for programming languages and operating systems (ASPLOS III). ACM, New York, NY, USA, 104-112.

[91] W. Bolosky, R. Fitzgerald, and M. Scott. 1989. Simple but effective techniques for NUMA memory management. In Proceedings of the twelfth ACM symposium on Operating systems principles (SOSP '89). ACM, New York, NY, USA, 19-31.

[92] Richard P. LaRowe, Jr., James T. Wilkes, and Carla S. Ellis. 1991. Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '91). ACM, New York, NY, USA, 122-132.

[93] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1994. Scheduling and page migration for multiprocessor compute servers. In Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS VI). ACM, New York, NY, USA, 12-24.

[94] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. 1992. The DASH prototype: implementation and performance. In Proceedings of the 19th annual international symposium on Computer architecture (ISCA '92). ACM, New York, NY, USA, 92-103.

[95] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating system support for improving data locality on CC-NUMA compute servers. In Proceedings of the seventh international conference on Architectural support for programming languages and operating systems (ASPLOS VII). ACM, New York, NY, USA, 279-289.

[96] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. 1995. Complete Computer System Simulation: The SimOS Approach. IEEE Parallel Distrib. Technol. 3, 4 (December 1995), 34-43.

[97] Julita Corbalan, Xavier Martorell, and Jesus Labarta. 2003. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In Proceedings of the 17th annual international conference on Supercomputing (ICS '03). ACM, New York, NY, USA, 121-129.

[98] Livio Soares and Michael Stumm. 2010. FlexSC: flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, Berkeley, CA, USA, 1-8.

[99] Multithreading in the Solaris Operating Environment. A Technical White Paper. Sun Microsystems Inc, 2002.

[100] Erik Hagersten, Anders Landin, and Seif Haridi. 1992. DDM: A Cache-Only Memory Architecture. Computer 25, 9 (September 1992), 44-54.

[101] Henry Burkhardt 111 et al. Overview of the KSR1 Computer System. Technical Report KSR-TR- 9202001, Kendall Square Research, Boston, February 1992.

[102] Fredrik Dahlgren and Josep Torrellas. 1999. Cache-Only Memory Architectures. Computer 32, 6 (June 1999), 72-79.

[103] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS X). ACM, New York, NY, USA, 211-222.

[104] Zhang, M.; Asanovic, K., "Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors," Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on , vol., no., pp.336,345, 4-8 June 2005.

[105] Noel Eisley, Li-Shiuan Peh, and Li Shang. 2008. Leveraging on-chip networks for data cache migration in chip multiprocessors. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08). ACM, New York, NY, USA, 197-207.

[106] Mohammad Hammoud, Sangyeun Cho, and Rami Melhem. 2009. Dynamic cache clustering for chip multiprocessors. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). ACM, New York, NY, USA, 56-67.

[107] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: near-optimal block placement and replication in distributed caches. In Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09). ACM, New York, NY, USA, 184-195.

[108] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. IEEE Micro 26, 4 (July 2006), 18-31.

[109] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. 2004. Scaling dcache with RCU. Linux J. 2004, 117 (January 2004), 3-.

[110] GOMP: An OpenMP implementation for GCC. web: http://gcc.gnu.org/projects/gomp [Accessed 08 Sep 2013].

[111] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. 2009. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)* (IISWC '09). IEEE Computer Society, Washington, DC, USA, 198-207.

[112] The Phoenix System for MapReduce Programming. Project Home Page. web: http://mapreduce.stanford.edu/ [Accessed 08 Sep 2013].

[113] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008), 107-113.

[114] GCC, the GNU Compiler Collection. Project Home Page. web: http://gcc.gnu.org/ [Accessed 08 Sep 2013].

[115] Kenneth C. Knowlton. 1965. A fast storage allocator. Commun. ACM 8, 10 (October 1965), 623-624.

[116] Donald E. Knuth. 1997. The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[117] Jeff Bonwick. 1994. The slab allocator: an object-caching kernel memory allocator. In Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference (USTC'94), Vol. 1. USENIX Association, Berkeley, CA, USA, 6-6.

[118] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39-59.

[119] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1990. Lightweight remote procedure call. ACM Trans. Comput. Syst. 8, 1 (February 1990), 37-55.

[120] Advanced Configuration and Power Interface Specification. Revision 5.0, p 154. December 2011.

[121] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. 1987. The duality of memory and communication in the implementation of a multiprocessor operating system. In Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87). ACM, New York, NY, USA, 63-76.

[122] Maurice J. Bach. 1986. The Design of the UNIX Operating System. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[123] Tera-Scale Multi-core Processor ARchitecture, MEDEA+ funded project #2A718. Project Profile. web : http://www.catrene.org/web/downloads/profiles_medea/2A718-TSAR-profile-out MEDEA2%20%2823-6-10%29.pdf [Accessed 08 Sep 2013].

[124] Tera-Scale Multi-core Processor ARchitecture, MEDEA+ funded project #2A718. Project Results. web : http://www.catrene.org/web/downloads/results_medea/2A718-TSAR-result-outMEDEA%20(13-3-12).pdf [Accessed 08 Sep 2013].

[125] Miro Panades, I.; Greiner, A.; Sheibanyrad, A., "A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach," *Nano-Networks and Workshops, 2006. NanoNet '06. 1st International Conference on* , vol., no., pp.1,5, Sept. 2006.

[126] Buchmann, R.; Greiner, A., "A fully static scheduling approach for fast cycle accurate systemC simulation of MPSoCs," Microelectronics, 2007. ICM 2007. Internatonal Conference on , vol., no., pp.101,104, 29-31 Dec. 2007.

[127] SoCLib - An Open Platform for Virtual Prototyping MP-SoCs. Project Home Page. web: http://www.soclib.fr [Accessed 08 Sep 2013].

[128] D. H. Bailey. 1989. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* (Supercomputing '89). ACM, New York, NY, USA, 234-242.

[129] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. 1994. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems* (ASPLOS VI). ACM, New York, NY, USA, 219-229.

[130] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures* (SPAA '91). ACM, New York, NY, USA, 3-16.

[131] Woo, S. C., Singh, J. P., and Hennessy, J. L. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Technical Report CSL-TR-93-593, Stanford University, December 1993.

[132] J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. 1993. A parallel adaptive fast multipole method. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing* (Supercomputing '93). ACM, New York, NY, USA, 54-65.

[133] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture* (ISCA '95). ACM, New York, NY, USA, 24-36.

[134] J. Stone. An efficient library for parallel ray tracing and animation. Technical report, In Intel Supercomputer Users Group Proceedings, 1995.