Etude de performances sur processeurs multicoeur : environnement d'exécution événementiel efficace et étude comparative de modèles de programmation

Sylvain Geneves

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Sylvain Genevès

Thèse dirigée par **Vivien Quéma**
et co-encadrée par **Renaud Lachaize**

préparée au sein **LIG**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Améliorations de performances des systèmes multi-cœur : environnement d'exécution événementiel efficace et étude comparative de modèles de programmation

Thèse soutenue publiquement le **5 Avril 2013**,
devant le jury composé de :

Prof. Didier Donsez
Professeur à Université de Grenoble 1, Président
Dr Gilles Muller
DR INRIA, Rapporteur
Dr Laurent Réveillère
MCF HDR à ENSEIRB-MATMECA, Rapporteur
Dr Sonia Ben Mokhtar
CR CNRS, Examinatrice
Prof. Vivien Quéma
Professeur à Grenoble INP, Directeur de thèse
Dr Renaud Lachaize
MCF à Université de Grenoble 1, Co-Encadrant de thèse

Remerciements

Je tiens tout d'abord à remercier Vivien Quéma, Professeur à Grenoble INP, et Renaud Lachaize, Maître de Conférences à l'Université Grenoble 1, pour m'avoir encadré, guidé, supporté et soutenu tout au long de cette thèse, et notamment dans les moments les plus difficiles.

Merci également aux membres du jury. Tout particulièrement, je remercie Laurent Réveillère, Maître de Conférences à l'ENSEIRB, et Gilles Muller, Directeur de recherche à l'INRIA, d'avoir accepté d'être rapporteurs de cette thèse et d'avoir évalué mes travaux de façon constructive. Mes remerciements vont aussi à Didier Donsez, Professeur à l'Université Grenoble 1, et à Sonia Ben Mokhtar, Chargée de Rercherche au CNRS, pour leur participation à l'évaluation de cette thèse.

Merci aux équipes SARDES et AMAZONES de m'avoir accueilli dans l'ambiance chaleureuse qui y règne. Merci à tous ceux qui ont contribué à ma bonne humeur, par tous les moyens :

- Tous les protagonistes du B218 : Fabien, Fabien et Baptiste, pour leurs discussions et leurs riches idées, les concours de propreté de code, les débats animés, sérieux ou non, les concours de mauvaise foi et les parties de Wormux.
- Pierre-Louis et Michael pour leurs discussions hautement philosophiques, et tous les bons moments partagés.
- Gautier, Willy, Jean, Olivier et tous les membres de l'équipe que la machine à café reconnaitra.
- Les membres du CITI : Nicolas, Hervé x2, Léo, Ben, Tanguy, Gaëlle, Stéphane x2, Frédéric, Matthieu, Mickaël et les autres, pour l'ambiance d'échange, les discussions intéressantes, les pots du vendredi soir, les soirées ciné et tout le reste.
- La ville de Lyon pour être à distance raisonnable de Grenoble.

Enfin, je garde une pensée toute particulière pour tous les membres de ma famille, et particulièrement mes parents et mon frère. Je ne les remercierai jamais assez pour leur soutien et leur accompagnement. Ils ont toujours su me supporter sans jamais perdre patience.

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

Depuis les débuts de l'ère informatique la puissance de calcul des processeurs n'a cessé de croître. Cette augmentation de puissance, qui semble suivre une évolution linéaire, est souvent confondue à tort avec la célèbre loi de Moore [1]. L'augmentation de la fréquence, couplée à des changements architecturaux au sein des processeurs ont permis de maintenir ce gain de puissance. Cependant, ces changements architecturaux, comme par exemple les mécanismes d'exécution dans le désordre, compliquent grandement les processeurs. De plus l'augmentation de la fréquence implique une augmentation de la consommation d'énergie et du dégagement de chaleur, et les processeurs actuels se rapprochent dangereusement de la limite de chaleur que supportent leurs composants. Partant de ce constat, les concepteurs de processeurs ont adopté une autre approche. Ainsi, au lieu de complexifier le cœur d'un processeur, leur idée est de multiplier le nombre de cœurs au sein d'un processeur. Cela permet de continuer à augmenter la puissance de calcul globale d'un processeur, tout en contournant les difficultés liées à l'augmentation de fréquence et à la dissipation thermique.

Au lieu d'accélérer le flot d'exécution du processeur, les multi-cœur créent d'autres flots d'exécution parallèles. Pour tirer parti de cette puissance de calcul additionnelle, il est donc nécessaire d'utiliser ces flots d'exécution supplémentaires. L'arrivée des architectures multi-cœurimplique une refonte massive et profonde des différentes couches logicielles existantes.

Contributions

L'objectif principal de ces travaux est de fournir des pistes pour améliorer le passage à l'échelle des serveurs de données avec le nombre de cœurs. Les serveurs de données sont devenus omniprésents depuis l'avènement du Web. Le principal défi pour ces serveurs est d'être capable de traiter un maximum de clients simultanément, tout en maintenant une certaine qualité de service. La solution communément utilisée est de distribuer les traitements sur plusieurs machines. Il est toutefois possible de réduire les coûts associés à cette solution (notamment le coût d'achat et la consommation énergétique). Pour cela, il faut tirer parti au maximum de chaque machine. Comme ces machines sont multi-cœur, cela revient à exploiter le parallélisme des processeurs multi-cœur. Les serveurs de données traitent souvent des requêtes indépendantes issues de clients différents. L'indépendance entre les traitements des requêtes rend la parallélisation plus aisée. Les serveurs de données constituent donc un bon point de départ pour traiter de l'exploitation du parallélisme en multi-cœur.

Dans le but d'améliorer le passage à l'échelle avec le nombre de cœurs des serveurs de

1. La loi de Moore ne mentionne en rien les performances des processeurs, elle s'énonce comme ceci "la densité des transistors sur une puce double tous les 18 mois".

données, plusieurs étapes sont nécessaires. Ainsi, il nous faut d'abord mieux comprendre les performances des serveurs de données, en mettant un fort accent sur l'aspect multi-cœur. Nous avons analysé le passage à l'échelle de serveurs de données au niveau d'un environnement d'exécution et de différents modèles de programmation. La première contribution se concentre sur le support efficace d'un modèle de programmation concurrent. La deuxième contribution étudie et compare les performances des trois modèles de programmation les plus employés pour implanter des serveurs de données.

Optimisation d'environnement d'exécution. L'environnement d'exécution a un impact important sur les performances des applications. Nous nous concentrons dans ce chapitre sur le modèle événementiel. Nous étudions la spécialisation colorée de ce modèle, où les couleurs sont des annotations qui permettent de tirer parti des architectures parallèles. En effet, nous pensons qu'il s'agit d'une bonne approche pour programmer des applications performantes sur les architectures multi-cœur. La gestion explicite et simplifiée du partage de données présente dans le modèle événementiel coloré nous fait préférer ce choix au modèle threadé. Un environnement d'exécution ne doit pas limiter les performances des applications qui l'utilisent. Notamment, il doit maximiser le passage à l'échelle des applications qui l'utilisent. Nous étudions la structure interne de l'implantation de référence du modèle événementiel avec coloration, Libasync-smp [93]. Nous exposons des faiblesses dans cette structure qui limitent les performances, et le passage à l'échelle des applications. De telle faiblesses apparaissent à deux niveaux. Tout d'abord, au niveau de la gestion de la mémoire. Nous montrons que le faux-partage est une des sources majeures des limitations observées. Nous proposons des mécanismes de gestion mémoire efficaces, dont notamment un nouvel allocateur mémoire, spécialisé pour le modèle de programmation considéré, et efficace en multi-cœur. L'originalité de ces travaux réside dans le fort couplage entre nos optimisations et la gestion de la coloration des événements. Ensuite, nous exposons des faiblesses au niveau des communications inter-cœurs. Nous proposons un mécanisme de réveil, ainsi qu'une technique pour amortir le coût des communications. Nous évaluons et validons ces optimisations sur un serveur Web, ainsi que sur une suite de bancs d'essai de l'état de l'art.

Comparaison de modèles de programmation. Dans la deuxième partie de ces travaux, nous comparons les performances de serveurs Web conçus selon différents modèles de programmation dans un contexte multi-cœur. Plus précisément, nous étudions les trois modèles de programmation les plus utilisés pour mettre en œuvre des serveurs de données : threads, événements et étages. Pour cela, nous choisissons trois implantations efficaces de serveurs Web : Knot, μserver et Watpipe, chacune représentative d'un modèle de programmation. Nous prenons soin au long de cette étude de nous placer dans des configurations telles que les performances de ces implantations soient comparables. Notamment, nous nous assurons que chaque serveur atteint ses performances maximales pour chaque cas étudié. Ainsi, nous pouvons notamment comparer le passage à l'échelle de chaque serveur avec le nombre de cœurs. Un profilage approfondi nous permet d'identifier les causes des différences et des similitudes de performances observées. Nous mettons notamment en avant que le protocole de cohérence de cache empêche toutes les implantations étudiées de passer à l'échelle idéalement. Nous montrons qu'il est possible d'améliorer les performances en tenant compte de la topologie mémoire lors du déploiement des serveurs. L'originalité de ces travaux réside dans les aspects multi-cœur, et notamment l'étude du passage à l'échelle des modèles de programmation représentés.

Organisation du document

Ce document est organisé en six chapitres.

Le premier chapitre présente les évolutions majeures des processeurs depuis les deux dernières décennies. Nous y analysons notamment les raisons de l'adoption des architectures multi-cœur, ainsi que leurs particularités. Puis nous étudions les différents développements envisagés dans le futur pour ces architectures.

Le deuxième chapitre montre l'impact des architectures multi-cœur sur les modèles de programmation. Nous discutons notamment les modèles existants que sont les threads et les événements, et leurs limitations respectives. Nous présentons également les modèles spécifiques aux multi-cœur.

Le troisième chapitre étudie l'impact des architectures multi-cœur sur les environnements d'exécution. Nous étudions ces environnements à deux niveaux : au sein des systèmes d'exploitation et au niveau utilisateur. Après avoir rappelé les différentes architectures des systèmes d'exploitation, nous présentons les systèmes repensés pour le multi-cœur. Enfin, nous décrivons les objectifs, ainsi que les principaux travaux, des environnements d'exécution de niveau utilisateur conçus pour passer à l'échelle.

Le quatrièment chapitre présente de façon générale les contributions de ce document. Nous détaillons tout d'abord les motivations de ces travaux, puis l'approche globale de chaque contribution.

Le cinquième chapitre présente nos travaux sur les performances en multi-cœur d'un environnement d'exécution dédié à la programmation événementielle colorée. Nous proposons dans ce chapitre des techniques efficaces de gestion de la mémoire et des communications inter-cœurs, dans le but de permettre un meilleur passage à l'échelle de l'environnement.

Le sixième chapitre décrit notre étude comparative des trois modèles de programmation couramment utilisés pour implanter des serveurs de données. Nous choisissons tout d'abord trois implantations de serveurs Web efficaces, chacune représentative d'un modèle de programmation. Après avoir identifié un problème de passage à l'échelle sur ces serveurs, nous proposons de lancer plusieurs copies indépendantes d'un serveur en tenant compte de l'architecture matérielle.

Enfin, nous dressons un bilan général des travaux présentés, et nous donnons un ensemble de perspectives et d'évolutions possibles à ces travaux.

Chapitre 1

L'émergence des processeurs multi-cœur

Sommaire

Ce chapitre présente l'évolution architecturale des processeurs. Entre les années 1980 et 2000, la fréquence des processeurs a augmenté de façon drastique (de quelques dizaines de MHz à plusieurs GHz). De plus, des évolutions architecturales ont permis des gains de performances. Cependant, ces deux facteurs deviennent de plus en plus difficilement améliorables. Les fabricants se tournent alors vers de nouvelles solutions. C'est ainsi qu'on voit apparaître dernièrement des architectures multi-cœur.

Dans ce chapitre, nous étudions tout d'abord les principaux composants des processeurs mono-cœur. Cela nous permet de comprendre les raisons de l'évolution vers différents types d'architectures parallèles, que nous présentons. Nous mettons en évidence les avantages et limitations associés à chaque changement architectural.

1.1 Les processeurs mono-cœur

Nous détaillons dans cette section l'architecture des processeurs mono-cœur. Pour cela, nous présentons les principaux composants d'un processeur, ainsi leur organisation. Enfin,

nous concluons cette section avec les techniques de multi-threading, ainsi que leurs limitations. Toute référence au terme *processeur* dans cette section désigne un processeur muni d'un seul cœur.

1.1.1 Principaux composants

Un processeur est organisé en différents composants qui peuvent communiquer entre eux, et parfois avec les composants externes, tels que la mémoire centrale notamment.

Unités de calculs. D'un point de vue logique, un processeur est un ensemble d'unités de calculs opérant sur des registres. Ces unités de calculs sont de différents types et peuvent accomplir des opérations arithmétiques, de calcul flottant ou encore du calcul booléen. Ces calculs sont effectués à partir de valeurs contenues dans des registres. Chaque registre contient un mot de 32 ou 64 bits qui peut être lu ou écrit dans la mémoire centrale ou bien une unité de calcul du processeur.

Traitement d'une instruction. Chaque instruction est lue depuis la mémoire grâce à un registre spécial qui contient l'adresse de la prochaine instruction à exécuter, appelé compteur de programme. Une fois la mémoire lue à cette adresse, plusieurs étapes sont nécessaires avant que l'instruction ne soit exécutée. Il faut en effet tout d'abord décoder l'instruction en plusieurs micro-opérations. Puis les registres doivent être préparés, en chargeant par exemple les opérandes nécessaires. L'opération est ensuite exécutée, au moyen d'une des unités de calcul, qui va placer son résultat dans un des registres disponibles. Si l'instruction le demande, ce résultat est ré-écrit dans la mémoire, ou alors gardé le registre en préparation d'une instruction suivante.

Parallélisme au niveau de l'instruction. L'ensemble de ces micro-opérations peut prendre plusieurs cycles d'horloge. Or, pendant le décodage d'une instruction, les unités de calculs sont disponibles et inutilisées. Aussi, pour augmenter le débit des instructions exécutées, les fabricants ont mis en place des techniques de *pipelining*. L'idée est de pouvoir exécuter plusieurs micro-opérations indépendantes simultanément, par exemple calculer une addition tout en décodant l'instruction suivante. Cela nécessite d'avoir un tampon (pipeline) capable de stocker les micro-opérations de plusieurs instructions, et d'organiser le processeur en étages capables chacun d'exécuter des micro-opérations de façon indépendante.

De tels processeurs capables d'exécuter plusieurs instructions simultanément sont dit super-scalaires. On parle également d'exécution dans le désordre, car les instructions sont réordonnées à la volée par le processeur.

Implications. Il faut bien noter que si l'exécution dans le désordre augmente le débit d'instructions exécutées, cela n'induit aucun bénéfice sur la latence vue pour ces instructions. Au contraire, celle-ci peut augmenter drastiquement. Prenons l'exemple de deux instructions dépendantes, si lorsque la première instruction est décodée pendant que la deuxième est récupérée en mémoire, alors cette dernière sera placée en attente jusqu'à ce que la dépendance soit résolue. Ainsi, la latence de la seconde instruction devient directement dépendante de celle de la première instruction.

De plus, lors d'un branchement, l'aspect séquentiel du flot d'instructions est rompu, ce qui coupe l'approvisionnement en instructions du pipeline. Souvent le pipeline est vidé pour effec-

tuer un branchement de façon sûre. Des techniques de prédiction de branchement permettent néanmoins de continuer l'approvisionnement du pipeline de façon spéculative.

La structure de contrôle pour gérer le pipeline est extrêmement complexe. Il s'agit non seulement de gérer les registres de façon à pouvoir exécuter le plus d'instructions possibles en parallèle, mais aussi et surtout de détecter des dépendances entre instructions (ne sont exécutées en parallèle que des instructions indépendantes). Outre cela, une exécution spéculative est couplée à un mécanisme de prédiction de branchement. Cela implique de pouvoir annuler l'effet de certaines instructions s'il s'avère que la prédiction a échoué.

1.1.2 Caches et accès mémoire

Le "Mur de la mémoire". Si on compare l'augmentation des fréquences des processeurs et celle des mémoires, on se rend compte que, bien que les deux soient exponentielles, la fréquence de la mémoire augmente bien plus lentement que celle des processeurs. Cela a pour conséquence directe une augmentation des temps d'accès à la mémoire vus par le processeur, elle aussi exponentielle. Ce phénomène est appelé "le Mur de la mémoire" (ou Memory Wall) selon Wulf *et al.* [92]. Il faut actuellement de l'ordre de 300 cycles processeur pour accéder à un mot mémoire. Pour masquer de telles latences, les fabricants de processeurs ont conçu des modifications architecturales, et notamment les caches.

Organisation. Le but recherché est de profiter d'une certaine localité des accès mémoire, c'est à dire de pouvoir réutiliser le mot chargé en cache pour économiser un deuxième accès à la mémoire centrale pour la même donnée. Un cache est une mémoire rapide embarquée dans le processeur. Les processeurs utilisent souvent plusieurs niveaux de caches : les plus proches étant plus rapides mais aussi plus petits que ceux plus éloignés du processeur. Le cas le plus classique est celui d'un processeur ayant deux niveaux de cache embarqués, avec parfois un troisième niveau situé sur la carte mère. Un cache contient un sous-ensemble de la mémoire centrale. Les caches peuvent être inclusifs, auquel cas les données du cache de premier niveau sont présentes dans celui de deuxième niveau, ou exclusifs dans le cas où ils contiennent des ensembles de données disjoints. Le contenu d'un cache est déterminé par les accès mémoires du processeur ainsi que par une politique d'éviction. Il existe des caches d'instructions, des caches de données, et des caches unifiés contenant aussi bien des instructions que des données. Il est assez courant d'avoir des caches de données et d'instructions séparés au premier niveau, et des niveaux suivants unifiés, comme montré sur l'exemple de la Figure 1.1.

Chaque entrée d'un cache est organisée en une ligne mémoire, souvent de 64 octets, pouvant contenir plusieurs mots. Lorsqu'un mot est demandé par le processeur et qu'il n'est pas présent en cache, il est chargé depuis la mémoire centrale jusque dans une ligne de cache, à un emplacement calculé en fonction de son adresse et de l'associativité du cache. Une associativité à N voies (N-way) signifie que pour une adresse donnée, N lignes du cache peuvent correspondre. L'associativité permet d'augmenter virtuellement la taille du cache en évitant des fautes de caches dues à des conflits d'emplacement. En revanche, une grande associativité demande au processeur d'explorer plusieurs lignes de cache pour trouver l'emplacement de la valeur cherchée, et de ce fait la latence d'accès au cache. Il s'agit donc d'un compromis. On remarque souvent qu'une faible associativité est choisie pour les caches de premier niveau pour les rendre plus rapides, tandis que les caches de deuxième niveau ont une associativité plus grande pour limiter les fautes de caches menant à des accès à la mémoire centrale. Le remplacement d'une ligne dû au chargement d'une nouvelle valeur implique une éviction des anciennes données contenues dans cette ligne. L'algorithme d'éviction s'apparente très sou-

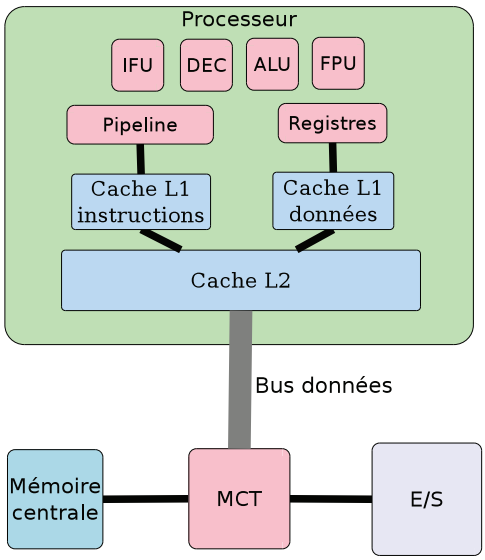

FIGURE 1.1 – Vue d'ensemble simplifiée d'une architecture mono-cœur (typique d'un Intel Pentium mono-cœur), avec ses unités de calculs et ses 2 niveaux de caches. L'accès à la mémoire centrale et les E/S se font de façon centralisée par le biais du contrôleur mémoire (MCT).

vent à une politique LRU [1], désignant ainsi les données accédées les moins récemment. Cela permet de favoriser une certaine localité des programmes. On distingue deux types de localité : la spatiale et la temporelle. Un programme utilisant des données proches en mémoire est dit avoir une bonne localité spatiale. La localité temporelle, quant à elle, peut s'illustrer par un programme réutilisant des adresses déjà accédées.

TLB. Un autre type de cache est présent sur les processeurs. Il s'agit de la TLB [2], un cache pour faciliter la traduction d'adresses. On sait que les systèmes d'exploitation standards utilisent de la mémoire virtuelle, pour améliorer la multiprogrammation, ainsi que pour des raisons de sécurité. En effet, le fait qu'un processus ne puisse accéder qu'à de la mémoire virtuelle permet de contrôler totalement les zones mémoires réelles que ce processus peut effectivement adresser. Pour mettre en œuvre cette virtualisation, un mécanisme de pagination est souvent utilisé. La mémoire est donc découpée en zones de petite taille appelées pages, et le système est en charge de gérer une table des pages, qui contient la correspondance entre des adresses mémoire virtuelles et physiques. Comme cette table des pages est extrêmement souvent accédée, il existe un cache matériel de la table des pages, appelé TLB. Cette TLB contient parfois un marquage par contexte de processus (espace d'adressage). Toutefois, ce marquage n'est pas présent dans les architectures x86 étudiées ici. A chaque changement de processus, la TLB doit être vidée des entrées liées à l'ancien contexte par mesure de protection. Lorsque la TLB n'est pas marquée avec les contextes, l'intégralité de la table est vidée lors d'un changement de contexte.

1. Least Recently Used, moins récemment utilisé.
2. Translation Look-Aside Buffer.

1.1.3 Multi-threading

Un des constats dressés très rapidement à la fois par la communauté système et celle des fabricants de matériel porte sur les changements de contexte. En effet, ceux-ci s'avèrent très coûteux, du fait qu'il faille sauvegarder le contexte d'exécution d'un flot, et en restaurer un autre. Sachant que ce contexte comprend l'ensemble des registres, les correspondances entre une partie de l'adressage virtuel et l'adressage physique, il faut donc écrire le contenu des registres en mémoire pour pouvoir le restaurer. Il faut également, pour maintenir un fonctionnement correct, vider les caches, et notamment la TLB (sauf si elle est marquée avec les contextes). Ainsi on s'assure qu'un flot d'exécution ne pourra pas intervenir sur un contexte autre que le sien. Certains caches sont marqués avec des contextes, à l'instar des TLB citées ci-dessus. Ainsi, lors d'un changement de contexte, seules les lignes correspondant au contexte sortant sont évincées du cache.

Une des solutions possibles est de permettre à la couche matérielle de gérer plusieurs contextes simultanément. Cela implique entre autres de dupliquer les bancs de registres. Ainsi, le processeur peut choisir d'utiliser tel ou tel duplicat et ainsi changer de contexte à la volée. Cette technique est appelée *Multi-Threading*, a pour but d'augmenter le débit d'instructions exécutées sur un processeur. Dans le cas du multi-threading, un processeur physique apparaît comme deux (ou plus) processeurs logiques au système d'exploitation, qui pourra alors exécuter plusieurs applications simultanément sur ces processeurs logiques. Toutes les implantations du multi-threading nécessitent entre autres une modification du pipeline pour que chaque instruction qu'il contient soit marquée avec son contexte. La TLB est aussi marquée avec le processeur logique de façon à pouvoir différencier les contextes et prévenir les accès interdits. Ce marquage permet également de ne vider qu'une sous-partie de la TLB lors d'un changement de contexte. Une fois intégrés ces contextes au sein du matériel, il reste cependant à savoir comment gérer l'ordonnancement entre ceux-ci. En effet, seuls les contextes sont dupliqués et non les unités de calcul. Il faut donc partager le temps des unités de calcul entre les contextes.

Il existe différentes mises en œuvre de cette technique, selon les choix d'ordonnancement qui peuvent être faits entre les contextes matériels. La plus répandue est le SMT [3], rendue célèbre par les processeurs Intel sous le nom d'*HyperThreading*. Le principe est d'exécuter plusieurs contextes au sein du même cycle, en plaçant des instructions de contextes différents sur les unités de calcul disponibles lorsque cela est possible. Cela a pour effet de maximiser l'utilisation des unités de calcul.

Il existe d'autres techniques d'ordonnancement pour le multi-threading. On peut par exemple citer le multi-threading à grain fin, adopté notamment par les architectures UltraSparc (nom de code Niagara). Le principe est simple : un changement de contexte est effectué à chaque cycle, selon une politique *round robin* entre les contextes matériels. Le but étant de masquer les latences des accès mémoire, tout en faisant l'hypothèse que la plupart des instructions font appel à la mémoire.

Partant de l'optique opposée, une dernière méthode d'ordonnancement pour le multi-threading est celle dite à gros grain. Elle a notamment été adoptée par Intel sur les processeurs Itanium 2 (nom de code Montecito). Le principe est simple : le processeur exécute un contexte jusqu'à ce que celui-ci se bloque, par exemple en attente de données en mémoire. Lors d'un tel blocage, un autre contexte matériel prend la main de la même façon. L'entrelacement ainsi produit se rapproche d'un ordonnancement coopératif, dans lequel un flot d'exécution libère des ressources dès qu'il ne peut plus s'exécuter de manière non-bloquante.

3. Simultaneous Multi-Threading

1.1.4 Les limites des mono-cœur et l'arrivée des multi-cœur

Chaleur et énergie. La limite des processeurs mono-cœur se trouve dans les lois de la physique. En effet, pour augmenter la fréquence d'un processeur les constructeurs gravent des composants de plus en plus petits, de telle sorte que le courant traverse les portes logiques aussi vite que possible. Or ces composants chauffent, et leur taille de plus en plus petite concentre d'autant plus la chaleur sur la puce. Nous sommes arrivés au point où cette dissipation d'énergie peut endommager la puce et les composants si l'on continue d'augmenter les fréquences d'horloge dans ce sens.

Complexité. Un autre facteur limitant est la complexité des processeurs modernes. En effet, à chaque nouvel élément matériel introduit pour gagner en performance (étage de pipeline, prédicteur de branchement, ...), les structures de contrôle de cet élément l'accompagnent, et nécessitent très souvent bien plus de composants que l'élément lui-même. La multiplication des structures de contrôle au sein d'un cœur entraîne une explosion combinatoire de la complexité pour réaliser un processeur. Certains fabricants vont même jusqu'à annuler la sortie de certains processeurs pourtant déjà avancés devant l'étendue des tests nécessaires à garantir leur bon fonctionnement.

Bien sûr cette complexité se retrouve également au sein de chaque cœur. Cependant, il devient bien moins complexe d'ajouter des cœurs au sein d'un processeur que de rajouter d'autres éléments matériels au sein de chaque cœur.

La solution multi-cœur. Pour pallier ces difficultés, le principe est simple : si on ne peut complexifier davantage le cœur d'un processeur, il faut pouvoir en disposer plusieurs sur la même puce de silicium. C'est pourquoi la dernière décennie a vu l'arrivée des processeurs multi-cœur sur le marché grand public.

1.2 Processeurs multi-cœur - Accès uniformes à la mémoire

Cette section décrit l'organisation et les spécificités des processeurs multi-cœur constituant une solution possible face aux limites des processeurs mono-cœur. Nous nous concentrons ici sur les processeurs commercialisés à ce jour dans le domaine grand public. D'autres architectures vont voir le jour, que nous décrirons dans la section suivante. Nous présentons ici tout d'abord des concepts globaux aux architectures parallèles : la notion de cœur, ainsi que l'impact des caches au sein de ces architectures. Nous détaillons ensuite les processeurs multi-cœur à accès à la mémoire uniformes.

1.2.1 La notion de cœur

Si beaucoup peuvent considérer qu'un cœur est la même entité qu'un processeur mono-cœur, cela est loin d'être exact. En effet, les systèmes multiprocesseurs (SMP [4]), sont connus et utilisés depuis plusieurs décennies par la communauté du calcul scientifique. Ils n'ont cependant pas du tout les mêmes spécificités que les multi-cœur(CMP [5]), notamment au niveau des accès mémoire.

Un cœur correspond en réalité à un sous-ensemble d'un processeur mono-cœur. En effet, s'il contient autant de registres, d'unités de calcul et un pipeline, les caches ont quant à eux

4. Symetric MultiProcessor.
5. Chip-level MultiProcessing.

un tout autre visage. Selon les architectures, un cœur peut avoir un ou plusieurs niveaux de caches privés, mais la différence majeure est que certains niveaux de cache peuvent être partagés entre plusieurs cœurs.

1.2.2 L'impact des caches

Cette organisation en cœurs et processeurs a un fort impact sur la hiérarchie mémoire, notamment au niveau des caches. En effet, on voit clairement apparaître ici la notion de caches partagés entre plusieurs cœurs. Jusqu'ici, un cache était privé à un processeur. Nous discutons ici les différences entre caches privés et partagés, puis nous abordons les opportunités ainsi que les difficultés d'optimisation des programmes liées à ces nouvelles hiérarchies de mémoire.

Caches privés et partagés

L'impact de cette notion de partage de caches a des répercutions importantes sur les performances. Lors d'un accès mémoire, un cœur doit d'abord aller chercher la donnée dans ses propres caches privés. Si elle n'y est pas, au lieu de la chercher directement dans les caches de niveaux supérieurs, il doit avant savoir si elle n'est pas présente dans un des caches privés des autres cœurs. Cela va donc introduire des traitements supplémentaires lors d'une faute de cache.

Comme chaque cache doit représenter un sous-ensemble de la mémoire, qui est cohérente, un protocole de cohérence entre les caches est mis en place. Dans le cas d'un accès mémoire en écriture, toutes les copies de l'ancienne donnée, notamment celles dans les caches privés des cœurs voisins, sont invalidées. La donnée sera rechargée par chaque cœur de façon normale lorsqu'elle sera réutilisée.

Cohabitation de caches. Lorsque plusieurs caches de même niveau cohabitent, on peut observer un phénomène néfaste sur les performances des applications : le *faux-partage* [81]. Le faux-partage apparaît lorsqu'une ligne d'un cache contient deux variables utilisées chacune par des cœurs différents. Chaque cœur modifie sa variable, et ces modifications se répercutent dans les caches qui lui sont rattachés. Or les deux variables sont sur la même ligne de cache, et un cache ne peut opérer que sur une ligne complète. On arrive ici à une situation où la mémoire n'est plus cohérente, car cette ligne a différentes valeurs selon le cache où elle se trouve. Dans la pratique, le protocole de cohérence de cache intervient avant pour prévenir cela et garder une mémoire cohérente. Dès qu'une ligne de cache est modifiée, cette modification est propagée aux autres caches voisins, qui peuvent alors mettre à jour leur valeur, ou simplement invalider la ligne concernée. Le faux-partage affecte grandement les performances car il se traduit par des propagations successives et répétées, plusieurs cœurs étant en compétition pour obtenir la dernière version d'une même ligne de cache modifiée.

Partage de caches. Le faux-partage est un phénomène présent aussi bien sur les architectures multiprocesseur (SMP) que sur les multi-cœur (CMP). Les architectures multi-cœur ont cependant des spécificités propres, qui n'apparaissent pas ou très peu dans les multiprocesseurs. L'une d'entre elles est le partage de caches. Le partage de caches offre l'opportunité d'un partage des données entre les cœurs. Cela signifie qu'une donnée présente en cache à la demande d'un cœur devient également disponible pour utilisation depuis les autres cœurs partageant ce cache.

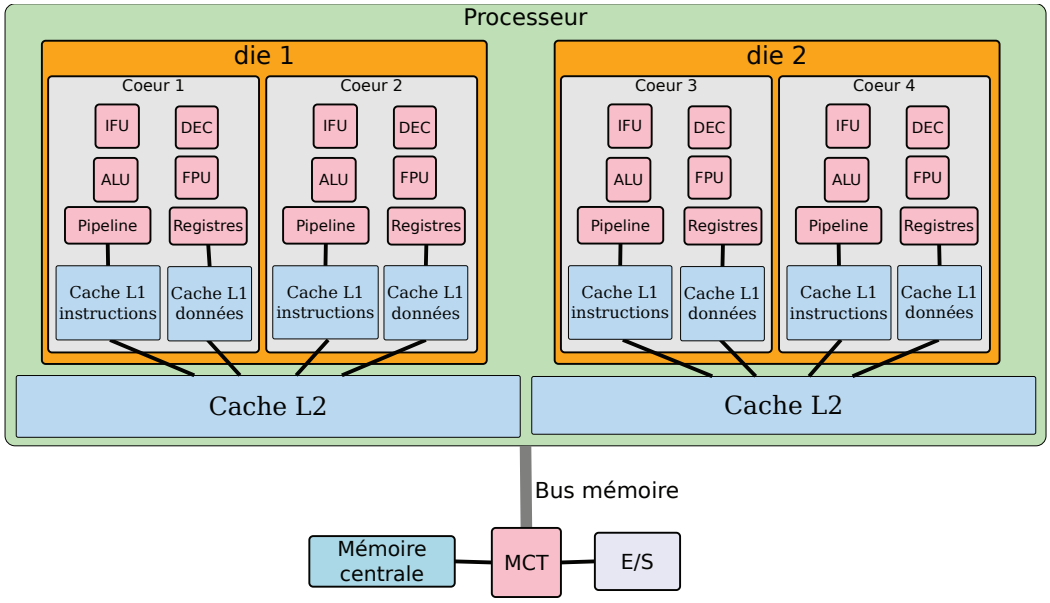

FIGURE 1.2 – Vue d'ensemble simplifiée d'une architecture multi-cœur (typique d'un Intel Core2). L'organisation des cœurs en deux dies séparées apparaît clairement.

Le partage de cache peut également avoir des effets néfastes sur les performances, dans le cas où les données demandées par tous les cœurs sont plus grandes que la taille du cache partagé. En effet, dans ce cas, un cache va évincer les données d'un autre pour charger les siennes, l'autre faisant la même chose on arrive également à une compétition pour la mémoire. Cette compétition ralentit les performances en comparaison de la même exécution sur une architecture où le cache n'est pas partagé. Comme très souvent il n'y a pas de cache partagé par tous les cœurs d'une machine, il est souhaitable de choisir les cœurs sur lesquels s'exécute l'application en fonction de la hiérarchie mémoire, pour éviter toute compétition entre les cœurs impliqués.

1.2.3 Organisation UMA : autour d'un bus mémoire

Nous présentons ici les processeurs à accès uniformes à la mémoire, ou UMA [6]. Ces processeurs sont ainsi nommés car les temps d'accès à la mémoire centrale sont les mêmes quelque soit le cœur demandant l'accès.

D'un point de vue physique, les cœurs d'un processeur peuvent être répartis sur des ensembles physiques différents, appelés *dies*. C'est notamment le cas de tous les processeurs multi-cœur basés sur la micro-architecture Intel Core2. C'est du coup le cas sur l'architecture à 8 cœurs que nous utilisons par la suite, qui est un système bi-processeur où chacun est un de ces quadri-cœurs. Cette organisation est représentée à la Figure 1.2. Sur cette architecture, on distingue les cœurs appartenant à une même *die* par le fait qu'ils partagent un cache L2.

Comme on peut le voir sur la Figure 1.2, le bus mémoire constitue le seul point d'accès de tous les cœurs vers la mémoire et les E/S. Lorsque le nombre de cœurs augmente, ce bus peut très vite devenir un goulot d'étranglement. Il est donc important d'utiliser au mieux la hiérarchie de caches, pour diminuer la pression sur le bus de données.

Les accès mémoire jouent un rôle primordial dans le monde du multi-cœur, plus important

6. Uniform Memory Access.

même qu'en mono-cœur. On a vu que la mémoire devient de plus en plus lente vis à vis de la puissance de calcul. En multi-cœur, les coûts des accès mémoires vont augmenter d'autant que le nombre de cœurs augmente, car chaque nouveau cœur envoie des requêtes vers la mémoire centrale, la surchargeant d'autant plus. La mémoire centrale devient alors le goulot d'étranglement en termes de performances des applications.

Dans les architectures multi-cœur classiques, la mémoire continue d'être vue comme une entité centralisée et cohérente. Ce point est crucial : pour que la mémoire soit cohérente, elle doit être vue et accédée de la même façon sur chaque cœur. Un processeur implante un protocole de cohérence de cache, de type MESI [22] ou MOESI [27] le plus souvent, pour assurer que le contenu des caches ne reflète jamais une mémoire incohérente. Ce protocole utilise des messages transitant sur le bus mémoire pour maintenir la cohérence des opérations de lecture/écriture.

1.3 Processeurs multi-cœur - Accès non uniformes à la mémoire

Dans cette section, nous présentons les processeurs à accès non uniformes à la mémoire, ou NUMA [7]. Les structures de cœur et de cache restent inchangées par rapport aux architectures UMA. La différence entre les processeurs UMA et NUMA se situe au niveau des communications avec la mémoire centrale. Nous détaillons tout d'abord les concepts de mémoire distribuée, puis les topologies d'interconnexion utilisées actuellement.

Organisation : autour d'une mémoire distribuée

Les processeurs UMA ont dans leur architecture un goulot d'étranglement : le bus mémoire. C'est pour répondre à ce constat que les fabricants ont mis au point les processeurs NUMA. L'idée est de distribuer la mémoire en plusieurs bancs, pour ne plus avoir un seul bus mémoire, et ainsi se débarrasser de ce goulot d'étranglement. Chaque banc mémoire est associé à un nœud NUMA. Chaque nœud NUMA peut regrouper plusieurs cœurs. Les nœuds NUMA communiquent via une interconnexion rapide.

Malgré la distribution physique de la mémoire, celle-ci apparaît tout de même de façon centralisée et cohérente au programmeur. Il est donc nécessaire que chaque cœur puisse accéder à toute la mémoire de la machine. Un cœur accède normalement au banc mémoire de son nœud. Lorsqu'en revanche il doit accéder à un nœud distant, un mécanisme de routage se met en place. En effet, la mémoire du nœud destination n'est pas nécessairement accessible en un seul saut. Les nœuds NUMA sont donc en charge de router les requêtes mémoire des différents cœurs. Chaque saut de nœud a un coût non négligeable, qui s'ajoute à la latence mémoire. C'est en cela que les accès mémoires ne sont pas uniformes : le coût d'un accès mémoire dépend du cœur demandeur, ainsi que du banc mémoire contenant la zone demandée. La topologie mémoire, c'est à dire le découpage en bancs mémoire et l'interconnexion, a donc un impact très important sur les performances.

Topologie de l'interconnexion

La mémoire contenue sur un nœud NUMA peut être accédée en un ou plusieurs sauts, selon la topologie de l'interconnexion. Cette interconnexion peut prendre différentes formes. Les

7. Non-Uniform Memory Access.

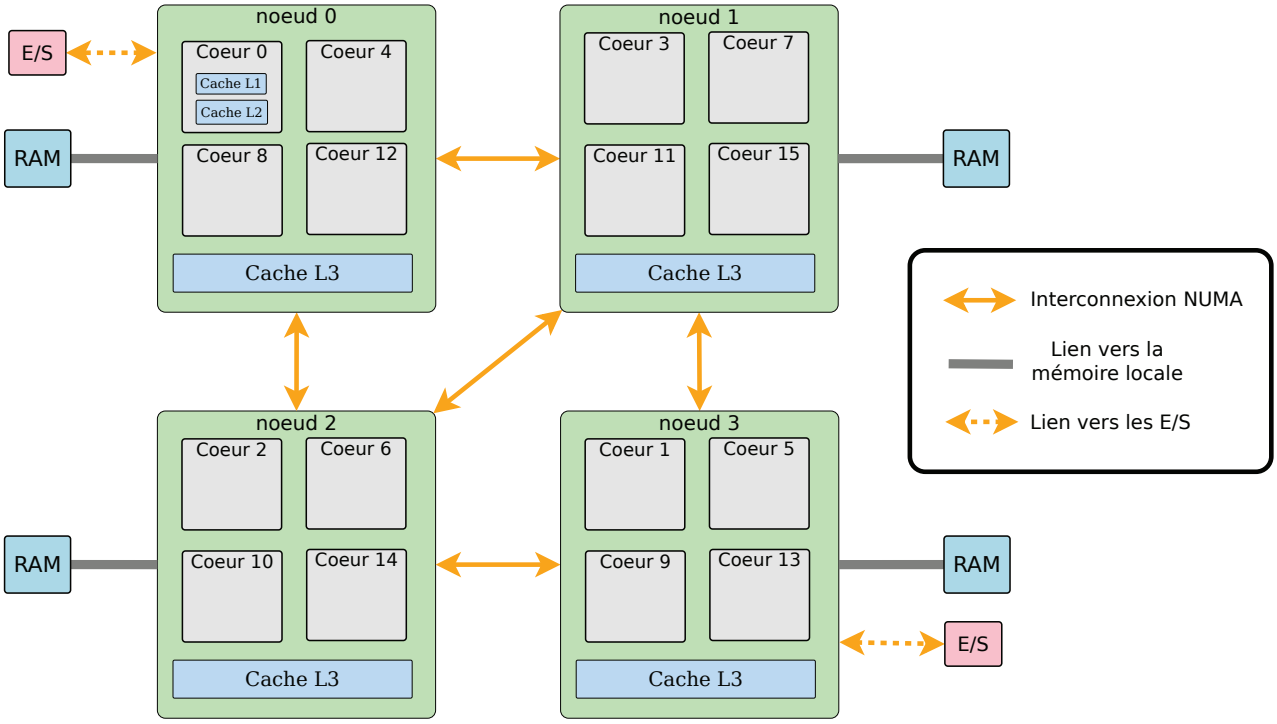

FIGURE 1.3 – Schéma d'une machine NUMA *Shanghai* d'AMD à 16 cœurs.

premières architectures NUMA d'AMD (Opteron *Barcelona*) ont été distribuées en 2007. Ont suivi les architectures *Shanghai*, *Istanbul*, et les premiers NUMA d'Intel (*Nehalem*), avec les *corei5* et *corei7*. Toutes ces architectures sont basées sur les interconnexions *HyperTransport* chez AMD et *QuickPathInterconnect* chez Intel. Ces deux interconnexions, si les détails de leurs implantations diffèrent, sont basées sur le même principe. Le principe est similaire aux réseaux, et les nœuds communiquent par paquets. Cela dit, les défis et les hypothèses de communications sont bien différentes sur puce que dans des systèmes distribués. Notamment, sur puce, les temps de communications sont bornés, et il n'y a pas de perte de paquets. Ce type d'interconnexions peut également avoir un usage autre que les connexions de nœuds NUMA. Par exemple, il peut servir à connecter les nœuds vers d'autres périphériques, comme par exemple des co-processeurs (FPGA [8], ...), ou encore implanter l'interconnexion interne de routeurs réseaux.

La Figure 1.3 montre un exemple de processeur NUMA à 16 cœurs, tiré d'une machine existante basée sur l'architecture Shanghai d'AMD. On peut y voir 4 nœuds NUMA interconnectés. Chaque nœud est responsable de son propre banc mémoire, et certains nœuds (0 et 3 sur la figure) sont également connectés vers les périphériques externes pour gérer les E/S. Chaque nœud est composé de 4 cœurs. Chaque cœur a des caches L1 et L2 privés, qui ne sont représentés que pour le cœur 0 sur la figure par souci de lisibilité. Un cache L3 est partagé entre tous les cœurs d'un même nœud. Un point remarquable sur cette architecture est que les nœuds ne sont pas complètement interconnectés. Il n'y a en effet pas de lien direct entre les nœuds 0 et 3, car ceux-ci sont reliés aux E/S. Cela implique que lorsqu'un cœur du nœud 0 doit accéder à la mémoire du nœud 3, la requête est routée par le nœud 1 ou 2 jusqu'au nœud 3. Il en va de même pour les accès aux E/S.

8. Field Programmable Gate Array (circuits logiques programmables).

Interconnexions en anneau. Depuis 2011, Intel a lancé les architectures NUMA *Sandy-Bridge* et *IvyBridge* basées sur une autre forme d'interconnexion. Cette interconnexion, plutôt que d'opter pour un graphe entre les nœuds NUMA, adopte une topologie en anneau. Ainsi, les messages émis sont transmis sur l'anneau jusqu'à leur destinataire. Chaque nœud NUMA représente un point de l'anneau. De plus, les contrôleurs d'E/S sont également des points sur l'anneau, ainsi que les différents accélérateurs présents dans le processeur (notamment accélérateurs graphiques). Ce type d'interconnexion permet d'ajouter facilement des points (nœuds NUMA ou autres) sur l'anneau, sans repenser toute la topologie.

Limitations. La première implication de la non uniformité des coûts d'accès à la mémoire est que les accès locaux sont toujours moins coûteux que les accès distants. Il est souvent difficile d'amortir les temps d'attente vers la mémoire et de les compenser : cela nécessite de grands pipelines, et complexifie grandement la logique de contrôle. C'est pourquoi il est crucial pour les développeurs de réduire les temps d'accès à la mémoire. Il convient donc pour les programmeurs de privilégier les accès locaux à la mémoire plutôt que les accès distants, plus coûteux.

Si les accès locaux sont moins coûteux que les accès distants, leur coût est toutefois bien plus important que celui des accès au cache de dernier niveau. Il est donc une bonne pratique pour les programmeurs de privilégier les accès en cache autant que possible, pour diminuer le nombre d'accès à la mémoire centrale, et ainsi amortir les coûts associés.

De plus, on retrouve dans les architectures NUMA les problèmes liés au partage et à la cohabitation des caches présents dans les architectures UMA. Ainsi, le faux-partage et la bonne utilisation des caches partagés restent des défis sur les architectures NUMA.

1.4 Le futur des processeurs multi-cœur

Dans cette section, nous présentons les conséquences des tendances actuelles sur les architectures multi-cœur à venir. Nous commençons à étudier quelques prototypes d'architectures envisagées pour accueillir plusieurs dizaines de cœurs. Puis dans un deuxième temps nous nous intéressons aux questions actuelles d'extensions des processeurs. Nous nous posons notamment la question du modèle d'accès à la mémoire, en décrivant plusieurs pistes possibles.

1.4.1 Les processeurs "many-core"

Nous l'avons vu, la tendance architecturale des nouveaux processeurs est à l'augmentation du nombre de cœurs. Après les multi-cœur présentés dans ce chapitre, les fabricants réalisent des processeurs *many-core* avec plusieurs dizaines de cœurs, voire une centaine.

Organisation : grille torique

Nous retenons les annonces sur les projets Tera-scale [21], et SCC [47] (*Single Chip Cloud*), comme référence de ce que seront les architectures many-core.

Tera-scale. Tera-scale est une architecture conçue pour expérimenter une organisation de cœurs simples en grille torique. Les cœurs sont reliés entre eux par un maillage d'interconnexions à 2 dimensions. Chaque cœur est équipé d'un contrôleur mémoire embarqué. L'originalité majeure de ce travail est d'imaginer une architecture à 3 dimensions. En effet, si la

grille de cœurs est en 2 dimensions, la troisième dimension est utilisée pour empiler les bancs mémoire directement sur les cœurs.

Chaque cœur est équipé de 5 connections, 4 vers les cœurs voisins, et une vers le banc mémoire associé. Tous les cœurs disposent également d'un routeur pour permettre les communications entre tous les éléments. L'architecture comporte un mécanisme de cohérence de cache, et offre donc une vue cohérente de la mémoire centrale. Ce mécanisme n'est pas basé sur les protocoles existants MESI et MOESI, qui écoutent l'intégralité de l'interconnexion. Il utilise un mécanisme de chemins pour garder trace des différentes copies d'une donnée.

De plus, la puce devient sensible aux pannes, au vu du grand nombre de cœurs présents sur celle-ci. Tera-scale comporte donc un mécanisme de tolérance aux fautes. Il s'agit notamment de prendre en compte la panne éventuelle d'un cœur. Dans ce cas, il faut redistribuer les calculs en attente et modifier le routage des communications.

Un prototype, nomme TeraFlops, a été réalisé à partir de cette architecture. Constitué de 80 cœurs simples, il a atteint un teraflops en consommant uniquement 62W. La grille est ici un découpage en blocs. Chaque bloc contient 8 cœurs simples. Chaque cœur a un cache L1 privé, le cache L2 est partagé par les cœurs d'un bloc. Un cache L3 est partagé par tous les cœurs. Chaque cœur gère 64Mo de RAM. Le processeur totalise ainsi plus de 5Go de mémoire centrale. Les cœurs ne sont pas généralistes et font uniquement des calculs en virgule flottante.

SCC. SCC [82] est un prototype d'architecture à 48 cœurs organisé en grille à 2 dimensions de $6 * 4$ tuiles. Chaque tuile contient 2 cœurs, une mémoire locale, et un routeur vers les 4 tuiles voisines. Les cœurs utilisés ici sont généralistes, basés sur l'architecture Pentium d'Intel. La mémoire centrale est divisée en 4 bancs externes au processeur (contrairement à Tera-scale), et n'est plus maintenue cohérente (les mécanismes de cohérence de cache ont été supprimés). Des instructions spéciales de passage de messages entre cœurs sont disponibles, et la mémoire locale d'une tuile est prévue principalement pour l'envoi de tels messages. Dans la configuration de base, chaque cœur se voit attribuer une zone privée dans la mémoire centrale. Cette configuration est susceptible de changer dynamiquement, et les cœurs peuvent au besoin demander une zone partagée en mémoire centrale. Chaque cœur peut donc partager de la mémoire avec d'autres où qu'ils soient sur la grille, soit en utilisant sa mémoire locale, soit en mémoire centrale. Cependant, la gestion de la cohérence doit être gérée manuellement en cas de partage de données.

On voit ici apparaître différentes nouvelles tendances architecturales. Tout d'abord, un mécanisme de passage de messages entre cœurs voit le jour. Cela permet des communications entre cœurs efficaces, sans utiliser de mémoire partagée (qui met en jeu des mécanismes coûteux de cohérence). Deuxièmement, et c'est très certainement le point le plus important, il n'y a plus de cohérence de cache entre des tuiles distinctes.

Ces prototypes existent surtout pour explorer de nouvelles pistes architecturales, et non pour une utilisation en production en l'état. Tera-scale examine différentes méthodes de communications inter-cœurs ainsi que la gestion fine de l'énergie. SCC étudie les principes qui peuvent permettre un bon passage à l'échelle avec le nombre de cœurs.

1.4.2 Principales tendances

Nous discutons ici les différentes tendances observées dans les prototypes many-core, ainsi que dans les nouvelles architectures du marché actuel.

Vers des mécanismes de passage de messages entre cœurs. L'apparition d'un mécanisme de passage de messages entre cœurs permet des communications asynchrones. L'avantage est que, contrairement à une communication synchrone, un cœur peut continuer ses calculs en attendant l'arrivée d'un message. Ce type de communication est extrêmement utile pour implanter un mécanisme de cohérence, par exemple. Ce type de mécanisme est notamment étudié avec le prototype SCC. Il a notamment été utilisé pour l'échange de messages entre les cœurs au sein du système d'exploitation Barrelfish [5] compatible SCC.

Vers des processeurs hétérogènes. On voit également apparaître des processeurs hétérogènes, comme le projet Fusion d'AMD [25], ou encore l'ancien projet Larrabee [71] d'Intel. L'idée générale consiste à intégrer des cœurs spécialisés à certains traitements, plutôt que de rajouter des fonctionnalités supplémentaires sur des cœurs généralistes. Le but final est de baisser la consommation globale du processeur, à fonctionnalités égales. Les fabricants imaginent également une hétérogénéité des cœurs au niveau des fréquences de fonctionnement, au sein du même processeur, toujours dans un but d'économie d'énergie.

Vers une disparition de la cohérence de cache. Le maintient de la cohérence de cache empêche un passage à l'échelle idéal des performances avec l'augmentation du nombre de cœurs. La cohérence de cache matérielle est donc remise en question, notamment avec le prototype SCC d'Intel. Il faut alors fournir des primitives spéciales pour partager la mémoire, et permettre la mise en place d'une cohérence locale, uniquement présente entre les cœurs qui partagent effectivement des données.

Si ce point se révèle être retenu, l'impact sur tout le logiciel sera considérable. En effet, chaque application parallèle écrite jusqu'à aujourd'hui fait l'hypothèse d'une mémoire cohérente. Si cette hypothèse s'avère fausse, il faudra très certainement réécrire la majeure partie des programmes actuels.

1.5 Bilan

Nous avons étudié dans ce chapitre l'évolution des architectures matérielles. Nous avons exposé les limitations des processeurs mono-cœur, et l'introduction du parallélisme. Cette introduction est progressive et prend place à plusieurs niveaux. Tout d'abord, on observe un parallélisme au niveau des instructions avec le mécanisme de pipeline. Puis une partie du processeur est dupliquée pour supporter des techniques de multi-threading (processeurs SMT). Enfin le multi-cœur apparaît, et le nombre de cœurs par processeur ne cesse d'augmenter depuis (processeurs CMP et many-core).

Toutes ces évolutions sont mises en place pour tenter de masquer les différences de performances (et notamment de fréquence) entre les processeurs et la mémoire. Cependant, ces différences de performances entre les processeurs et la mémoire continuent de s'accentuer, et les latences mémoire limitent de plus en plus les performances des applications. Pour combler cet écart, le parallélisme introduit au sein des processeurs prend de plus en plus d'ampleur. Il en résulte des défis de toujours plus importants, avec un impact grandissant sur les applications. En effet, pour tirer parti de ces nouvelles architectures, deux points sont à importants sont à considérer.

Tout d'abord, la plupart des applications classiques mettent en jeu un flot unique d'exécution. Cela rend presque impossible la pleine utilisation d'architectures multi-cœur. Le modèle classique à base de threads n'est pas satisfaisant, notamment du fait de la complexité de

la gestion de la cohérence des données entre les threads. Il faut donc repenser les modèles de programmation classiques pour permettre aux développeurs d'exprimer le parallélisme dans leurs applications. De plus, si la cohérence de cache matérielle vient à disparaître, le changement de modèle de programmation risque d'être extrêmement radical. Nous discutons différents modèles de programmation au Chapitre 2.

Il faut également mettre à niveau les supports d'exécution actuels. Notamment les systèmes d'exploitation et les environnements d'exécution utilisateurs doivent être capables d'utiliser pleinement les architectures multi-cœur. Pour cela, la répartition efficace des tâches sur les cœurs reste un des défis majeurs. Un autre défi intervient dans la gestion de la mémoire, qui doit maintenant prendre en compte des aspects de localisation physique des données pour limiter au maximum les communications inutiles. Nous discutons des impacts du multi-cœur sur les environnements d'exécution au Chapitre 3

Chapitre 2

L'impact des processeurs multi-cœur sur les modèles de programmation concurrente

Sommaire

Les multi-cœur prennent une part de plus en plus dominante sur le marché. De plus, nous avons vu au chapitre précédent que les tendances sont à l'augmentation du nombre de cœurs. Pour tirer parti de ces architectures, il faut que les applications deviennent hautement parallèles.

Dans ce chapitre, nous étudions les effets de l'émergence des multi-cœur sur les couches logicielles. Plus précisément, nous étudions d'abord comment programmer des applications parallèles, puis comment les exécuter efficacement. Dans un premier temps, nous analysons les modèles de programmation concurrente classiques, puis les améliorations qui en découlent. Nous étudions ensuite de nouveaux modèles de programmation pour les multi-cœur. Enfin, nous explorons plusieurs axes pour repenser les systèmes d'exploitation et les environnements d'exécution pour le multi-cœur.

2.1 Modèles concurrents

Les systèmes d'exploitation fournissent pour la plupart aux programmeurs une abstraction de *processus*. Cette abstraction comporte plusieurs points importants. Le flot d'instructions d'un processus est associé à un contexte d'exécution. Ce contexte est notamment constitué d'une *pile* d'exécution, où sont sauvés à chaque appel de fonction les paramètres de celle-ci, l'adresse de retour, ainsi que l'éventuelle valeur de retour. Elle contient aussi les variables locales. Une autre partie de la mémoire d'un processus, appelée *tas*, est réservée pour les allocations dynamiques du programme. On compte également dans le contexte d'un processus une *table des pages*, généralement gérée par le système d'exploitation. Celle-ci sert à configurer la mémoire virtuelle d'un processus, et assurer par ce moyen l'isolation mémoire entre les processus. Un processus comprend enfin l'ensemble des *ressources utilisées*, telles que notamment les descripteurs de fichiers ouverts. Les systèmes d'exploitations utilisent en général un ordonnancement préemptif. Or la structure de processus, et notamment le contexte associé sont nécessaires pour partager le temps processeur entre ces processus. Le système d'exploitation est également en charge d'ordonnancer les processus sur les cœurs disponibles.

Au dessus de l'abstraction de processus, le programmeur doit choisir un modèle de programmation pour écrire des applications concurrentes. Les deux modèles classiquement utilisés sont les threads et les événements, que nous présentons dans cette section. L'abstraction de processus permet de tirer naturellement parti des multi-cœur en ordonnançant plusieurs applications mono-processus sur les différents cœurs. Cependant, avec l'augmentation du nombre de cœurs, le besoin de parallélisation devient présent au sein de chaque application.

2.1.1 Threads

Le modèle à base de *threads* (ou processus légers) est l'extension naturelle des processus. En effet, le flot d'exécution d'un thread est directement apparent au programmeur. Chaque thread correspond à un flot d'exécution indépendant. Cela permet d'exprimer facilement du parallélisme : en décrivant le code associé à chaque thread d'une application. À la différence des processus, les threads partagent le même espace d'adressage. Ils peuvent donc partager des données en mémoire, ainsi que les ressources gérées par le système (par exemple des fichiers ou sockets ouverts par le processus).

Les threads, à l'instar des processus, sont très souvent ordonnancés de façon préemptive. C'est à dire qu'un quantum de temps est associé à chacun des threads placés sur un même cœur. Pour cela, chaque thread est doté d'un contexte similaire à celui d'un processus, contenant une pile, ainsi qu'une copie des registres. À la fin d'un quantum de temps, l'ordonnanceur procède à un changement de contexte. Cela consiste premièrement à sauvegarder l'état du thread sortant, à savoir sa pile, ainsi que l'état des registres. Dans un deuxième temps, l'ordonnanceur choisit le nouveau thread qui s'exécutera pendant le prochain quantum de temps. Enfin, le contexte du nouveau thread est mis en place, et le contrôle lui est rendu, jusqu'à la fin de son quantum de temps, ou jusqu'à ce qu'il décide de rendre la main explicitement, par exemple, sur une E/S, ou lors de la fin de son exécution.

Comme les threads d'un même processus partagent le même espace d'adressage, un changement de contexte entre deux threads d'un même processus est moins coûteux qu'un changement de contexte entre processus. En effet, il n'est alors plus nécessaire de vider la TLB. Le nombre de TLB résultant d'un changement de contexte est alors réduit dans ce cas.

S'il existe plusieurs interfaces de threads, la plus utilisée à ce jour est celle des *POSIX threads* (ou pthreads), implémentée notamment par la bibliothèque NPTL [30].

Limitations

Le partage de mémoire entre threads fait apparaître la notion d'accès concurrents aux données (*race conditions* en anglais). Cela soulève notamment de gros problèmes de cohérence des données.

Nous illustrons ces problèmes avec un exemple. Soit deux threads A et B, qui incrémentent tous les deux la même variable x. Il est extrêmement difficile (voire impossible) de prédire la valeur de x après incrémentation. En effet, une incrémentation est exécutée en trois étapes. Il faut tout d'abord charger la valeur de la variable depuis la mémoire. L'incrémentation se fait sur cette valeur, qui est locale au thread (dans un registre processeur). Enfin, la nouvelle valeur est écrite en mémoire. Dans notre exemple, les threads A et B exécutent ces trois étapes en parallèle. N'importe quel entrelacement est valide tant qu'il respecte l'ordre d'une incrémentation au sein d'un thread. L'ordonnancement absolu des six étapes n'est donc pas défini. Il se peut donc que le thread A charge la valeur initiale de x, l'incrémente, puis que B charge la valeur de x au même moment. Cela mène nécessairement à une erreur, car la valeur finale de x (la dernière écrite en mémoire) n'aura été incrémentée qu'une seule fois et non deux.

Ce comportement peut être évité si l'accès à une donnée partagée, dans notre exemple l'incrémentation, se fait de façon atomique. Cela peut se faire au moyen de primitives de synchronisation, telles que les verrous par exemple. L'idée est alors de prendre un verrou lors de chaque accès à une variable partagée, pour protéger ces sections critiques. Cela complexifie grandement la programmation par threads. En effet, une mauvaise utilisation de primitives de synchronisation peut mener à des inter-blocages (*deadlock*). Par exemple, c'est le cas lorsque deux verrous M1 et M2 sont en jeu, si le premier thread acquiert d'abord M1 puis tente d'acquérir M2, et qu'un deuxième thread acquiert d'abord M2 puis tente de prendre M1. Aucun des deux threads ne peut prendre le deuxième verrou, et donc les deux se retrouvent bloqués sans que l'application puisse progresser.

Outre cela, il existe également un problème d'inter-blocage où chaque thread continue de s'exécuter, sans toutefois que l'application ne progresse globalement. On parle ici de *livelock*. Les différentes situations d'inter-blocages sont extrêmement difficiles à détecter. De plus, l'ordonnancement préemptif augmente les possibilités d'entrelacements possibles, empêchant par là une anticipation de la part des programmeurs, et rendant très difficile la reproduction de certains comportements.

2.1.2 Événements

Un autre modèle de programmation concurrente existe, celui à base d'événements. L'objectif principal du modèle événementiel est de pouvoir gérer plusieurs dizaines de milliers de traitement concurrents, et répondre ainsi au problème dit C10K [50]. Le principe était alors de servir plusieurs dizaines de milliers de clients simultanés, là où il était impossible d'avoir autant de threads sur une seule machine.

Dans ce modèle, le code n'est pas organisé autour d'un flot d'exécution comme pour les threads. Il est au contraire structuré en traitants d'événements exécutés par une boucle de contrôle. La boucle de contrôle est responsable de la bonne réception des événements arrivant dans l'application. Ces événements peuvent être de nature et de provenance différentes, par exemple des E/S, ou bien des événements internes. Lors de la réception d'un événement, la boucle de contrôle est en charge d'appeler le traitant correspondant. Il est possible de faire suivre des données d'un événement à l'autre, par le biais de *continuations*. Celles-ci

sont contenues dans la structure d'événements, comme paramètres du traitant associé, et permettent de maintenir un contexte, comme le fait la pile pour les threads [1].

Pour le bon déroulement de l'exécution, certaines hypothèses sont faites sur les traitants d'événements. Tout d'abord, l'application est structurée autour d'un seul processus exécutant une boucle de contrôle. Cela implique que si un traitant vient à se bloquer, alors toute l'application se retrouve bloquée à son tour. L'hypothèse de base est donc qu'un traitant n'est jamais bloquant. Il est donc nécessaire d'utiliser exclusivement des E/S non-bloquantes dans le code des traitants.

Ne disposant pas d'E/S asynchrones, Pai *et al.*ont conçu Flash [63], un serveur Web basée sur une architecture AMPED [1]. L'idée est d'avoir un serveur Web événementiel, qui, en lieu et place d'appels à des E/S asynchrones, fait appel à des processus externes (*helpers*). Ces processus se bloquent sur les E/S, laissant ainsi la possibilité au processus principal gérant le serveur Web de continuer de s'exécuter.

On remarque que l'ordonnancement des traitants se fait de manière coopérative, au contraire du standard préemptif utilisé par les threads. Bathia *et al.* [7] utilisent un ordonnancement coopératif, travaillant de pair avec un allocateur mémoire, pour ordonnancer les traitants selon leurs besoins mémoire. Cela leur permet un contrôle fin sur l'utilisation des caches matériels. Avec l'aide d'une analyse statique, ils sont capable d'évaluer la consommation mémoire de chaque traitant. Cette évaluation demande des annotations de la part du programmeur pour les allocations dynamiques, dont la taille peut rarement être déterminée de façon statique. À partir de ces informations, leur solution est capable de maximiser l'utilisation des caches, sans toutefois entraîner les fautes de caches liées à une sur-utilisation. Le but étant de minimiser les accès à la mémoire centrale dûs à une mauvaise utilisation des caches, pour gagner en performance. Pour cela, l'ordonnanceur privilégie les traitants dont l'empreinte mémoire tiendra en cache, compte tenu l'utilisation actuelle de celui-ci. Il peut aussi favoriser l'ordonnancement de traitants qui libéreront des données en cache.

Libasync [24] et Libevent [66] proposent chacune une implantation du modèle événementiel. Ces deux implantations fournissent une boucle de contrôle générique, à l'origine pour créer des serveurs de données événementiels. Elles proposent également des événements liés aux E/S asynchrones sur les descripteurs de fichiers, à la réception de signaux ou à l'expiration d'un délai. Le programmeur peut aussi définir ses propres événements, et faire de n'importe quelle fonction un traitant.

Limitations

L'utilisation du modèle événementiel a mené à de controverses [84] du fait de la complexité des programmes engendrés. En effet, le flot d'exécution n'est pas apparent, et les programmes deviennent difficilement maintenables. De plus, cela rend les effets de bords de chaque traitant bien plus difficiles à retrouver. De plus, lorsqu'un traitant se termine, il retourne vers la boucle de contrôle. Cela implique que la succession d'appels des traitants est donc perdue. On dit que la pile d'exécution (contenant la succession des appels de fonctions) est dé-corrélée de l'exécution de l'application. On remarque notamment que l'utilisation d'outils de déboguage est grandement impactée, car ces outils se basent beaucoup sur le contenu de la pile.

L'utilisation des continuations au lieu de la traditionnelle pile mène au phénomène de *stack-ripping* [2]. Cela correspond au fait de gérer manuellement la sauvegarde et la restauration du contexte entre les traitants. L'accès aux données depuis un tel contexte pose de gros problèmes. Il est notamment difficile de déterminer la portée de la variable. La dé-allocation

1. Asymetric Multi-Process Event-Driven.

de telles variables pose donc des difficultés. La meilleure solution consiste à mettre en œuvre un mécanisme de ramasse-miettes. Cependant, ce genre de mécanisme peut se révéler très coûteux, et ainsi contribuer à faire perdre l'avantage du modèle événementiel sur le modèle à base de threads.

De plus, l'utilisation des multi-cœur n'est pas directement possible avec le modèle événementiel décrit plus tôt. En effet, ce modèle est structuré autour d'une boucle de contrôle, et donc d'un seul processus. Il existe cependant un moyen de tirer tout de même parti des multi-cœur dans certaines circonstances. Il est possible de lancer plusieurs fois le même processus, une fois sur chaque cœur, et de modifier la charge en entrée pour répartir les traitements sur l'ensemble des cœurs. Cette méthode est appelée N-COPY. Elle requiert cependant de contrôler la charge en entrée, ce qui n'est pas toujours possible. De plus, si l'utilisation de processus indépendants permet de réutiliser telle quelle une application événementielle prévue pour des mono-cœurs, elle ne permet pas de gérer facilement un état partagé entre les cœurs.

2.2 Évolutions des modèles traditionnels

Du fait des avantages et inconvénients présents dans les modèles threadé et événementiel, ces deux-là ont été sujets à une longue polémique [56, 62, 84, 79, 40, 64]. Ces débats ont menés à de nombreuses tentatives d'améliorations, que nous décrivons ici.

2.2.1 Évolutions du modèle à base de threads

Mémoires transactionnelles L'objectif des mémoires transactionnelles [32, 55] est de faciliter la gestion des accès concurrents à la mémoire. L'idée est qu'au lieu de protéger ces accès avec des primitives de synchronisation, le programmeur encapsule ces accès dans des *transactions*. Ces transactions ont sensiblement les mêmes propriétés que celles utilisées dans les bases de données. La propriété la plus importante pour les transactions qui nous intéresse ici est l'atomicité. En effet, dans la plupart des mémoires transactionnelles, le programmeur déclare un bloc de code comme étant atomique pour créer une transaction. La mémoire transactionnelle est en charge d'assurer que le bloc de code atomique s'exécute de façon atomique. Pour cela, la plupart des solutions adoptent la méthode dite optimiste. Cela consiste à créer une copie des variables du bloc atomique, le thread demandeur du bloc travaille alors sur cette copie. Si, à la fin de l'exécution du bloc, les variables partagées n'ont pas changé de valeur dans la mémoire centrale, alors la transaction procède au *commit*. C'est à dire que les nouvelles valeurs de la copie locale sont propagées vers la mémoire, et deviennent visibles depuis les autres threads. Si ce n'est pas le cas et qu'au moins une variable a changé de valeur pendant l'exécution du bloc, alors la transaction procède à un *rollback*. C'est à dire que la copie locale est abandonnée, et que le bloc atomique devra être ré-exécuté plus tard.

Une mémoire transactionnelle peut être implantée en logiciel ou en matériel. L'idée est qu'une application utilisant des mémoires transactionnelles n'utilise pas de verrous. Ainsi les seules prises de verrous se font au sein de l'implantation de la mémoire transactionnelle. De fait, ces mémoires peuvent fournir des garanties intéressantes, telle que l'absence d'inter-blocages, tout en simplifiant grandement la programmation. La machine virtuelle McRT[69] montre également un passage à l'échelle des performances plus important que celui atteint avec des verrous. Toutefois, de telles performances ne sont observées que lorsqu'il y a peu de rollbacks (donc sur des applications faisant peu d'écriture sur des données partagées).

L'utilisation des mémoires transactionnelles a cependant des limitations. En effet, si elles simplifient la gestion de la mémoire partagée, et donc la programmation par threads, elles sont en général très coûteuses. Par exemple, Larus *et al.* [55] mentionnent un ralentissement des applications allant de 2 à 7 fois. D'autres études, comme celle de Cascaval *et al.* [16] observent des ralentissements plus importants encore, allant jusqu'à 40 fois. Les mémoires transactionnelles implantées en matériel diminuent drastiquement ces coûts, mais ne sont que très peu déployées à l'heure actuelle, et donc très peu utilisées. Il existe également d'autres limitations, plus fondamentales, aux mémoires transactionnelles. En effet, certaines parties de code, comme les E/S notamment, communiquent avec l'extérieur, et ne peuvent donc pas être soumises au rollback car leurs effets sont difficiles à annuler.

Facilité de programmation OpenMP [11] est un outil destiné à faciliter le découpage d'une application en threads. Du point de vue du programmeur, il s'agit un ensemble de directives de pré-compilation destinées à faciliter la création de tâches parallèles. Cela permet notamment de décharger le programmeur des opérations de création et destruction de threads. OpenMP est donc en charge de la gestion des threads, et s'adapte à l'architecture sous-jacente pour tirer parti au mieux des cœurs disponibles.

Ce support d'exécution propose aussi des directives pour automatiser les découpages classiques. On retient notamment le *parallel for*, qui permet d'exécuter le corps de la boucle en parallèle, en traitant chaque itération avec un thread indépendant. OpenMP supporte aussi la programmation de type *fork-join*, qui permet d'associer un traitement à un thread indépendant, et de se synchroniser dessus pour s'assurer d'avoir le résultat au bon moment.

2.2.2 Évolutions du modèle événementiel

Facilité de programmation Tame [53] propose une sur-couche syntaxique à la programmation classique par événements, pour éliminer notamment les problèmes de stack-ripping. Trois nouveautés syntaxiques sont apportées. La première permet d'écrire l'application événementielle comme un code séquentiel, ce qui a pour effet de rendre le flot d'exécution à nouveau apparent. La deuxième consiste à sauvegarder automatiquement les continuations, ce qui élimine effectivement le stack-ripping. Les continuations sont alors stockées dans des objets nommés *closures*. Enfin, Tame propose également au programmeur une primitive d'attente sur un ou plusieurs événements, pour continuer l'exécution. Ceci constitue un équivalent à la primitive *join* présente dans la plupart des implantations de threads, et permet de s'assurer qu'un résultat a bien été calculé avant de progresser plus avant dans l'application.

Adaptation aux multi-cœur Nous avons vu que le modèle événementiel ne tire pas bien parti des multi-cœur. Pour remédier à ce problème, Zeldovich *et al.* [93] propose d'augmenter le modèle événementiel avec des annotations sur les événements. Ces annotations, appelées couleurs, permettent de déterminer si l'exécution de deux traitants d'événements peut se faire en parallèle ou non. Libasync-smp est l'implantation de référence d'un modèle événementiel à base de couleurs. Dans ce modèle, deux traitants de même couleur s'exécutent en exclusion mutuelle. L'implantation Libasync-smp force l'exécution de tous les traitants de même couleurs sur le même cœur, assurant ainsi l'exclusion mutuelle par couleur du modèle. Un mécanisme de vol de tâches est présent pour répartir la charge sur les cœurs disponibles. Enfin, tout événement non colorié obtient la même couleur par défaut. Ceci garantit la correction, et permet au programmeur d'injecter progressivement du parallélisme au sein des applications.

L'extension des couleurs permet donc au modèle événementiel de tirer parti des multi-cœur. Cependant, si les couleurs permettent d'exprimer l'exclusion mutuelle entre événements, elles sont insuffisantes pour exprimer des problèmes de synchronisation plus complexes. Nous prenons l'exemple d'un verrou de type *lecteurs/rédacteurs*, où plusieurs lecteurs peuvent accéder en parallèle à la donnée partagée tout en maintenant la cohérence de celle-ci. Jannotti *et al.* [48] proposent d'ajouter un deuxième niveau d'annotations, les teintes, pour régler ce problème. Ils proposent également une annotation automatique des programmes. Le but de cette automatisation est de garantir avant tout la correction de l'application, en ne coloriant que les événements ne partageant aucune donnée. L'absence d'évaluation de performances du papier laisse penser que cette solution automatique pourrait écarter des possibilités de parallélisme dont on pourrait peut-être bénéficier avec une coloration manuelle.

2.2.3 Modèles hybrides

Capriccio

La bibliothèque Capriccio [85] fournit une interface de programmation à base de threads. L'objectif cette bibliothèque est de supporter efficacement un nombre de threads très important. Pour cela, le cœur de Capriccio repose sur un moteur événementiel. Cela mène à un compromis entre les deux modèles, qui permet de gérer efficacement plusieurs points.

Tout d'abord, le fait que Capriccio soit une bibliothèque de threads utilisateurs réduit les coûts d'ordonnancement. En effet, les threads n'étant plus gérés par le noyau, il n'est plus nécessaire d'utiliser des appels systèmes pour ordonnancer les threads. Au contraire, le coût d'un changement de contexte est celui d'un appel de fonction. Il est cependant difficile pour les threads utilisateurs de tirer parti des multi-cœur. En effet, pour tirer parti de tous les cœurs disponibles, il faut au moins un thread noyau par cœur. Une bibliothèque de threads utilisateurs doit donc mettre en place une correspondance de type $M <=> N$, avec M le nombre de threads utilisateurs et N le nombre de threads noyau (au moins égal au nombre de cœurs), si elle veut pouvoir tirer parti d'architectures multi-cœur. Ce n'est pas le cas de Capriccio, qui gère tous ses threads au sein d'un seul thread noyau, et donc ne supporte pas les architectures multi-cœur.

Un problème majeur des bibliothèques de threads classiques est la consommation mémoire. En effet, à chaque thread est associé une pile, qui doit être suffisamment grande pour supporter par exemple des appels récursifs à une fonction. Lorsque le nombre de threads est grand, la quantité de mémoire allouée pour gérer les différentes piles peut excéder la quantité de mémoire centrale, ce qui pose de gros problèmes de performances. Pour éviter cela, Capriccio a mis au point une gestion fine et dynamique de la taille des piles. Par un mécanisme d'analyse statique exécuté lors de la compilation, Capriccio est capable d'évaluer la taille de pile requise pour chaque thread. Pour cela, un graphe est créé pour chaque thread à la compilation, dont les nœuds sont les fonctions. Chaque arête entre deux nœuds représente un appel de fonction. Ce graphe est ensuite décoré avec la taille de pile requise par chaque fonction. Dans le cas de fonctions récursives, des vérifications sont placées dans le code, pour réajuster la taille de la pile au besoin lors de l'exécution.

Une autre avancée de Capriccio est d'avoir un ordonnancement qui tient compte des ressources utilisées par chaque thread. En fait, Capriccio désigne sous le terme de bloc le code exécuté entre deux changements de contexte. Pour chaque bloc, la bibliothèque garde des informations sur les ressources utilisées, comme la quantité de mémoire demandée, ou encore le nombre de descripteurs de fichiers utilisés. Ainsi, si une de ces ressources est surchargée, l'ordonnanceur peut privilégier le ou les threads qui libèrent cette ressource.

Acteurs

Haller *et al.* [41] proposent un modèle de programmation à base d'acteurs unifiant les événements et les threads. Le modèle d'acteurs est une évolution du modèle d'objets. Un acteur est un objet muni d'un flot d'exécution. Il peut être composé d'un ensemble de variables et d'un ensemble de traitants de messages. Les acteurs communiquent exclusivement par passage de messages, aucune autre forme de mémoire partagée n'est permise.

Lors de la réception d'un message, un acteur peut envoyer des messages à d'autres acteurs, créer de nouveaux acteurs, ou bien manipuler ses propres données. Les acteurs sont supposés être suffisamment légers pour être présent en grand nombre dans le système, permettant ainsi beaucoup d'opportunités de parallélisme. Le modèle d'acteurs peut donc tirer aisément parti des multi-cœur.

Une des limitations de ce modèle vient du grand nombre d'acteurs présents dans une application. En effet, on retrouve des similarités avec les limitations du modèle événementiel. C'est à dire que le découpage du code est tel qu'il devient difficile de déterminer le flot naturel d'exécution d'une application.

Fibers

Adya *et al.* [2] présentent les différences entre les modèles de threads et d'événements. Pour mieux exprimer ces différences, ils définissent deux axes : la gestion des tâches, et la gestion de la pile. La gestion des tâches peut se faire de trois façons : préemptive, coopérative, ou par lots (batch). La gestion de la pile peut être manuelle ou automatique. Dans le cas d'une gestion de pile manuelle, le programmeur doit utiliser explicitement des continuations pour stocker le contexte de la tâche en cours d'exécution. Le contexte est automatiquement stocké et restauré dans la pile dans le cas d'une gestion de pile automatique.

De ces observations, Adya *et al.* concluent que le modèle threadé correspond à une gestion de pile automatique et une gestion des tâches préemptive, tandis que le modèle événementiel utilise une gestion des tâches coopérative avec une gestion de pile manuelle. Des tensions étaient présentes dans leur groupe de travail, relatives au choix de modèle de programmation pour leur projet. De fait, ils proposent un mécanisme de *fibers* pour concilier les deux styles de programmation au sein d'un même système. Les fibers sont ordonnancées de manière coopérative, au sein d'un même thread. Elles ne tirent donc pas parti de façon native des architectures multi-cœur . Les fibers peuvent être implantées avec une gestion de pile manuelle ou automatique, ce choix étant laissé au programmeur. Un mécanisme d'*adapters* est présent pour gérer le passage d'un modèle de gestion de pile à l'autre. Le programmeur est en charge de coder ces adapters, conformément au modèle de gestion de pile choisi.

Étages

Welsh *et al.* [89] introduisent la programmation par étages avec son implantation de référence, SEDA [2]. Dans ce modèle, l'application est découpée en étages, communicants entre eux de façon asynchrone par envoi de messages. Ce découpage en étages est très similaire au découpage en traitants présent dans le modèle événementiel.

Chaque étage est doté d'une file de réception de messages, ainsi que d'un ensemble de ressources. Parmi ces ressources, on trouve notamment un pool de threads pouvant exécuter le code de l'étage. Un étage s'exécute de façon concurrente, et doit donc protéger l'accès à ses variables par des primitives de synchronisation. Le programmeur utilise donc le modèle de

2. Staged Event-Driven Architecture

threads classique au sein d'un étage. De fait, un étage n'a pas nécessité d'être non-bloquant, comme c'est le cas avec les traitants d'événements. En revanche, la communication inter-étages se faisant de manière asynchrone, on retrouve le besoin d'une sauvegarde de contexte manuelle de la part du programmeur à chaque changement d'étage.

L'intérêt d'un tel modèle est que d'une part il constitue une unification intéressante des modèles de threads et d'événements, et d'autre part il permet une gestion fine et automatique des ressources associées à chaque étage. Par exemple, dans SEDA, des contrôleurs de ressources sont présents pour chaque étage, et permettent notamment de gérer le pool de threads, ainsi que la manière de récupérer les messages depuis la file d'événements. Le contrôleur de gestion de threads adapte le nombre de threads dans le pool en fonction du nombre de messages dans la file et du temps passé à attendre par chaque thread. Le contrôleur de batch gère le nombre de messages récupéré en une fois depuis la file par chaque thread, en fonction du débit d'événements sortant de l'étage. Cela permet de gérer notamment la contention sur la file.

De par la présence de multiples threads, le modèle à étages permet de tirer parti nativement des architectures multi-cœur. Cependant, les difficultés de programmation sont multiples. On retrouve en effet les problèmes de stack-ripping lors des communications inter-étages, ainsi que les problèmes de synchronisation pour gérer les accès aux variables partagées, à la fois au sein de chaque étage et aussi de façon globale.

2.3 Modèles de programmation pour multi-cœur

Nous avons vu jusque là les modèles de programmation parallèles classiques ainsi que leurs différentes évolutions. Dans cette section nous présentons des modèles de programmation conçus spécifiquement pour tirer parti au mieux des architectures multi-cœur. Ces modèles ont pour objectif de faciliter la parallélisation des applications, et de maximiser l'utilisation des cœurs disponibles.

2.3.1 Fork-Join

Le modèle *fork-join*, implanté notamment par Cilk [9], repose sur la création de tâches indépendantes. Dans ce modèle, le programmeur est invité à créer très simplement des tâches parallèles, par le biais d'une fonction *Fork* (*spawn* dans Cilk). Ces tâches seront exécutées efficacement par le support d'exécution. La fonction *Fork* accepte comme paramètre une autre fonction, qui sera exécutée en parallèle de l'appelant. Une fonction classique peut renvoyer une valeur. Bien sûr la valeur de retour d'une tâche n'est pas définie avant que cette tâche ne soit terminée. Il est donc possible de récupérer la valeur d'une tâche parallèle en attendant sa terminaison. Cela se fait par le biais d'une fonction *Join* (*sync* dans Cilk).

Dans ce modèle, il est très peu recommandé d'avoir des tâches travaillant sur des données partagées. En effet, si l'utilisation de primitives de synchronisation est possible, elle n'est pas souhaitable car cela altère les performances du support d'exécution sous-jacent. Le support d'exécution est en charge de créer le nombre de threads adéquat (selon l'application et l'architecture matérielle), ainsi que de répartir ces tâches sur les threads, pour maximiser l'utilisation des cœurs disponibles. Pour cette raison, on retrouve souvent des mécanismes de vol de tâches dans les supports d'exécution de type Fork-Join. Le découpage des données nécessaire pour rendre les tâches indépendantes est laissé à la charge du programmeur.

TBB [54] est une implantation du modèle Fork-Join par Intel. Cette implantation est conçue pour tirer efficacement parti des multi-cœur. À la différence de Cilk, TBB ne requiert

pas de compilateur spécifique, et peut être compilé avec n'importe quel compilateur C++.

2.3.2 La solution de Google

MapReduce[26] est un modèle de programmation conçu pour exploiter le parallélisme dans les grappes de calculs. Une implantation de ce modèle pour les architectures multi-cœur a été présentée par Ranger *et al.* [67]. Ce modèle permet de paralléliser aisément des programmes manipulant de grandes quantités de données et écrits suivant un schéma fonctionnel. L'utilisation du schéma de programmation fonctionnel assure l'absence d'effets de bords, et facilite donc la parallélisation des traitements.

Le modèle MapReduce propose une interface simple pour paralléliser le traitement de données, via une représentation <clé, valeur>. La première étape est le découpage des données en blocs indépendants, chaque bloc a une clé et une valeur associée. Le traitement de chaque bloc est déferré à une unité d'exécution (un nœud dans le cas d'une grappe, et un cœur sur une machine multi-cœur). Chaque cœur est responsable d'une clé, et chaque valeur associée représente une partie bien définie des données initiales. La primitive *Map* permet de lancer le calcul parallèle, en exécutant la fonction de traitement spécifiée sur chaque bloc. Cette fonction de traitement, définie par le programmeur, peut utiliser une primitive *EmitIntermediate* pour rajouter des couples <clé, valeur> au résultat de la phase de Map. La table de hachage résultante, sont ensuite traitées par la fonction de *Reduce*. L'objectif de cette phase est d'assurer l'unicité de la valeur associée à chaque clé. Cette fonction est également lancée en parallèle, mais avec une répartition différente : chaque cœur va traiter toutes les valeurs associées à une clé.

MapReduce est très efficace pour gérer de grandes quantités de données. Cela est dû principalement à deux aspects. Tout d'abord les fonctions de Map et de Reduce n'ont pas d'effets de bords incontrôlés (les seuls effets de bord utilisent EmitIntermediate), ce qui permet des implantations efficaces sans verrous. De plus, comme une grande partie des manipulations de données se font localement, l'implantation multi-cœur permet de tirer efficacement parti des caches. Ranger *et al.* montrent en effet des accélérations super-linéaires. Cependant, on notera comme limitation que le découpage des données en blocs indépendants est laissé à la charge du programmeur.

2.3.3 Files d'éléments

Click [60] est un outil créé pour réaliser des routeurs logiciels. Les routeurs logiciels ont l'avantage d'être aisément extensibles, et l'inconvénient d'être moins performants que leurs homologues matériels. L'approche choisie dans Click permet de créer des routeurs modulaires et performants. Pour cela, un langage spécifique est utilisé. L'application est structurée en *éléments*. Chaque élément est connecté aux autres via des *ports*. Cette structure n'est pas sans rappeler le modèle événementiel. Cependant, au contraire de ce modèle, les ports de Click représentent des communications synchrones. Chaque routeur est donc une chaîne d'éléments, dont les ports d'entrée et de sortie représentent les liens réseaux du routeur. Il est possible d'insérer des points de découplage dans la chaîne. La communication au niveau d'un tel point devient alors asynchrone, et le programmeur doit insérer explicitement une file de messages. Cela découpe la chaîne d'éléments en blocs synchrones.

De ce modèle sont dérivés deux projets supportant les architectures multi-cœur, SMP Click [19] et RouteBricks [28]. Le principe de SMP Click est de répartir les blocs synchrones sur les cœurs disponibles. Cette répartition est réévaluée périodiquement. Chaque paquet change donc potentiellement de cœur lorsqu'il passe d'un bloc à l'autre. SMP Click exploite

donc le parallélisme entre blocs. RouteBricks utilise une autre approche de parallélisation, et exploite le parallélisme entre paquets. Pour cela, le principe est de dupliquer les blocs sur les cœurs. Cela garantit que lorsqu'un paquet change de bloc, il restera sur le même cœur. RouteBricks a montré que le parallélisme entre paquets est plus performant pour construire des routeurs logiciels.

2.3.4 La solution de MacOS

Grand Central Dispatch (GCD) [15] permet au système MacOS X de programmer efficacement les architectures multi-cœur. L'API de GCD est disponible depuis les langages C et Objective-C. L'idée est assez similaire à celle de la programmation événementielle. L'application est découpée en blocs. Un bloc peut être vu comme un traitant d'événement, dans le sens où il est exécuté sans préemption. On note toutefois qu'un bloc peut être bloquant, à la différence d'un traitant. Pour assurer le parallélisme, les blocs sont envoyés dans des files d'exécution. GCD est en charge de gérer les threads qui vont scruter ces files pour exécuter les blocs qu'elles contiennent. À la différence de la programmation événementielle colorée, GCD expose donc les files au programmeur. Tout programme a accès à quelques files globales, chaque file représentant une priorité différente. Toutes ces files exécutent les blocs en parallèle. Pour permettre de synchroniser les calculs, une file spéciale est accessible et garantit une exécution sérialisée des blocs. Le programmeur peut également créer ses propres files sérialisées. GCD adapte le nombre de threads en fonction du nombre de files, du nombre de blocs dans chaque file, et du nombre de cœurs disponibles. Cette technologie est également disponible sur FreeBSD, sous le nom *libdispatch*.

2.4 Bilan

Nous avons décrit dans ce chapitre différents modèles de programmation concurrente. Nous avons notamment vu que les plus utilisés se basent sur les modèles de threads et d'événements. Beaucoup de travaux tentent d'améliorer différents aspects de chacun de ces modèles, que ce soit au niveau de leurs performances, de leur facilité de programmation ou encore de leur expressivité. Nous retiendrons les limitations inhérentes à chaque modèle. À savoir, pour le modèle de threads, la principale difficulté se trouve dans la gestion explicite de la mémoire partagée, et des primitives de synchronisation associées. Dans le cas du modèle événementiel, la principale limitation est la complexité du code, qui augmente significativement avec les extensions permettant de gérer les multi-cœur.

Certaines études ont montré différentes façon d'unir ces deux modèles. En effet, les modèles de threads et d'événements diffèrent par leur approche de la gestion de la pile et de l'ordonnancement. Il est donc possible d'imaginer différentes combinaisons de ces deux paramètres pour créer de nouveaux modèles de programmation parallèles. D'autres études ont menés à des modèles de programmation spécialement orientés pour la programmation d'applications parallèles sur multi-cœur. Ces études constituent les premiers pas vers une adaptation des couches logicielles pour les architectures multi-cœur. Cependant, elles sont souvent spécialisées à un type de parallélisme ou une classe d'applications précise.

Nous pensons que le modèle événementiel est un paradigme adapté au type de concurrence trouvé dans les multi-cœur. En effet, nous avons vu au chapitre précédent que les communications entre cœurs constituent souvent des difficultés au niveau matériel. Or les événements sont une bonne manière de rendre explicite ces communications au programmeur, par le biais d'un passage de messages. De ce fait, le modèle événementiel apporte plus de flexibilité avec

son approche de communications asynchrones que son homologue threadé, qui prône plutôt une vision synchrone. De plus, nous argumentons qu'il est possible de construire une exécution synchrone à partir de communications asynchrones lorsque cela pourrait être voulu, tandis que la construction inverse présente bien plus de difficultés.

La principale difficulté restante à l'heure actuelle est que le découpage des données pour un traitement parallèle s'effectue presque toujours de façon explicite. À notre sens, ce point est dorénavant crucial car, selon l'ensemble de données considérées, cela peut s'avérer être un problème plus complexe que le choix d'un modèle de programmation adapté parmi la large palette disponible.

Chapitre 3

Supports d'exécution parallèles

Sommaire

Dans le chapitre précédent, nous avons étudié différents modèles de programmation concurrente. Ces modèles facilitent la création de tâches parallèles au sein des applications. Ces tâches peuvent prendre plusieurs formes selon que le modèle de programmation choisi se base sur des threads, des événements, ou une solution intermédiaire. Dans ce chapitre, nous nous intéressons de plus près à l'exécution proprement dite de ces tâches. Cette exécution s'appuie à la fois sur des mécanismes du système d'exploitation et également sur ceux de niveau utilisateur, fournis par l'environnement d'exécution.

Nous étudions dans ce chapitre les mécanismes permettant l'exécution de tâches concurrentes. Pour cela, nous présentons tout d'abord les mécanismes de niveau noyau. Nous rappelons les différentes architectures de systèmes existants, et présentons également des systèmes pensés spécifiquement pour les machines multi-cœur. Dans un second temps, nous étudions les propriétés des mécanismes d'exécution de tâches de niveau utilisateur.

3.1 Les systèmes pour le multi-cœur

Un système d'exploitation a pour rôle de fournir aux applications une abstraction du matériel sous-jacent. Cette abstraction définit notamment la notion de processus vue au chapitre précédent, brique de base sur laquelle repose les définitions de tâches. Dans cette section, nous détaillons dans un premier temps les différentes architectures classiques de systèmes d'exploitation. Dans un second temps, nous étudions les évolutions apportées à ces architectures pour créer des systèmes spécifiques aux multi-cœur.

De façon interne, tous les systèmes sont composés d'un noyau et de bibliothèques, dont les rôles respectifs diffèrent selon l'architecture adoptée par chaque système. L'objectif principal des systèmes d'exploitation est d'abstraire les ressources matérielles. Nous nous intéressons ici surtout à l'abstraction du processeur par le système d'exploitation. Le processeur expose un

jeu d'instructions (ISA - *Instruction Set Architecture*), qui contient notamment les registres et le jeu d'instructions nécessaires au bon fonctionnement des applications. Le système propose une abstraction de plus haut niveau à partir de celle fournie par le processeur. Notamment, le système expose une API (*Application Programming Interface*), ainsi qu'une ABI (*Application Binary Interface*). L'API correspond à l'ensemble des appels systèmes, et l'ABI définit quant à elle le standard binaire des applications, qui sert notamment lors des appels de fonctions, pour déterminer les paramètres passés par registres, ainsi que la manière de gérer la pile. L'ABI correspond au format des objets binaires (bibliothèques, applications), tandis que l'API porte sur les fichiers sources.

L'API du système définit notamment l'ensemble des primitives d'E/S. Cela couvre entre autres les E/S vers le réseau, vers des périphériques de stockage, ou encore celles destinées à des périphériques multimédia.

3.1.1 Différentes architectures de systèmes

Nous détaillons ici les différentes architectures de systèmes, et plus précisément les différentes architectures de noyaux existantes.

Noyaux monolithiques

Les systèmes d'exploitation les plus utilisés à l'heure actuelle (Windows, Linux...) se basent sur des noyaux monolithiques. Dans une architecture monolithique, tous les services sont gérés au sein du noyau, en espace mémoire privilégié. Ces services comportent notamment, en plus de la gestion des pilotes de périphériques matériels, les couches logicielles menant aux abstractions de plus haut niveau. Par exemple, au dessus du pilote de carte réseau peuvent être présentes plusieurs couches du modèle *OSI (Open Systems Interconnection)*. C'est ainsi qu'on retrouve entre autres l'ensemble de la pile TCP/IP dans les noyaux courants. On note que le support des couches supérieures au sein des noyaux monolithiques peut aller jusqu'à certaines couches applicatives. C'est notamment le cas avec les systèmes de fichiers distribués, comme NFS par exemple.

Les noyaux monolithiques présentent deux limitations majeures. La première est liée à leur maintenabilité. En effet, comme tous les services sont intégrés dans le noyau, la taille de celui-ci devient de plus en plus importante au fur et à mesure des fonctionnalités disponibles. Par exemple, le noyau Linux est constitué de plus de 8 millions de lignes de code. La maintenabilité, autrement dit l'évaluation de la portée des modifications apportées par chaque développeur devient presque impossible à ce stade. La deuxième limitation des noyaux monolithiques vient du fait que tous les services partagent le même espace d'adressage. De ce fait, la faute d'un service peut affecter le fonctionnement de l'ensemble du noyau. Cela rend également le noyau plus sensible à des problèmes de sécurité.

Micro-noyaux

Les micro-noyaux sont apparus pour pallier aux limites des noyaux monolithiques. L'idée est de réduire la taille et la complexité du noyau. Pour cela, il faut redéfinir le rôle du noyau, et bien le distinguer du rôle des services. Dans une architecture à base de micro-noyau, la partie noyau est ainsi seulement en charge d'abstraire le matériel. Les services sont quant à eux exécutés chacun dans un espace mémoire à part.

Cette architecture a de multiples avantages sur celle des noyaux monolithiques. Tout d'abord, le noyau ne contient plus les services, donc sa taille est largement réduite. Cela

permet de régler les difficultés de maintenabilité du code au sein du noyau. Ensuite, les services sont implantés comme des serveurs. Comme chaque service s'exécute dans un processus séparé, la faute d'un service n'impacte pas le fonctionnement du noyau. Comme ces services tournent en espace utilisateur, les attaques vers le noyau deviennent également plus compliquées. Les communications entre les services et le noyau se font à l'aide d'*IPC (Inter-Process Communications)*.

Première génération Mach [1] fait partie de la première génération de micro-noyaux. Cette génération a eu beaucoup de difficultés à rendre le code du noyau générique, pour l'adapter à plusieurs architectures. Cette généricité nécessite de retravailler les interfaces fournies aux applications, ainsi que le code du noyau. De plus, une architecture de micro-noyau implique un grand nombre de communications (et donc d'IPC) entre services. Les performances générales du système dépendent donc en grande partie de l'efficacité des IPC.

Seconde génération Une seconde génération de micro-noyaux a vu le jour, dont L4 [57] est un bon exemple. L'idée de cette deuxième génération est d'optimiser les performances des systèmes à base de micro-noyaux. L4 est à l'origine optimisé pour les processeurs de type x86. L'idée est de tirer parti de certaines spécificités de l'architecture matérielle pour gagner en performance, notamment sur les IPC. L'observation de base est qu'une IPC nécessite au minimum un changement de contexte entre les deux contextes concernés. Là où Mach devait copier des données d'un contexte à l'autre, L4 choisit d'utiliser certains registres pour faire passer les données entre les deux contextes. Ainsi, au moment de l'IPC, l'appelant charge ses données dans les registres et le noyau effectue un changement de contexte. L'appelé peut alors directement utiliser les données de l'appelant, qui se trouvent toujours en registres. Dans L4, le coût d'une IPC est réduite au coût d'un changement de contexte, et ne nécessite pas de copie de données. L4 expose ainsi des performances significativement meilleures que celles de Mach.

Exo-noyaux

L'idée des exo-noyaux a été présentée avec *Exokernel* [31]. Il s'agit ici de minimiser les abstractions fournies par le noyau. Cette idée part du constat que les abstractions fournies par le noyau pour gérer les ressources matérielles ne sont pas toujours adaptées aux besoins des développeurs d'applications finales. De fait, les exo-noyaux laissent aux développeurs libre choix dans les abstractions qu'ils utilisent.

Pour cela, un système à base d'exo-noyau consiste en un ensemble de bibliothèques d'abstractions, et d'un exo-noyau. Dans un tel système, l'exo-noyau est en charge de multiplexer de façon sécurisée les ressources matérielles. Les ressources matérielles ainsi multiplexées sont alors exposées sans fournir d'abstractions de plus haut niveau. Une application peut donc demander au noyau des interruptions, des blocs de disques, des pages mémoires ou encore des quanta de temps processeur. Des abstractions de ces ressources sont alors construites sous forme de bibliothèques utilisateur (contrairement aux serveurs des micro-noyaux) au dessus de l'interface de l'exo-noyau.

Avec ces définitions, un système à base d'exo-noyau permet de minimiser la taille de ce noyau. Le faible niveau d'abstraction des ressources matérielles rend le code de l'exo-noyau simple, et surtout performant. Le fait de gérer les abstractions supplémentaires sous forme de bibliothèques de niveau utilisateur laisse aux programmeurs le choix des abstractions les plus adaptées, parmi un ensemble de possibilités.

3.1.2 Des noyaux pour le multi-cœur

Nous avons vu jusqu'ici différentes architectures de systèmes d'exploitation. Le but de ces systèmes est d'abstraire les ressources matérielles pour les multiplexer entre les applications disponibles. Avec l'arrivée des architectures multi-cœur, les systèmes d'exploitation doivent relever de nouveaux défis. En effet, les systèmes doivent maintenant faire cohabiter la concurrence de tâches engendrée par le multiplexage temporel du CPU avec un réel parallélisme matériel. Nous avons déjà discuté des implications du parallélisme réel au chapitre 2. Les systèmes d'exploitation doivent maintenant faire face aux accès réellement concurrents aux données, ainsi qu'à des aspects de localité des ressources allouées. Ces deux défis ont un impact majeur sur la conception des systèmes d'exploitation.

Comme nous l'avons vu au chapitre précédent, l'accès concurrent aux données peut être géré via des primitives de synchronisation. Ainsi, on peut imaginer la prise d'un verrou englobant chaque appel système, pour protéger les données internes au noyau. C'est la solution adoptée par Linux dans sa version 2.4, avec le *BKL (Big Kernel Lock)*. Cependant, cette approche a un impact très néfaste sur les performances en multi-cœur. En effet, aucun des cœurs bloqués sur le verrou n'est capable de faire de progrès. De plus, lors d'une mise à jour des données partagées le protocole de cohérence de cache doit entrer en action, réduisant aussi les performances. Il convient donc de se tourner vers d'autres conceptions. On notera dans ce sens l'effort de la communauté Linux pour se débarrasser du BKL dans les travaux de la version 2.6.

Le deuxième défi concerne l'approche utilisée pour l'allocation de ressources. Précédemment sur les processeurs mono-cœur, l'allocation du temps processeur était faite sur la base d'un partage de temps entre les applications. L'idée était de multiplexer un processeur vers plusieurs processus. La nature même de ce multiplexage change en multi-cœur, et devient un multiplexage de n cœurs vers m processus. Il convient donc d'ajouter au partage de temps un autre aspect, spatial cette fois. Ce nouvel aspect spatial doit également être pris en compte pour l'allocation mémoire, notamment dans les machines NUMA, ainsi que dans la gestion des interruptions. En effet, nous avons vu au chapitre 1 que les coûts d'accès à la mémoire varient selon la localité des données. Le fait d'allouer des données dans un banc mémoire proche du cœur qui va les traiter réduit généralement les communications et améliore les performances. De même, les interruptions liées aux périphériques peuvent être affectées à un ou plusieurs cœurs particuliers. L'*IO-APIC (I/O Advanced Programmable Interrupt Controller)* permet d'associer des lignes d'interruption avec des cœurs. Le système peut donc associer une interruption avec le cœur en charge de traiter celle-ci. Ainsi, les données liées à l'interruption sont déjà proches du cœur qui les traite, ce qui permet de tirer parti des caches, et d'améliorer les performances du traitement.

Dans cette section, nous étudions les travaux récents sur la conception de systèmes d'exploitation pour les architectures multi-cœur. Comme le nombre de cœurs augmente et que les communications entre eux sont coûteuses, ces travaux s'inspirent beaucoup de techniques venant des systèmes distribués.

K42 K42 [90] est l'évolution du système d'exploitation Tornado [35]. K42 se base sur un modèle objet pour être modulaire et adaptable. C'est un système basé sur une architecture à micro-noyau, avec des serveurs en mode utilisateur. L'objectif de K42 est d'être reconfigurable à chaud, et donc hautement adaptable, tout en offrant un passage à l'échelle des performances en multi-cœur. K42 fait également le choix d'exposer une ABI et une API compatibles avec celles de Linux, ce qui permet une portabilité des applications existantes.

K42 est architecturé autour d'un micro-noyau. Ce noyau fournit un gestionnaire de mémoire, de processus, une pile réseau, la gestion des périphériques, et surtout un mécanisme d'IPC efficace. Les IPC de K42 sont appelées des *PPC (Protected Procedure Call)*. Elles existent sous forme synchrone et asynchrone. La forme synchrone tire au maximum parti de l'architecture sous-jacente, et utilise les registres pour passer les données lorsque cela est possible. Cela implique que les deux partis concernés par l'IPC synchrone s'exécutent sur le même cœur. L'IPC asynchrone a la forme d'un appel de méthode dans un autre espace d'adressage, et peut être appelée depuis un cœur différent.

Au dessus de ce noyau des services sont fournis par le biais de serveurs. Chaque serveur est représenté par un objet. Dans le modèle objet, chaque service est représenté par une interface bien définie. Un objet qui implante un tel service hérite de son interface. Il est donc possible de remplacer une implantation par une autre, tant qu'elles sont compatibles (et héritent de la même interface). Cela permet notamment à K42 de changer sa politique de traçage dynamiquement, selon les observations faites. En effet, un traçage a gros grain, impactant peu les performances, est présent en permanence. Lorsqu'une anomalie est détectée, le service en question active alors un traçage à grain plus fin, pour préciser la nature de l'anomalie. Ce mécanisme d'adaptation dynamique peut servir à mettre en œuvre des aspects orthogonaux aux fonctionnalités des services de base, comme par exemple des mécanismes de tolérance aux pannes.

Pour permettre un bon passage à l'échelle avec le nombre de cœurs, K42 introduit le concept d'objets fragmentés (*clustered objects*). L'idée est de gérer un objet de façon distribuée, avec par exemple un fragment de l'objet par cœur, et cela de façon transparente. Le programmeur fournit tout d'abord une interface classique de haut niveau. Puis l'implantation de l'objet fragmenté se fait de façon distribuée.

Corey Boyd *et al.* [12] fait le constat que les abstractions fournies par les systèmes d'exploitation classiques ne permettent pas d'afficher de bonnes performances en multi-cœur. Leur hypothèse est que, pour passer à l'échelle, les applications doivent contrôler le partage de données. Or cela n'est pas possible si le système d'exploitation a lui-même besoin de partager ses propres données entre les cœurs. De plus, l'abstraction classique de thread implique un partage automatique de certaines ressources. Ils proposent donc Corey, un système qui garantit l'absence de partage de données tant que celui-ci n'est pas spécifié explicitement par l'application. Pour mettre en œuvre un partage de données explicite au sein des applications, Corey propose trois abstractions, décrites ci-dessous.

Les threads déclarent explicitement pour chaque donnée une *plage d'adresses (Address range)* dans laquelle elle sera placée. Ces plages peuvent être partagées entre les threads d'une application ou privées. La mise en œuvre des plages d'adresses se fait via une gestion des tables de pages à deux niveaux. Le premier niveau correspond à la table des pages pour les plages d'adresses partagées, maintenant l'association entre les adresses virtuelles et physiques. Le niveau supérieur contient les tables des pages pour les plages d'adresses privées. Au niveau supérieur, il y a donc une table des pages par thread. Cette conception à deux niveaux permet d'éviter le partage des tables privées, et donc la contention sur ces accès.

Une application peut également demander au noyau de dédier un cœur à certains traitements (*Kernel cores*). L'idée ici est de permettre de réduire la contention, et de tirer pleinement parti des caches en localisant des traitements noyaux sur un cœur donné. Cela évite le partage de données au sein des pilotes matériels. Par exemple, il est possible de dédier un cœur à la gestion du réseau.

Corey permet également de créer des *zones de partage (shares)*. L'idée est ici similaire

aux plages d'adresses, mais appliquée cette fois aux descripteurs de fichiers. Le programmeur déclare donc des descripteurs de fichiers privés à un thread, ou bien partagés. À nouveau, cela nécessite une gestion à deux niveaux de la table des descripteurs de fichiers. Cela permet au noyau de localiser les traitements sur un descripteur de fichier privé sur un seul cœur, dans le cas où les traitements se font au sein du même thread que celui qui a créé le descripteur.

McRT *McRT* [69] *(Many-Core RunTime)* est une machine virtuelle spécifiquement conçue pour passer à l'échelle sur les futurs processeurs many-core. Cette machine virtuelle peut s'exécuter en lieu et place du système d'exploitation (mode *bare-metal*), ou bien au dessus d'un système hôte. McRT est axé autour de trois composants majeurs : un allocateur mémoire, une abstraction de thread utilisateurs avec leur ordonnanceur, et un mécanisme de mémoire transactionnelle. Les concepteurs de McRT mettent en avant la synergie entre ces différents composants.

L'ordonnanceur de McRT utilise un ordonnancement coopératif. Cela permet notamment la synergie avec les autres composants. Le modèle de threads choisi se base sur le modèle Fork-Join, vu au chapitre 2. L'abstraction de threads fournie par McRT peut être adaptée pour fournir des interfaces de programmation classiques. On trouve ainsi des adaptations pour Pthreads et OpenMP, notamment.

La mémoire transactionnelle de McRT permet de simplifier la programmation, en supprimant la notion de verrous de la vue du programmeur. Les auteurs montrent qu'à une même granularité de synchronisation, la mémoire transactionnelle passe à l'échelle avec le nombre de cœurs, mais pas l'implantation à base de verrous. Au contraire des mémoires transactionnelles actuelles qui utilisent pour la plupart des primitives non-bloquantes, celle de McRT est bloquante. En effet, elle est fortement couplée à l'ordonnanceur, et utilise ses propriétés d'ordonnancement coopératif. Cela permet de gérer de façon simple les abortions de transactions en présence de plusieurs threads.

L'allocateur mémoire de McRT est fortement couplé à la mémoire transactionnelle. En effet, si des allocations mémoires sont faites au sein d'une transaction, il faut les dé-allouer si cette transaction échoue (rollback). De plus, la libération effective d'une zone mémoire ne doit être faite que si aucune référence vers cette zone n'est accessible depuis une autre transaction. C'est une autre source de couplage entre l'allocateur et la mémoire transactionnelle de McRT.

Barrelfish Là où les concepteurs de systèmes d'exploitation pour multi-cœur prônent l'utilisation de verrous à grain fin ou de partitionnement entre cœurs, les concepteurs de Barrelfish [5] adoptent un point de vue radicalement différent. En effet, ils considèrent une machine multi-cœur comme un système distribué, où chaque cœur représente un nœud du système. Le noyau est donc distribué sur chacun des cœurs. La vue d'une mémoire centrale cohérente et partagée entre les cœurs est ainsi délaissée, au profit de mécanismes efficaces de passage de messages. De ce fait, Barrelfish reprend beaucoup d'algorithmes du domaine des systèmes distribués. Barrelfish est architecturé autour d'un exo-noyau.

L'exo-noyau s'exécute sur chaque cœur, et est appelé *CPU driver*. En tant qu'exo-noyau, son objectif n'est pas de fournir des abstractions de plus haut niveau. En revanche, un CPU driver fournit un mécanisme de communication efficace entre les cœurs. Cette communication est basée sur de l'envoi de message, et s'effectue de manière asynchrone. Comme le noyau est lié à une architecture matérielle spécifique, il est possible de tirer parti de cette architecture pour augmenter les performances des primitives de gestion des messages. Par exemple, sur les architectures multi-cœur actuelles, un passage de message s'effectue via mémoire partagée.

On note que cette abstraction permet d'utiliser de façon transparente des architectures hétérogènes, même si certains points restent peu clairs. Par exemple, il n'est pas précisé comment peut s'effectuer la compilation d'une application pour plusieurs architectures de processeurs.

Au dessus de ce noyau se trouvent les moniteurs, qui implantent les abstractions du système. Ces moniteurs sont répliqués sur les cœurs à la manière des systèmes distribués. De même, ils maintiennent un état cohérent via des consensus, atteints grâce au mécanisme de passage de messages du noyau. Barrelfish utilise notamment des moniteurs pour gérer l'allocation mémoire, ou encore l'ordonnancement.

Les flots d'exécution des applications sont distribuées sur les cœurs. Pour gérer les communications, il y a deux possibilités. Une application peut utiliser explicitement le mécanisme de messages du noyau, et ainsi adopter la philosophie distribuée de Barrelfish. Il est également possible d'exécuter des applications classiques, reposant sur une mémoire globale cohérente. Si l'architecture matérielle ne fournit pas de mémoire globale cohérente, alors le mécanisme de cohérence de cache logicielle présent dans Barrelfish entre en jeu. Ce mécanisme utilise les messages pour fournir une abstraction de mémoire partagée cohérente.

3.2 L'exécution de tâches en multi-cœur

Nous avons vu jusqu'ici les différentes architectures possibles de systèmes d'exploitation, ainsi que l'impact des multi-cœur sur la conception de tels systèmes. Le rôle du système d'exploitation est d'allouer du temps processeur aux différentes applications. Selon l'architecture des systèmes, cette allocation peut se faire au sein du noyau ou bien en espace utilisateur. Lorsqu'un ordonnanceur est présent en espace utilisateur, il demande des ressources au noyau. Il existe des mécanismes pour que le noyau notifie l'ordonnanceur lorsque les disponibilités de ces ressources changent [3].

L'ordonnancement de tâches en multi-cœur se fait à la fois dans une dimension temporelle et spatiale. La dimension temporelle correspond au partage de temps entre les tâches au sein d'un même cœur (*time-sharing)*. Cette dimension a été étudiée depuis les premiers systèmes multiprogrammés. La dimension spatiale correspond quant à elle à la répartition des tâches sur les cœurs disponibles. Cette dimension prend un grand intérêt avec l'arrivée des processeurs multi-cœur et de leurs spécificités, comme notamment la hiérarchie mémoire. Nous nous intéressons ici uniquement aux spécificités de l'ordonnancement en multi-cœur. De ce fait, nous laissons volontairement des aspects liés à la dimension temporelle. Nous n'abordons pas non plus les problèmes de famine ou d'équité. En revanche, nous nous concentrons sur la dimension spatiale de l'ordonnancement, liée aux multi-cœur. Le principal défi à relever ici est de déterminer quel cœur affecter à chaque tâche.

Dans cette section, nous étudions les deux aspects à prendre en compte pour l'affectation d'un cœur à une tâche. Le premier aspect est de répartir les tâches, de façon à maximiser le parallélisme effectif. Dans un deuxième temps, il est également nécessaire de tenir compte des communications engendrées par la répartition des tâches. Comme nous l'avons vu au chapitre 1, les communications sont coûteuses. L'objectif est donc de minimiser ces communications pour maximiser les performances.

3.2.1 Répartir les tâches

Pour tirer parti au maximum des architectures multi-cœur, il faut éviter au maximum de laisser des cœurs inactifs. La condition minimum pour cela est d'avoir au moins autant de tâches que de cœurs. Le chapitre précédent montre comment découper une application en

tâches à ordonnancer. Il est donc de la responsabilité du programmeur de fournir suffisamment de tâches à l'ordonnanceur pour tirer parti des multi-cœur. Nous étudions ici différentes techniques d'ordonnancement, plus précisément comment choisir sur quel cœur ordonnancer une tâche. Dans la suite de ce chapitre nous faisons l'hypothèse qu'il y a plus de tâches à ordonnancer que de cœurs. L'objectif est ici d'équilibrer la charge, pour maximiser l'occupation des cœurs autant que possible. Les tâches à exécuter sont généralement stockées dans des files d'attente. On dégage deux architectures possibles pour la gestion de ces files.

Une première façon de faire est d'utiliser une file globale. Toutes les tâches en attente sont insérées dans cette structure, et tous les cœurs scrutent cette file pour récupérer une tâche à exécuter. C'est l'architecture adoptée dans Linux 2.4. Cependant, pour maintenir la cohérence de la file d'attente, il est nécessaire d'en protéger les accès par des verrous. Cette prise de verrou de la part de chaque cœur entraîne une contention qui empêche le passage à l'échelle de cette solution. Une deuxième architecture est envisageable pour permettre un meilleur passage à l'échelle avec le nombre de cœurs. Il s'agit ici de limiter les prises de verrous. L'idée est d'avoir une file d'attente par cœur. Chaque cœur scrute ainsi sa propre file. Le défi revient en revanche à dispatcher les tâches sur chaque file. Cette solution avec une file par cœur est adoptée dans Linux 2.6 ainsi que dans la plupart des ordonnanceurs récents.

Comme dit plus haut, le défi est de répartir équitablement les tâches sur les files d'attente. En effet, des tâches différentes n'ont pas le même temps d'exécution. Cela tend naturellement vers un déséquilibre des files d'attente, et ajoute à la complexité de la répartition équitable du temps d'exécution sur les cœurs. Faute de savoir prévoir le temps d'exécution des tâches présentes dans le système, il faut trouver une solution pour rééquilibrer la charge sur les cœurs de façon dynamique. Il existe plusieurs méthodes pour répartir les tâches sur les files d'attente. Click-SMP [19], présenté en section 2.3, opte pour un gestionnaire chargé de rééquilibrer périodiquement les files d'attente. Ce gestionnaire doit pouvoir évaluer de façon spontanée la charge présente sur chaque cœur. Lorsque les cœurs surchargés et sous-chargés sont déterminés, ce gestionnaire rééquilibre la charge. Ce rééquilibrage consiste à faire migrer les tâches d'un cœur surchargé vers un cœur sous-chargé.

Vol de tâches. Une autre technique de rééquilibrage de charge est de procéder à du vol de tâches. Le principe du vol de tâches à été décrit par les créateurs de Cilk [8, 9, 10]. Cilk se base sur le modèle Fork-Join décrit en section 2.3. Le principe du vol de tâches n'est pas limité à ce modèle, et se retrouve entre autres dans Linux 2.6 et Libasync-smp [93]. L'avantage du vol de tâches par rapport à la solution de Click-SMP réside dans la décentralisation complète du mécanisme de rééquilibrage de charge. Le principe du vol de tâches peut s'exprimer ainsi : lorsqu'un cœur n'a plus de tâches à exécuter dans sa propre file, il vole des tâches présentes dans les files d'autres cœurs. On note que cela réintroduit des accès concurrents aux différentes files d'attente, et donc la prise de verrous à chaque accès à ces files. Une structure spéciale, la *DEqueue* [18] *(Double-Ended queue, ou file à deux entrées)*, permet de maintenir la cohérence au sein de la file sans prise de verrou chez la victime du vol. Cette structure a une taille dynamique, et permet trois opérations. Les deux premières opérations possibles sont les opérations classiques sur les files : ajouter une tâche en queue, et récupérer une tâche en tête. La troisième opération possible est le vol d'une tâche en queue. Ainsi, un seul thread accède la tête d'une DEqueue, et récupérer une tâche peut donc se faire sans prise de verrou. Cela permet de ne pénaliser que le cœur voleur de tâche, au profit d'une exécution rapide (*fastpath*) pour le déroulement "classique" d'un programme (sans vol). Cette structure, aussi intéressante soit-elle, repose sur des hypothèses fortes. Tout d'abord,

toutes les tâches considérées doivent être indépendantes ou protéger explicitement les accès à des données partagées. De plus, les tâches ne peuvent pas être produites (insérées dans la DEqueue) par d'autres cœurs que le propriétaire de la DEqueue. Ces hypothèses sont valides entre autres dans le modèle Fork-Join. Cependant, elles ne tiennent pas dans le modèle de Libasync-smp par exemple.

Migration de tâches. La migration d'une tâche d'un cœur vers un autre n'est pas sans coût. Que cette migration soit le fait d'un gestionnaire centralisé ou d'un mécanisme de vol de tâches, elle nécessite la migration des données nécessaires à l'exécution de la tâche. C'est à dire que les données manipulées par la tâche volée devront transiter jusqu'au cœur destination. Le coût de ce transit doit être pris en compte. Une des explications du coût d'une migration de tâche se trouve au niveau des caches. En effet, il est probable que les données demandées par une tâche soient déjà en cache lors de son exécution. Si cette tâche est volée, alors elle doit recharger ces données dans un nouveau cache. Ce coût est réduit si le cœur voleur et le cœur victime partagent un cache. Les coûts d'une migration ne se limitent cependant pas aux caches. Le coût d'implantation du vol peut devenir important selon la granularité des tâches considérées : le coût du processus de vol en lui-même peut parfois être comparable au coût d'exécution de la tâche considérée. Gaud *et al.* [37] propose une amélioration du système de vol de tâches de Libasync-smp. Il réduit les coûts du vol de tâches original de trois façons. Premièrement, en tenant compte du temps nécessaire pour effectuer un vol de tâche, et du temps d'exécution des tâches dans le système (estimées par le biais d'annotations), il est capable de ne déclencher le mécanisme de vol que lorsqu'il est rentable. Deuxièmement, il privilégie les vols vers des cœurs victimes partageant un cache avec le cœur voleur. Enfin, il vole plusieurs tâches indépendantes en une fois, pour amortir le coût d'un vol, et diminuer globalement la fréquence des vols. Strong *et al.* [78] étudie des techniques efficaces de migration de thread. Il propose entre autres de minimiser le nombre de changements de contexte lors d'une migration. Il observe également que le réveil d'un cœur inactif est coûteux. Il propose ainsi d'utiliser d'un mécanisme d'attente active au lieu d'interruptions pour réveiller les cœurs inactifs.

3.2.2 Réduire les communications

Nous avons vu comment maximiser l'utilisation des cœurs. Cela peut se faire de façon centralisée via un gestionnaire qui rééquilibre périodiquement la charge, ou bien de façon distribuée via vol de tâches. Cependant, en plus de cela, il est nécessaire de tenir compte des interactions entre les tâches ordonnancées. L'objectif est ici de réduire les communications entre les cœurs, pour minimiser la charge sur le principal goulot d'étranglement matériel (cf. Chapitre 1) : l'interconnexion entre les cœurs.

En effet, le partage de cache matériel entre deux cœurs peut influer grandement les performances. Deux principaux facteurs jouent ici. Premièrement, si les tâches présentes sur les cœurs communiquent entre elles, alors le partage d'un cache sera souvent bénéfique, car cela permet notamment de décharger l'interconnexion. En revanche, si deux tâches demandant beaucoup de données sont ordonnancées sur des cœurs partageant un même cache, elles entrent en compétition. Cette compétition est à éviter, car chaque tâche va évincer des données de l'autre du cache partagé pour y placer les siennes. Cela augmente considérablement le nombre d'accès à la mémoire, la charge sur l'interconnexion, et diminue les performances. Pour éviter ces problèmes, plusieurs études ont été menées pour déterminer des stratégies de placements efficaces des tâches sur les cœurs. La principale difficulté est ici de détecter la

présence de communications ou de compétition entre les tâches, autrement dit détecter les affinités entre tâches.

Tam *et al.* [80] proposent d'utiliser les informations fournies par le processeur pour détecter les affinités entre les threads. L'idée est ici d'éviter de demander au programmeur d'annoter ses applications, mais plutôt de concevoir un support d'exécution qui détecte les affinités en ligne. Il est en effet très difficile pour le programmeur, voire impossible, de garder la trace de toutes les communications entre les tâches, surtout quand des bibliothèques tierces sont utilisées. La détection se fait via les compteurs de performances remontés par le processeur. La solution de Tam *et al.* fonctionne en quatre étapes. Tout d'abord, ils évaluent l'impact sur les performances d'éventuels accès aux caches distants. Cet impact est évalué via une décomposition des cycles passés à attendre. S'il est significatif, alors l'étape suivante est activée. Dans cette étape, ils détectent les zones de partage mémoire entre les threads. Cela est possible via une spécificité des compteurs de l'architecture *Power*. Ces compteurs permettent en effet de tracer l'adresse de la donnée qui a causé une faute de cache L1. Pour approximer le partage de données entre les threads, il est possible de coupler cette information avec le nombre de fautes de cache entraînant un accès distant. À partir de cela, ont peut alors construire pour chaque thread une table des zones partagées (*shMap*) qui provoquent des accès distants. La troisième étape se base sur ces cartes pour classer les threads, et regrouper ceux qui partagent les données d'une shMap. Enfin, la dernière étape est de migrer les threads sur les cœurs de façon à ce que ceux d'un même groupe soient le plus proches possible. Pour réduire les coûts d'observation des compteurs, Tam *et al.* utilisent un échantillonnage à deux niveaux. Tout d'abord, ils n'observent qu'une partie de la mémoire virtuelle. Enfin, ils ne considèrent qu'un échantillon tous les N accès.

Ordonnancement de processus. D'autres études se sont penchées sur l'ordonnancement de processus sur les architectures multi-cœur. Ce problème est similaire à celui de l'ordonnancement des threads dans le principe. Cependant, beaucoup d'études font des hypothèses différentes qu'il est bon de rappeler. Tout d'abord, les processus fonctionnent dans des espaces d'adressage différents, et ne partagent donc pas de mémoire. Ensuite, les études que nous citons ici font également les hypothèses suivantes. Premièrement, il y a exactement un (et un seul) processus par cœur (et un thread par processus). Enfin, un seul cache est partagé entre les cœurs : le cache de dernier niveau (aussi appelé LLC[1]).

Chandra *et al.* [17] proposent de prédire l'impact de l'ordonnancement de deux processus sur deux cœurs partageant un cache. Ils présentent pour cela trois modèles possibles. Deux modèles à base d'heuristiques : *FOA (Frequency Of Access), SDC (Stack Distance Competition)*, et un modèle analytique, basé sur des probabilités. Ces modèles s'appuient notamment sur deux métriques. Premièrement, l'associativité du cache partagé joue un grand rôle. L'autre métrique considérée est la *stack distance*, qui correspond à l'écart entre deux accès à une même donnée. Le modèle le plus précis est l'analytique. Il est cependant compliqué à mettre en œuvre, et une implantation en ligne de ce modèle au sein d'un ordonnanceur est difficile.

Jiang *et al.* [49] présentent un algorithme capable de déterminer l'ordonnancement optimal des processus, au sens du placement optimal des processus sur les cœurs. Cet algorithme crée un graphe des tâches, dont chaque arête contient le coût de colocalisation des tâches si elles étaient ordonnancées de façon à partager un cache. Cela requiert donc de connaître à l'avance le coût du placement de deux processus sur deux cœurs partageant un cache. L'algorithme

1. Last-Level Cache.

du placement idéal est polynomial si le LLC est partagé par deux cœurs. Dans le cas où plus de deux cœurs partagent le LLC, alors l'algorithme devient *NP-complet*. Dans ce cas, il est toutefois possible d'approximer la solution optimale de façon satisfaisante en temps polynomial.

Knauerhase *et al.* [51] et Zhuralev *et al.* [94] proposent de mettre en œuvre des mécanismes d'ordonnancement basés sur des approximations de l'algorithme de Jiang *et al.*. Pour évaluer les coûts de colocalisation, ils observent les fautes de cache de dernier niveau lorsque l'application est seule (pas de compétition). Cela donne un aperçu de la demande en cache d'une application (*working set*). L'idée est d'assurer que les processus avec les plus grosses demandent ne partagent pas de cache. Cette méthode permet une mise en œuvre en ligne au sein d'un ordonnanceur.

architectures hétérogènes Le cas des architectures hétérogènes rajoute des contraintes sur l'ordonnancement des threads. En effet, sur de telles architectures, chaque unité d'exécution n'a pas les mêmes fonctionnalités.

Knauerhase *et al.* [51] présentent un ordonnancement basé sur l'apprentissage. Chaque thread met à jour ses propres contraintes. Plus précisément, lorsqu'un thread demande une fonctionnalité non accessible sur le cœur courant, il lève une exception. Un thread qui lève une exception est migré sur un autre cœur. Lorsqu'un thread génère plusieurs fois une exception pour un même cœur, il est marqué incompatible et ne sera plus ordonnancé sur ce cœur.

HASS [72] est un ordonnanceur pour architectures dites asymétriques. Les cœurs de ces architectures ont les mêmes fonctionnalités, mais tournent à des fréquences différentes. Le système observe chaque thread seul lors d'une première exécution. Il en déduit un profil qui résume les besoins du thread. Avec la connaissance du matériel et du profil de chaque thread, HASS ordonnance les threads périodiquement de façon à maximiser les performances. Un choix typique lorsqu'on rencontre un thread dont les performances sont limitées par les accès mémoire, est de l'ordonnancer sur un cœur de faible fréquence. En effet, choisir un cœur plus rapide n'aurait que peu d'intérêt car le thread passerait alors la majeure partie de son temps à attendre la mémoire.

3.3 Bilan

Dans ce chapitre, nous avons vu l'impact des architectures multi-cœur sur les supports d'exécution. Cet impact est présent à différents niveaux. Nous avons étudié les supports d'exécution depuis les systèmes d'exploitation jusqu'aux environnements utilisateurs.

Le partage de mémoire est rendu systématiquement explicite au programmeur dans Corey et Barrelfish. Les architectures multi-cœur sont de plus en plus considérées comme des systèmes distribués, comme on le voit notamment dans Barrelfish et K42.

Nous avons ensuite traité des évolutions des supports d'exécutions. La principale évolution liée aux multi-cœur concerne l'ordonnanceur. En effet, il devient nécessaire pour l'ordonnanceur de gérer une nouvelle dimension, spatiale, de répartition des tâches sur les différents cœurs. Le premier objectif de cette répartition est d'occuper tous les cœurs. Il faut donc répartir la charge équitablement. Toutefois, pour atteindre de bonnes performances, il faut également prendre en compte les communications entre les différentes tâches, ainsi qu'avec la mémoire. Ces communications créent en effet des affinités entre certaines tâches, et des compétition entre d'autres. Cela est directement lié à la charge exercée sur les composants mémoire, comme les caches ou l'interconnexion entre les cœurs et la mémoire. Il est très

difficile de prendre en compte en même temps les critères de répartition de charge, d'affinités et de compétition.

À notre connaissance, malgré les efforts pour unifier ces trois critères au sein d'un même support d'exécution, aucun ordonnanceur ne tient pleinement compte de ces trois critères. On citera tout de même Stingy [7], McRT [69] et TBB [54] dont l'ordonnanceur et l'allocateur mémoire fonctionnent de pair pour offrir de meilleures performances.

Chapitre 4

Présentation générale de la contribution

Sommaire

Nous avons présenté l'état de l'art dans les chapitres précédents. Nous y avons tout d'abord montré que la technologie multi-cœur permet aux processeurs de gagner en performance, et devient largement adoptée. Cette approche n'est toutefois pas sans impact sur les couches logicielles. Cet impact porte à la fois sur les modèles de programmation et les supports d'exécution. La première partie de notre contribution explore différentes pistes au sein d'un support d'exécution, lié à un modèle de programmation donné. Dans la seconde partie de la contribution, nous étudions les performances d'une application en changeant le modèle de programmation. Dans ce chapitre, nous présentons le contexte et l'approche de nos contributions.

4.1 Motivations

De nombreuses études de performances existent en multi-cœur. Ces études portent sur des applications tirées de différents domaines. On peut citer parmi ces domaines les applications liées au calcul scientifique (HPC [1]), aux traitements vidéo, et plus récemment les serveurs de données. Les serveurs de données ont été beaucoup étudiés en mono-cœur, mais relativement peu en contexte multi-cœur. Nous appelons serveur de données toute application qui reçoit des requêtes de clients par l'intermédiaire d'un réseau, les traite, puis renvoie des réponses aux clients via le réseau. Cette classe d'applications est intéressante pour l'étude du parallélisme car les requêtes peuvent être traitées en parallèle lorsqu'elles sont indépendantes, ce qui est souvent le cas. Il y a deux manières d'exploiter ce parallélisme inhérent aux serveurs de données. On peut tout d'abord associer le traitement d'une requête à chaque cœur. Si le nombre de requêtes est suffisant, alors tous les cœurs seront utilisés. Une deuxième méthode

1. *High-Performance Computing* - Calcul Haute Performance.

est d'associer une sous-partie de la chaîne de traitement d'une requête à chaque cœur. Pour cela, on peut imaginer un découpage des traitements en lecture réseau, puis analyse des requêtes, traitement des requêtes proprement dit, puis envoi des réponses.

Malgré ce parallélisme inhérent, des études précédentes montrent que des serveurs de données n'ont pas un passage à l'échelle idéal en multi-cœur [83]. Au vu de ce constat, nous souhaitons étudier plus avant les performances des serveurs de données dans un contexte multi-cœur, et analyser leur passage à l'échelle avec le nombre de cœurs. Plus précisément, nous voulons déterminer la part de gains de performances possibles par les couches logicielles dans cette situation. Pour cela, nous étudions dans un premier temps les performances d'un environnement d'exécution multi-cœur. Dans un second temps, nous comparons différentes architectures de serveurs Web en multi-cœur.

4.2 Optimisations d'un environnement d'exécution événementiel multi-cœur

Nous avons discuté des limitations des modèles de programmation classiques au Chapitre 2. Nous pensons que le modèle événementiel est un bon paradigme pour concevoir des serveurs de données efficaces. Ce constat se base principalement sur trois points. Premièrement, en mono-cœur, l'ordonnancement coopératif empêche les accès concurrents aux données (*races*). Cela a pour effet de grandement simplifier la gestion des données partagées (une des limitations du modèle de threads). Deuxièmement, les traitant d'événements demandent moins de ressources que les threads (empreinte mémoire plus petite, coûts de changements de contextes fortement réduits). Enfin, nous pensons que le modèle événementiel convient bien pour décrire les serveurs de données. En effet, si l'on considère les requêtes comme des événements, alors le serveur est un ensemble de traitants d'événements. L'inversion de contrôle propre aux traitants d'événements est donc un atout ici.

Cependant, comme nous l'avons vu au Chapitre 2, le modèle événementiel ne permet pas en soi de tirer parti des architectures multi-cœur. En effet, dans le modèle événementiel classique, il n'y a qu'une boucle de contrôle, au sein d'un seul thread. Il existe deux méthodes pour permettre au modèle événementiel de tirer parti des multi-cœur.

L'approche la plus simple, nommée N-COPY, consiste à dupliquer le processus, de façon à avoir une instance par cœur. Ainsi, chaque cœur exécute une boucle de contrôle, et est en mesure de traiter des événements. Cependant, comme chaque cœur exécute un processus indépendant, les communications entre les copies sont limitées. Cela pose plusieurs difficultés, comme notamment la répartition de charge entre les cœurs, qui doit être faite en amont.

La deuxième approche se base l'annotation d'événements. Chaque événement se voit ainsi doté d'une *couleur* par le programmeur. Le support d'exécution est en charge de garantir que deux événements de même couleur s'exécutent en exclusion mutuelle. Ici, un thread est créé par cœur, et la mémoire est donc partagée entre les cœurs.

Libasync-smp [93] est l'implantation de référence du modèle événementiel avec coloration d'événements. Nous commençons notre étude par montrer les limitations de Libasync-smp en multi-cœur. Pour cela, nous utilisons deux bancs d'essai représentatifs de schémas d'applications événementielles. Ces bancs d'essai nous permettent de pointer deux types de limitations au sein de Libasync-smp. Tout d'abord nous montrons qu'une mauvaise gestion de la mémoire au sein de Libasync-smp empêche le passage à l'échelle des applications. Cela est notamment dû à la présence de faux-partage. Nous exposons ensuite des coûts de communications inter-cœurs importants.

Après avoir identifié différents points dans Libasync-smp empêchant le passage à l'échelle des applications, nous proposons une série de quatre optimisations pour y remédier. Deux de ces optimisations permettent d'empêcher les situations de faux-partage observées. Nous proposons notamment un nouvel allocateur mémoire, EMA (*Event-driven Memory Allocator* - Allocateur mémoire événementiel), fortement couplé avec le système de coloration d'événements de Libasync-smp. Les deux autres optimisations permettent de limiter et d'amortir les coûts de communications inter-cœurs.

Nous validons chacune de ces optimisations sur nos bancs d'essai. Puis nous évaluons notre environnement d'exécution optimisé sur des applications réelles. Pour cela, nous étudions les performances d'un serveur Web événementiel. Nous modifions également le cœur de la bibliothèque de MapReduce en multi-cœur, Phœnix [67], pour la rendre événementielle. Cela nous permet d'évaluer nos optimisations sur la suite d'applications de Phœnix. Nous montrons ainsi que notre environnement d'exécution optimisé améliore les performances du serveur Web jusqu'à 111% par rapport à la version originale de Libasync-smp, et jusqu'à 66% par rapport à la bibliothèque Phœnix.

4.3 Analyse comparative d'architectures de serveurs Web en multi-cœur

Nous avons construit au Chapitre 2 une comparaison qualitative des modèles de programmation. Notre deuxième contribution se veut une comparaison quantitative des performances de certains modèles de programmation en multi-cœur. Nous sélectionnons les modèles les plus utilisés pour concevoir des serveurs de données, à savoir le modèle de threads, l'événementiel, et le modèle à étages. L'objectif étant de déterminer si certains modèles sont plus à mêmes que d'autres de présenter de bonnes performances en multi-cœur.

Pour réaliser notre étude, nous étudions différents serveurs Web, chacun représentatif d'un modèle de programmation précis. Notre choix reprend les implantations comparées par Pariag *et al.* [64] et Harji [42, 43]. Nous choisissons donc Knot, μserver et Watpipe, chacun représentatif respectivement des modèles threadé, événementiel et à étages. Cependant, contrairement aux études précédentes, nous mettons l'accent sur l'aspect multi-cœur de cette comparaison, et le passage à l'échelle des serveurs avec le nombre de cœurs.

Nous observons que les implantations suivant les modèles à étages et événementiel ont de meilleures performances que celle représentant les threads à 1 cœur. Cependant, ces tendances s'inversent à 4 cœurs, où l'implantation threadée devient meilleure. Les trois serveurs étudiés ont des performances similaires à 8 cœurs. Notre analyse nous permet de conclure à une limite matérielle au niveau du bus d'instructions due aux communications inter-cœurs. Ces communications sont induites par le protocole de cohérence de caches. Pour palier ce problème, nous étudions différents déploiements N-COPY possibles pour chaque serveur. Nous montrons ainsi que les déploiements N-COPY tenant compte de l'architecture matérielle, et notamment de la hiérarchie mémoire mènent à de meilleures performances. Nous montrons une amélioration du débit de 13% à 4 cœurs, qui permet aux modèles événementiel et à étages d'atteindre des performances similaires à celles du modèle threadé. Cependant malgré cela, le problème de passage à l'échelle persiste à 8 cœurs. Nous concluons donc notre étude en fournissant des pistes pour remédier à cette difficulté.

Chapitre 5

Optimisations d'un environnement d'exécution événementielle multi-cœur

Sommaire

Les récentes avancées logicielles, notamment le système d'exploitation Barrelfish [5] et l'étude de Pariag *et al.* [64], laissent penser que le modèle événementiel est une bonne approche pour la conception de serveurs de données. Nous avons également vu que les architectures multi-cœur sont de plus en plus prédominantes, et que la tendance est à l'augmentation du nombre de cœurs. Il est donc important que les environnements d'exécution événementielle puissent tirer parti des architectures multi-cœur. C'est le but l'approche prise par la coloration d'événements, illustrée notamment par Libasync-smp [93]. Cette approche introduit des annotations simples permettant aux développeurs de paralléliser leurs applications de façon incrémentale.

Dans ce chapitre, nous étudions les performances de Libasync-smp en multi-cœur. Nous constatons une certaine dégradation des performances avec Libasync-smp lorsque le nombre

de cœurs augmente. Nous identifions des baisses de performances à deux niveaux. Pour chaque limitation identifiée, nous proposons une optimisation correspondante, que nous implantons dans Libasync-smp. Nous étudions tout d'abord la gestion de la mémoire au sein de l'environnement d'exécution. Cela nous permet notamment de concevoir EMA, un allocateur mémoire efficace pour les environnements d'exécution événementiels avec coloration d'événements. Nous nous tournons ensuite vers l'optimisation des communications inter-cœurs. Enfin, nous évaluons les performances de nos optimisations sur des bancs d'essai ainsi que des applications réalistes. Cela nous permet de montrer que notre environnement d'exécution optimisé améliore jusqu'à 111% les performances d'un serveur Web par rapport à la version originale de Libasync-smp.

5.1 L'environnement Libasync-smp

Dans cette section, nous présentons plus en détails Libasync-smp, un environnement d'exécution événementiel multi-cœur. Nous commençons tout d'abord par un aperçu de la programmation événementielle colorée. Nous détaillons ensuite le fonctionnement interne de Libasync-smp. Enfin, nous présentons deux bancs d'essai, représentatifs d'applications types, ainsi que les limitations de performances qu'ils mettent en lumière dans Libasync-smp.

5.1.1 Programmation parallèle avec coloration d'événements

Comme nous l'avons vu précédemment, le modèle événementiel dans sa version classique ne gère pas le parallélisme. Il est donc inadapté aux architectures multi-cœur. Pour pallier cette limitation, Zeldovich *et al.* [93] introduisent dans leur étude le concept de coloration d'événements. Dans ce modèle, la gestion de la concurrence se fait en fonction d'annotations de code.

Plus précisément, une couleur est associée à chaque événement, et Libasync-smp assure que deux événements de même couleur ne s'exécuteront pas en concurrence. Une couleur par défaut est associée aux événements qui ne sont pas annotés ; cela permet de garantir la correction de l'application. La coloration des événements d'une application permet donc une parallélisation incrémentale.

L'implantation de Libasync-smp est basée sur celle de Libasync. Libasync est une bibliothèque écrite en C++ pour aider au développement d'applications événementielles efficaces. Libasync est utilisée dans de nombreux services réseau [33, 34, 23, 52, 58, 59, 76, 77, 87]. On y trouve notamment la fonction `wrap`, qui a pour but de créer un objet événement (appelé `callback` dans le code). Cette fonction prend en paramètres une fonction (le traitant de l'événement), ainsi que les paramètres correspondants à l'événement. L'événement ainsi créé peut ensuite être utilisé de plusieurs manières. Il peut être enregistré (ou "posté") directement pour exécution, en utilisant la primitive `cpucb` ou `cpucb_tail`, pour poster respectivement en tête ou en queue de la file d'événements. Il peut aussi être associé à un descripteur de fichiers avec `fdcb`. Dans ce cas, l'événement n'est posté que lorsque Libasync détecte de l'activité sur le descripteur de fichier. Le développeur doit spécifier s'il s'agit d'une scrutation pour une activité en lecture ou en écriture. Il est également possible d'associer des événements à des signaux POSIX avec `sigcb`, de programmer l'exécution d'un événement à un temps donné avec `timecb`, ou encore de retarder d'un temps donné en secondes l'exécution d'un événement avec `delaycb`.

Du point de vue de l'interface de programmation, Libasync-smp rajoute à Libasync une primitive `cwrap` qui permet de créer un objet événement coloré. Tout comme `wrap`, `cwrap`

prend en paramètres une fonction (le traitant de l'événement), ainsi que les paramètres correspondants à l'événement, et rajoute un troisième paramètre entier : la couleur de l'événement. Le principe est que deux événements de même couleurs soient exécutés en exclusion mutuelle. La primitive `wrap` est revisitée pour associer la couleur 0 aux événements, ainsi Libasync-smp est rétro-compatible avec Libasync.

Certaines hypothèses sont faites sur les événements dans le modèle événementiel. Par exemple, comme il n'y a pas de préemption (l'ordonnancement des traitants d'événements est fait de façon coopérative), il est souhaitable qu'un traitant d'événement ne se bloque pas. En effet, si un traitant se bloque, cela stoppe alors le progrès des autres événements de la file d'exécution. Cela a des conséquences néfastes sur les performances de l'application. Il est donc admis en général qu'un événement soit non-bloquant. Cela implique notamment l'utilisation d'E/S non-bloquantes, dont le support est présent sur les systèmes d'exploitation récents. Une autre implication est l'absence de prise de verrous et autres primitives de synchronisation, auxquelles il convient de préférer le système des couleurs, dans le cas multi-cœur (un traitant d'événement étant exécuté de façon atomique, aucune synchronisation ne devrait être nécessaire dans le cas mono-cœur). Ces hypothèses sont valables aussi bien dans Libasync que dans Libasync-smp.

Le système des couleurs a une limitation intrinsèque en termes d'expressivité, qui est plus faible que celle des verrous classiques. Il n'est en effet pas possible d'exprimer un programme avec une synchronisation complexe de type lecteur-rédacteur avec le système des couleurs. Dans ce cas, tous les événements manipulant la donnée partagée, que ce soit en lecture ou en écriture, se verront nécessairement associer la même couleur, et seront donc exécutés en exclusion mutuelle. Avec des verrous classiques, il est possible d'avoir une implantation de ce problème autorisant un accès parallèle aux lecteurs, tout en maintenant la cohérence avec une exclusion mutuelle entre plusieurs rédacteurs, et entre rédacteurs et lecteurs. Cela dit, les cas applicatifs où de tels schémas de synchronisation entrent en jeu sont peu fréquents, et la coloration d'événements a une expressivité suffisante pour définir la plupart des types de serveurs de données.

5.1.2 Fonctionnement interne

Pour garantir les propriétés d'exécution spécifiées par le système des couleurs, Libasync-smp fait le choix de toujours associer les événements de même couleur à un seul cœur. Cette association est gérée par une fonction de hachage simple. À chaque cœur est associé une file d'événements. De fait, l'environnement d'exécution doit garantir que deux événements de même couleur se retrouvent toujours dans la même file, ce qui est assuré par la cohérence de la table de hachage.

Libasync-smp permet également de surveiller des descripteurs de fichiers, avec un appel à la primitive `fdcb`. L'environnement d'exécution scrute lui-même l'ensemble des descripteurs de fichiers concernés. En pratique, il utilise un événement particulier posté régulièrement qui est chargé de scruter les descripteurs de fichiers via `select`. Cette scrutation appelle les traitants enregistrés si l'activité détectée sur ces fichiers correspond à celle pour laquelle ils ont été enregistrés.

Il est également possible de poster directement un événement depuis le code d'un traitant d'événement. Pour cela, il suffit d'utiliser la primitive `cpucb` ou `cpucb_tail`, qui enregistre un événement dans la file correspondante à la couleur associée, respectivement en tête de file ou en queue. Un appel asynchrone de ce type implique la suite d'opérations suivante : *a*) allouer en mémoire une structure contenant le contexte du nouvel événement *b*) déterminer le cœur

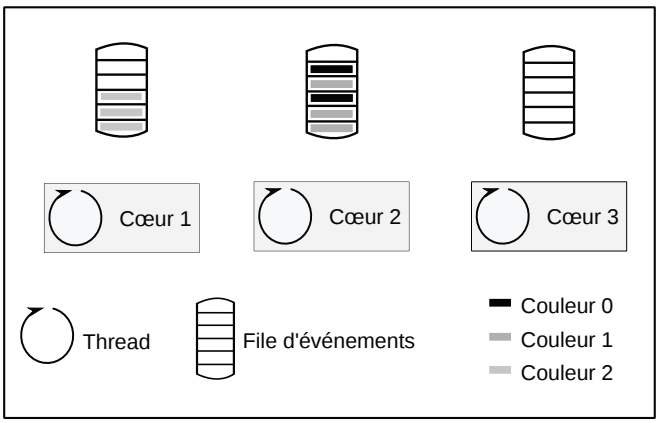

FIGURE 5.1 – Architecture interne de Libasync-smp

récepteur responsable du traitement de cet événement (en fonction de sa couleur) *c*) acquérir le verrou de la file d'événements du cœur récepteur *d*) enfiler l'événement chez le récepteur *e*) éventuellement réveiller le récepteur et finalement, relâcher le verrou du récepteur. De telles opérations entraînent donc des communications inter-cœurs, sous la forme de prise de verrou, échanges et modification de structures de données distantes, ou encore réveil d'un cœur distant.

Chaque cœur exécute la même boucle de contrôle, qui consiste à récupérer un événement de la file et appeler le traitant correspondant. Plus précisément, il s'agit de *a*) acquérir le verrou de la file d'événements, *b*) choisir l'événement à exécuter, *c*) relâcher le verrou et finalement, appeler le traitant correspondant. Lorsque sa file d'événements est vide, un cœur s'endort en utilisant la primitive `poll` sur un tube[1]. Il y a un tube par cœur. Le réveil d'un cœur s'effectue par une écriture dans le tube lui correspondant.

Un mécanisme d'équilibrage de charge est présent dans Libasync-smp. Il est basé sur le principe de vol de tâches. Les travaux de [37] ont montré que l'activation de ce mécanisme dégrade les performances de serveurs Web dans la plupart des cas. Nous désactivons donc ce dispositif lors de nos travaux. Le placement des événements est ainsi déterminé de façon statique : seule la fonction de hachage entre en jeu.

5.1.3 Modifications effectuées

Le code original de Libasync-smp de [93] ne fonctionne qu'avec la version 2.95 de gcc, et n'a été testé que sous la version 2.4.18 de Linux. Nous avons donc modifié cette base de code, pour la rendre compatible avec des distributions Linux plus récentes, et donc de nouvelles versions de gcc et de la glibc. Dans le souci d'améliorer les performances de l'environnement d'exécution, nous avons également remplacé l'utilisation de `select` par `epoll`. `Epoll` est une implantation de scrutation plus récente que `select`, et dont les performances passent mieux à l'échelle lorsque le nombre de fichiers surveillés augmente.

Dans la suite de ce chapitre, lorsque nous mentionnons la version patrimoniale, ou originale, de Libasync-smp, nous faisons référence à la version modifiée décrite ici, sans les autres optimisations décrites plus loin.

1. *Pipe* Unix

5.1.4 Limitations

Lors de nos premières expérimentations avec notre implantation de serveur Web sur Libasync-smp, nous remarquons un certain nombre de limitations de performances. Pour mettre en lumière ces limitations, nous avons conçu deux bancs d'essais pour Libasync-smp. Tout au long de ce chapitre, les résultats relatifs à ces deux bancs d'essai présentent une déviation toujours inférieure à 4%.

Conditions expérimentales. Nos expérimentations sont conduites sur une machine 8 cœurs, munie de 2 processeurs quadri-cœurs Intel E5410, nom de code `Harpertown`. Chaque processeur est composé de 4 cœurs cadencés à 2.33GHz et groupés par paires. Chaque paire de cœurs partage un cache L2 de 6Mo. Les temps d'accès à la mémoire sont uniformes. Cette machine est équipée de 8Go de RAM, ainsi que de 8 cartes réseau Ethernet 1Gb/s. Dans ce chapitre, les machines utilisées fonctionnent sur un noyau Linux 32bits 2.6.24, où le support des compteurs de performances est activé. Notre environnement d'exécution et les applications testées sont compilées avec Gcc version 4.3.2 et l'option d'optimisation -O2, et sont liées à la Glibc version 2.7.

Faux-partage. Le premier banc d'essai est baptisé `ping-pong local`. Le principe est d'avoir deux types d'événements : `ping` et `pong`. Chaque événement `ping` enregistre un événement `pong` de la même couleur, et vice et versa. Ces événements ne font rien d'autre, et on considère leur empreinte mémoire et leur temps de calcul négligeables.

Chaque cœur fait un traitement des événements des couleurs qui lui sont attribuées, indépendemment des autres. Il n'y a donc a priori aucune communication inter-cœurs nécessaire, et on s'attend à un passage à l'échelle avec le nombre de cœurs idéal. Autrement dit, on s'attend à retrouver toujours le même débit d'événements quel que soit le nombre de cœurs utilisés.

Les résultats sont très loin de nos attentes. En effet, le débit à 8 cœurs connaît une baisse de 84% par rapport à celui à 1 cœur. Nous observons un comportement pathologique en cache. En effet, le nombre d'accès au cache L2 par événement traité augmente de 0 à 11 lorsqu'on passe de 1 à 8 cœurs. Ce nombre recouvre à la fois les fautes et les succès dans le cache L2. Ce que nous observons correspond donc à une augmentation drastique des fautes de cache L1, qui génèrent des accès au cache L2. Certains de ces accès en L2 correspondent à des fautes de cache L2, très coûteuses car le L2 est ici le dernier niveau de cache, et implique donc des accès vers la mémoire centrale.

Nous expliquons l'augmentation des fautes de cache L1 par un phénomène mémoire de faux-partage [81] : plusieurs cœurs modifient des données différentes, mais présentes dans la même ligne de cache. Le mécanisme de cohérence de cache entre alors en jeu pour faire en sorte que toutes les copies en cache soient cohérentes. Son fonctionnement se base sur l'invalidation en cache des copies obsolètes, d'où l'augmentation des fautes de cache L1. Cette situation réduit les performances dès que 2 cœurs modifient des données sur la même ligne de cache.

Communications inter-cœurs. Le deuxième banc d'essai utilisé ici se base sur le modèle de producteur-consommateurs. Dans cette application, un cœur `producteur` enregistre et répartit équitablement 100 événements entre tous les cœurs présents (lui compris). Tous les cœurs deviennent alors des `consommateurs` et exécutent les événements enregistrés. Lorsqu'il a fini, le cœur `producteur` répète l'opération.

L'idée est ici de reproduire un comportement bien connu à la fois dans les serveurs de données et dans les applications de calcul parallèle. Ce comportement comprend une phase de `production`, dans laquelle un cœur est en charge d'accepter de nouvelles connexions, ou bien de recueillir de nouveaux calculs, puis de distribuer ces traitements à tous les cœurs disponibles. Vient ensuite la phase de `consommation`, où chaque cœur effectue les traitements qui lui sont attribués. Pour mieux mettre en valeur les limitations soupçonnées, le traitement des événements est vide, comme précédemment.

Nous observons une augmentation du temps passé dans des opérations de synchronisation entre 1 et 8 cœurs. En effet, à 8 cœurs, le producteur passe 69% de son temps dans des primitives de synchronisation, contre 7% de temps passé dans les mêmes primitives à 1 cœur. Ce banc d'essai montre donc que les primitives de synchronisation de Libasync-smp passent mal à l'échelle avec le nombre de cœurs. Des applications faisant intervenir beaucoup de communications inter-cœurs, comme par exemple celles basées sur un modèle producteur-consommateurs, verront leurs performances en pâtir.

5.2 Gestion efficace de la mémoire

Nos premières observations sur le faux-partage suggèrent une mauvaise gestion de la mémoire au sein de Libasync-smp. Dans cette section, nous étudions comment mieux gérer la mémoire pour obtenir de meilleures performances. Nous nous penchons tout d'abord sur les structures allouées statiquement, puis sur celles allouées dynamiquement dans un second temps.

5.2.1 Placements statiques

Dans un premier temps nous cherchons à éliminer le faux-partage sur les structures internes à Libasync-smp allouées statiquement. Pour cela, nous utilisons la technique du bourrage, qui consiste premièrement à aligner le premier octet de donnée sur le début d'une ligne de cache, et deuxièmement à allouer plus de mémoire que demandé, dans le but de remplir entièrement la dernière ligne de cache. Ainsi, on évite que deux données différentes se retrouvent dans la même ligne de cache, ce qui provoque le faux-partage. La contrepartie étant bien entendu une surconsommation de mémoire.

La version originale de Libasync-smp utilise déjà le bourrage sur quelques structures internes. Nous souhaitons aller plus loin ici, et bourrer systématiquement toutes les structures internes de l'environnement d'exécution. Ces structures ne sont pas en grand nombre, et le bourrage systématique de chacune demande en tout 80 lignes de cache, soit une surconsommation de 5Ko au total. Pour gagner sur cette empreinte mémoire, nous regroupons toutes les structures selon un principe simple. Il s'agit de regrouper les données relatives à un cœur sur un même ensemble de lignes de cache. Ainsi, nous évitons toujours le faux-partage, car les données regroupées ne sont accédées que par un seul cœur. En revanche, cela nous permet de réduire la consommation mémoire à seulement 8 lignes de cache, soit 512 octets.

La Figure 5.2 présente une comparaison des résultats obtenus avec et sans le bourrage des structures internes de Libasync-smp, sur notre banc d'essai `ping-pong local`. Les histogrammes blancs représentent la version originale de Libasync-smp, tandis que les gris représentent la version avec bourrage (les histogrammes noirs sont détaillés plus loin). Nous observons que le bourrage permet d'augmenter le débit du banc d'essai (de 164% à 8 cœurs) par rapport à la version originale, et que `ping pong local` passe alors à l'échelle jusqu'à 2 cœurs. En revanche, cette technique, si elle est toujours meilleure que la version originale (elle

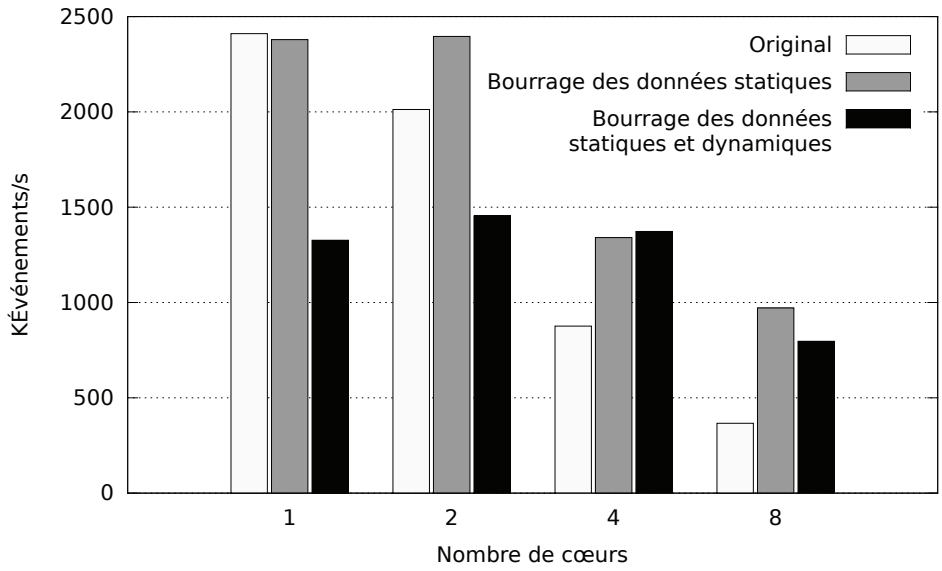

FIGURE 5.2 – Débit par cœur avec et sans bourrage des structures internes à Libasync-smp sur le banc d'essai `ping-pong local`.

multiplie le débit original par 2,6 à 8 cœurs), n'est pas suffisante pour permettre un passage à l'échelle idéal.

5.2.2 Allocation dynamique

Étant donné l'amélioration observée par le bourrage sur les données statiques, nous nous posons la question de l'efficacité du bourrage sur les données allouées dynamiquement.

Bourrage en cache

Pour réaliser un bourrage sur les structures allouées dynamiquement, nous surchargeons l'opérateur d'allocation mémoire `new`, pour utiliser la fonction `posix_memalign`. Cela permet d'aligner les données allouées sur des lignes de cache. Au besoin, nous agrandissons artificiellement la taille demandée pour remplir la dernière ligne de cache, et ainsi réaliser un bourrage similaire à celui fait sur les structures statiques.

La Figure 5.2 présente les résultats obtenus avec cette technique (histogrammes noirs), et permet de comparer les gains de performances par rapport au bourrage des données statiques, ainsi qu'à la version originale de Libasync-smp. On observe que le bourrage systématique de la mémoire allouée dynamiquement a un effet néfaste sur les performances. Pour déterminer les causes de ces observations, nous étudions le comportement en cache selon les différents types de bourrage.

Le faux-partage augmente sensiblement le nombre de fautes de caches L1 à cause des nombreuses invalidations. Chaque faute de cache L1 provoque un accès en L2. Un accès en cache résulte soit en un succès si la donnée accédée est présente dans le cache, soit en une faute si elle n'y est pas. Dans le cas du faux-partage, on observe des succès en L2 lorsque le cache L2 est commun aux deux cœurs qui rivalisent pour l'accès à une ligne de cache. En effet, dans ce cas, la faute de cache L1 peut être résolue en L2. Dans le cas contraire, quand les deux cœurs impliqués ne partagent pas de cache L2, alors on observe des fautes en L2.

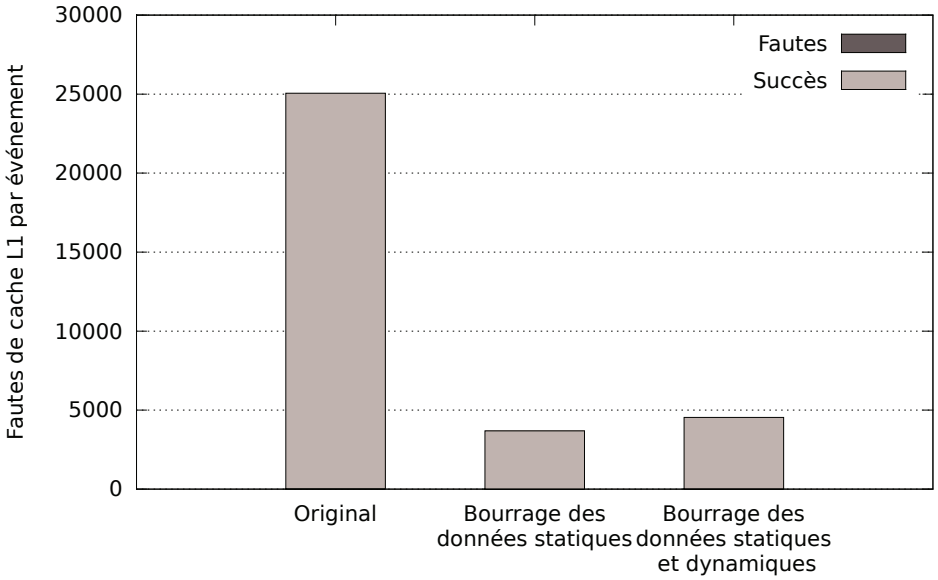

FIGURE 5.3 – Détail des accès aux caches L1 (données et instructions) lors du test `ping pong local` à 8 cœurs.

Dans ce dernier cas la résolution de la faute est faite en la mémoire centrale. On rappelle que, sur notre architecture, les cœurs partagent un cache L2 deux à deux.

La Figure 5.3 présente le nombre d'accès aux caches L1 par événement sur notre `ping pong local` à 8 cœurs. Le nombre de fautes de caches L1 est négligeable face au nombre de succès. On voit que la version originale demande plus de 4 fois plus d'accès que les versions avec bourrage des données. Cela illustre de façon nette la présence de faux-partage dans ce test. En effet, dans une situation "normale" (sans faux-partage), comme le bourrage demande plus de mémoire pour stocker les données, le nombre d'accès au cache L1 devrait être plus élevé pour les versions avec bourrage par rapport à la version originale. Le fait que cette tendance soit inversée de façon si importante montre la présence de faux-partage. Comme une faute de cache L1 se répercute toujours en un accès au cache L2, les différences entre les fautes de cache L1 par événement sont visibles à la Figure 5.4. Comparer la somme du nombre de succès et de fautes en L2 par événement revient en effet à comparer le nombre de fautes en L1 par événement.

La Figure 5.4 présente le nombre d'accès aux caches L2 par événement sur notre `ping pong local` à 8 cœurs. La version originale fait près de 300 accès en cache L2 par événement traité, dont 220 sont des fautes. Le bourrage des structures allouées statiquement permet de réduire de 82% les accès aux caches L2, en comparaison avec la version originale. Cette baisse se retrouve autant au niveau des succès (-82%) que des fautes (-86%) de caches. Cela signifie que la réduction du faux-partage est valable pour tous les cœurs : que ceux-ci partagent un cache L2 ou non. Ceci nous montre que globalement, le bourrage des données statiques est une bonne approche pour éviter le faux-partage.

la Figure 5.4 montre que le bourrage systématique des données dynamiques en plus des données statiques réduit également le nombre d'accès aux caches L2 par rapport à la version originale. En revanche, on observe une augmentation du nombre d'accès aux caches par rapport au bourrage des seules données statiques. Cette augmentation est due à la surconsommation mémoire induite par le bourrage, qui dépasse alors la capacité des caches. Dans notre banc d'essai, la taille moyenne des structures de données augmente de 38% avec le

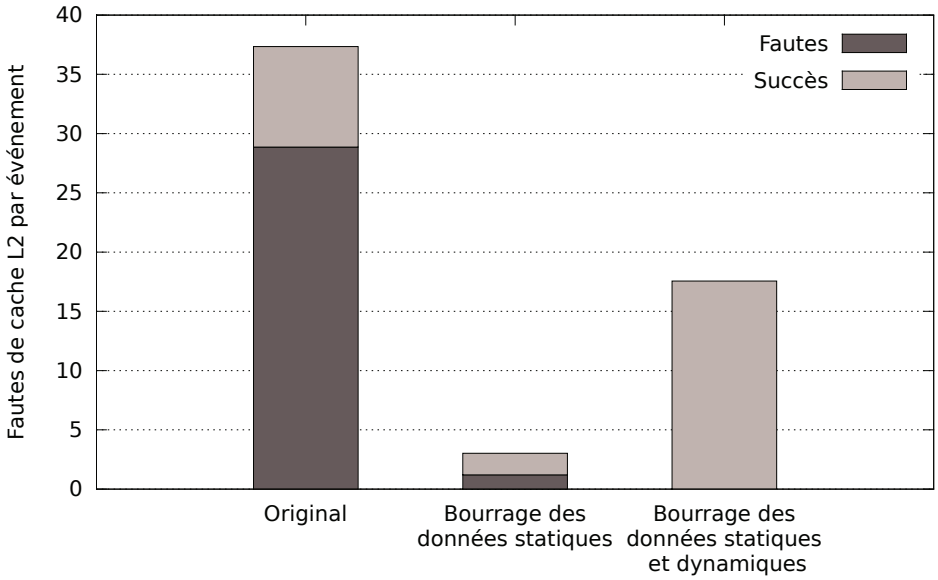

FIGURE 5.4 – Détail des accès aux caches L2 lors du test `ping pong local` à 8 cœurs.

bourrage des structures dynamiques. Cela se traduit par l'apparition de fautes de cache : près de 140 fautes de cache par événement soit 10 fois plus qu'avec le bourrage des données statiques seules. Cette technique, bien que meilleure que la version originale à partir de 4 cœurs, est moins intéressante que le bourrage des données statiques. Nous nous tournons donc vers d'autres techniques d'allocation dynamique de mémoire efficaces pour résoudre les problèmes de faux-partage sur les données dynamiques.

Allocateurs mémoire

Nous avons vu que le bourrage des données statiques est une technique efficace pour éviter le faux-partage. Toutefois, le faux-partage peut aussi intervenir sur les données alloués dynamiquement. Or nous avons vu jusqu'ici que le bourrage des structures dynamiques ne permet pas un passage à l'échelle idéal sur notre test `ping pong local`. Nous nous tournons dans cette section vers des allocateurs mémoire efficaces. Notre objectif est de trouver une méthode d'allocation dynamique permettant de faire passer à l'échelle notre `ping pong local`. Nous testons plusieurs allocateurs mémoire de l'état de l'art. Nous décrivons d'abord brièvement leur fonctionnement interne, avant de détailler les résultats obtenus.

Nous étudions en premier lieu l'allocateur mémoire de la glibc, nommé ptmalloc, de [91]. Il est communément utilisé, et n'est pas spécialement optimisé pour les architectures multi-cœur. Son principal objectif est de réduire la fragmentation, la réduction du faux-partage ne fait pas partie de ses attributions. Le tas du processus est divisé en `arènes`. La notion d'`arène` permet de découper la gestion du tas global entre les threads qui qui souhaitent allouer de la mémoire. Lorsqu'un thread veut allouer de la mémoire, il parcourt la liste des `arènes` disponibles, en commençant par la dernière qu'il a utilisé. Il prend un verrou sur la première `arène` disponible, ou en crée une s'il n'en existe pas. Les morceaux de mémoires (*chunks*) alloués sont obtenus en découpant la mémoire disponible dans l'arène.

Les autres allocateurs étudiés ici ont été conçus avec les architectures multi-cœur en tête. Leurs principaux objectifs sont d'améliorer la localité de cache et de diminuer la fragmentation. Pour cela, l'idée est de garder les objets alloués par chaque thread au plus proche en

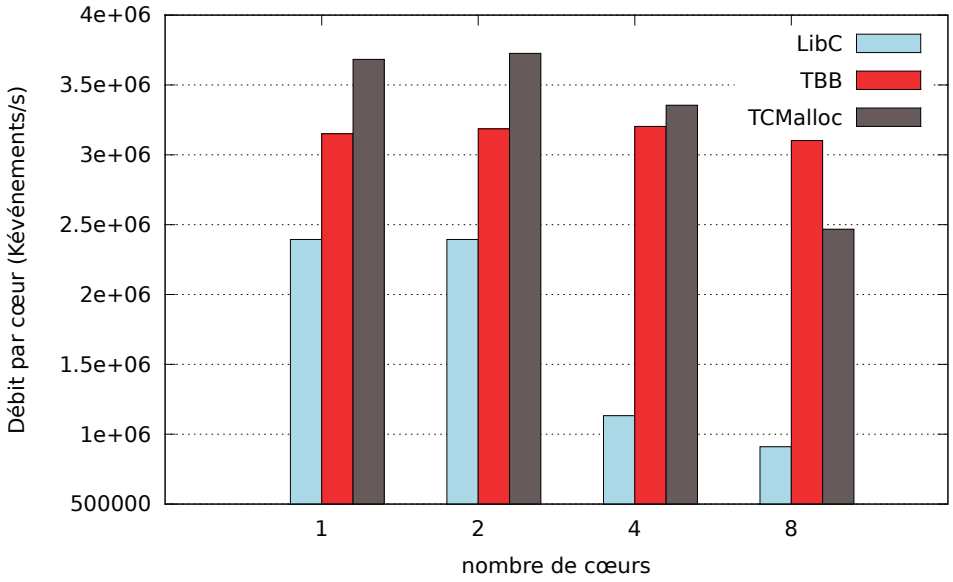

FIGURE 5.5 – Débit de `ping pong local` en fonction de l'allocateur mémoire et du nombre de cœurs

mémoire de leur propriétaire. Ces allocateurs essayent également de réduire le faux-partage en allouant les objets de threads différents sur des lignes de caches différentes. Bien qu'ils soient différents et aient leurs spécificités propres, on peut dégager un fonctionnement commun à TCMalloc [39] et TBB d'Intel [54]. Chaque thread maintient plusieurs listes d'objets allouables. Cela évite notamment la prise de verrou, qui limite le passage à l'échelle lorsque le nombre de cœurs devient trop grand. Cela permet aussi de tirer parti de la localité de cache, dans la mesure où chaque thread gère seul ses propres listes. Chaque liste d'objets correspond à un éventail de tailles allouables. Par exemple, tous les objets entre 17 et 20 octets sont alloués dans la liste des objets de 20 octets. Ces listes sont souvent allouées depuis des blocs alignés sur des lignes de caches, pour éviter le faux-partage. L'avantage de cette architecture est qu'elle permet un cas normal d'utilisation très rapide (fastpath). En effet, il suffit de retrouver la liste correspondante à la taille demandée et d'en choisir le premier élément.

TCMalloc et TBB diffèrent dans leur représentation d'une liste d'objets. Pour chaque taille d'objet et chaque thread, TCMalloc utilise une simple liste d'objets libres. Lorsque cette liste est vide, d'autres objets de cette taille sont alloués depuis le tas global, commun à tous les threads. Au contraire, TBB choisit une gestion à deux niveaux d'indirections. Pour chaque taille d'objet et chaque thread, TBB gère une liste doublement chaînée de blocs. Chaque bloc contient une liste d'objets allouables, similaire à celles de TCMalloc. Pour éviter la fragmentation, un seul bloc est actif à chaque instant, par thread et par taille d'objet. Lorsqu'un bloc est plein, il est placé à la fin de la liste et un nouveau bloc actif est choisi parmi ceux présents en début de liste. Lorsque suffisamment d'objets ont été libérés dans un même bloc, ce bloc repasse en tête de liste.

La Figure 5.5 et le Tableau 5.1 présentent les résultats observés sur notre banc d'essai `ping pong local` avec différents allocateurs mémoire. On retrouve le passage à l'échelle limité de ptmalloc (discuté Figure 5.2. En effet, le temps d'allocation mémoire de ptmalloc augmente de 205% entre 1 et 8 cœurs. Cette limitation est dûe à deux choses. Tout d'abord elle s'explique part par le fait que tout les threads accèdent de façon concurrente à la liste des arènes, et

Allocateur	Temps d'allocation	Augmentation / 1 cœur
LibC (ptmalloc)	498 cycles	205%
TCMalloc	97 cycles	24%
TBB	113 cycles	7%

TABLE 5.1 – Temps d'allocation de chaque allocateur à 8 cœurs sur `ping pong local`, avec l'augmentation correspondant au passage de 1 à 8 cœurs.

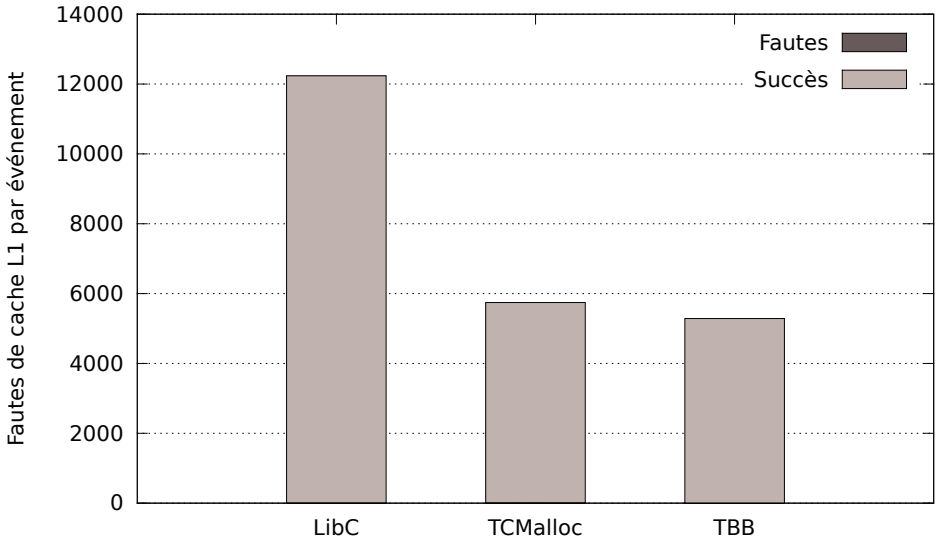

FIGURE 5.6 – Détail des accès aux caches L1 sur `ping pong local`à 8 cœurs, selon l'allocateur mémoire.

les coûts de synchronisation associés augmentent avec le nombre de cœurs. Ensuite, ptmalloc n'empêche pas le faux-partage, et deux threads peuvent allouer des données depuis la même arène, et donc potentiellement la même ligne de cache.

Nous observons sur la Figure 5.5 que TCMalloc et TBB maintiennent de bonnes performances, même si TBB se comporte mieux à 8 cœurs que TCMalloc. Ce comportement peut s'expliquer par le fait que ces allocateurs gèrent uniquement des structures par thread, ce qui permet de conserver un coût d'allocation presque constant quelque soit le nombre de cœurs (pas de synchronisation nécessaire). De plus, TBB et TCMalloc s'assurent au maximum d'éviter des situations faux-partage lorsque chaque cœur alloue la mémoire qu'il utilise.

On observe toutefois que TCMalloc voit ses performances 22% moins bonnes que TBB à 8 cœurs. Une explication possible à cela est la dûe à une possibilité de faux-partage au sein de TCMalloc. Sur la Figure 5.6, TCMalloc fait légèrement plus d'accès L1 que TBB à 8 cœurs. De plus, on observe un nombre d'accès aux caches L2 bien plus important pour TCMalloc que pour TBB, et même supérieur à celui de ptmalloc sur la Figure 5.7. Cela montre la présence de faux-partage, même s'il est bien moins important que celui observé avec la LibC. Plus précisément, lorsqu'un thread veut libérer un objet dans TCMalloc, il ajoute cet objet à sa liste locale correspondante. Cela peut causer une situation de faux-partage si un objet est libéré par un thread différent de celui qui l'alloue et l'utilise. Comme TBB gère la libération des objets de manière différente, il n'est pas sujet à ce phénomène. Les événements `ping` et `pong` de notre banc d'essai `ping pong local` sont libérés par le cœur qui les alloue et

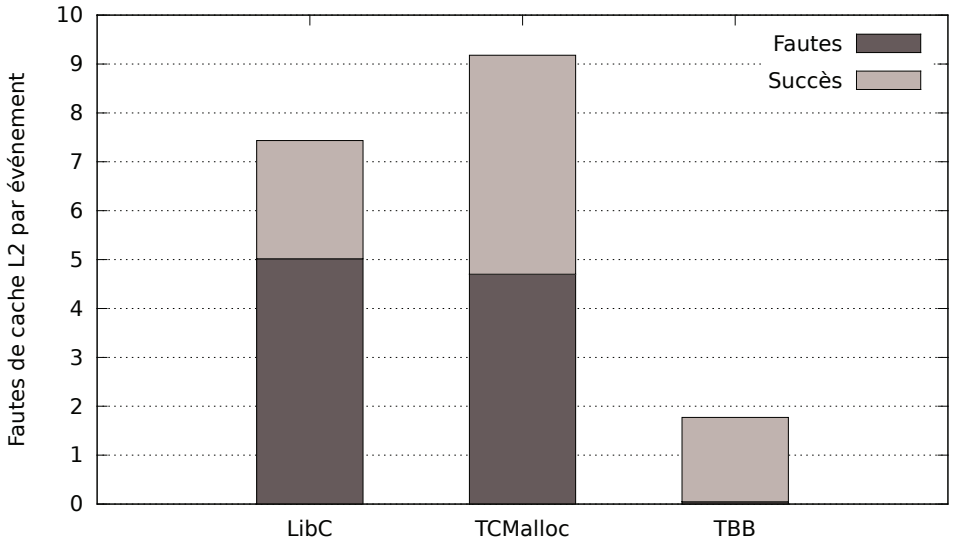

FIGURE 5.7 – Détail des accès aux caches L2 sur `ping pong local` à 8 cœurs, selon l'allocateur mémoire.

les utilise, et ne posent donc pas ce problème. Cependant, certains événements internes à Libasync-smp peuvent engendrer du faux-partage lors de leur libération (qui a lieu sur un cœur différent de leur allocation). Nous pensons que ces événements sont responsables du faux-partage observé avec TCMalloc.

La Figure 5.6 présente le nombre d'accès au cache L1 (données plus instructions) par événement sur `ping pong local` à 8 cœurs, selon l'allocateur mémoire. On observe que la LibC demande environ deux fois plus d'accès à la mémoire que TBB et TCMalloc, à charge de travail égale. L'allocateur de la LibC (ptmalloc) est donc à l'origine d'un faux-partage important. Le nombre de fautes en L1 est négligeable par rapport au nombre de succès en L1.

La Figure 5.7 présente le nombre d'accès au cache L2 (données plus instructions) par événement sur `ping pong local` à 8 cœurs, selon l'allocateur mémoire. On observe que la LibC et TCMalloc induisent environ 5 fautes de cache L2 par événement. En comparaison, TBB fait en moyenne bien moins d'une faute L2 par événement. On en déduit donc que le comportement de la LibC et de TCMalloc traduit une situation de faux-partage. TBB permet d'éviter le faux-partage ici, et maintient un faible taux d'accès au cache L2.

Producteur-consommateurs. Nous menons ensuite les mêmes expériences sur notre deuxième banc d'essai, producteur-consommateurs. Nous avons modifié ce banc d'essai pour ces tests mémoire, de façon à mieux mettre en avant les situations de faux-partage. Chaque événement produit contient maintenant un petit tableau de 22 octets. À la place des traitants vides, chaque consommateur exécute 50 fois une même procédure. Cette procédure consiste à remplir le tableau de valeurs aléatoires, puis de les trier selon l'algorithme du tri à bulles. L'objectif est ici de reproduire un traitement sur des données de petite taille passées en paramètre aux consommateurs.

La Figure 5.8 montre les résultats obtenus. Les tendances s'inversent totalement sur ce type d'application par rapport au `ping pong local`. De façon intéressante, le meilleur débit est atteint à 2 cœurs consommateurs, où ptmalloc affiche un débit nettement supérieur à

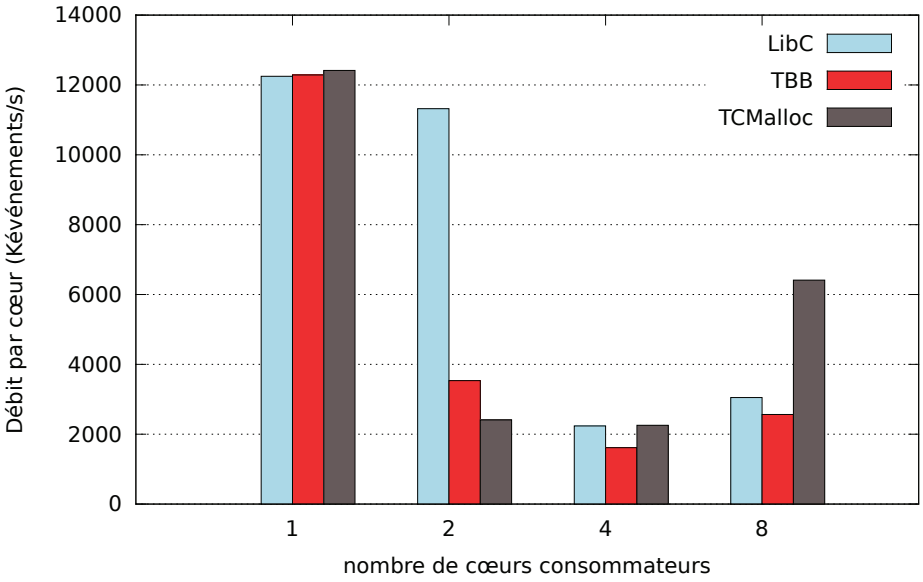

FIGURE 5.8 – Débit du banc d'essai producteur-consommateurs, selon l'allocateur mémoire utilisé et le nombre de cœurs consommateurs.

celui de TBB et TCMalloc. Ces deux derniers ont un faible débit car, sur cette application, tous les événements sont alloués par le producteur, et proviennent donc de la même zone mémoire. Lors de leur traitement par les consommateurs, le faux-partage apparaît nettement (cf. Figures 5.9 et 5.10).

Au contraire, nous observons ici que la politique d'allocation de ptmalloc permet d'éviter le faux-partage entre le producteur et les consommateurs à 2 cœurs. En effet, ptmalloc alloue ses objets de manière à éviter au maximum la fragmentation. Pour cela, il a été montré [29] que l'algorithme du *nearest-fit* (plus proche) n'était pas satisfaisant, et l'algorithme du *best-fit* (meilleur choix) conserve une faible fragmentation. Suivant ce principe, ptmalloc alloue les événements dans des arènes différentes (ou bien dans des chunks non contigus), évitant ainsi le faux-partage entre producteur et consommateurs. De plus, comme toutes les allocations sont faites par le thread producteur, les temps d'allocation de ptmalloc ne souffrent pas de coûts de synchronisation exorbitants.

La Figure 5.9 présente le nombre d'accès au cache L1 (données plus instructions) par événement sur notre test producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire. On observe que la LibC demande 65% moins d'accès à la mémoire que TBB et 75% que TCMalloc, à charge de travail égale. On en déduit que de façon surprenante TCMalloc et TBB sont à l'origine d'un faux-partage important, alors que la LibC permet ici d'éviter le faux-partage. Le nombre de fautes en L1 est négligeable par rapport au nombre de succès en L1.

La Figure 5.10 présente le nombre d'accès au cache L2 (données plus instructions) par événement sur notre producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire. On observe que ptmalloc diminue de plus de 90% le nombre de fautes L2 par événement par rapport à TCMalloc et TBB. Cela corrobore nos précédentes observations sur le fait que ptmalloc élimine tout faux-partage ici, au contraire de TCMalloc et TBB.

L'ajout d'autres cœurs consommateurs modifie fortement les tendances, et aucun allocateur ne passe à l'échelle au delà de 2 cœurs. On observe à 4 cœurs de mauvaises performances parmi les trois allocateurs testés. Pour les trois allocateurs à 4 cœurs, on observe que plus de

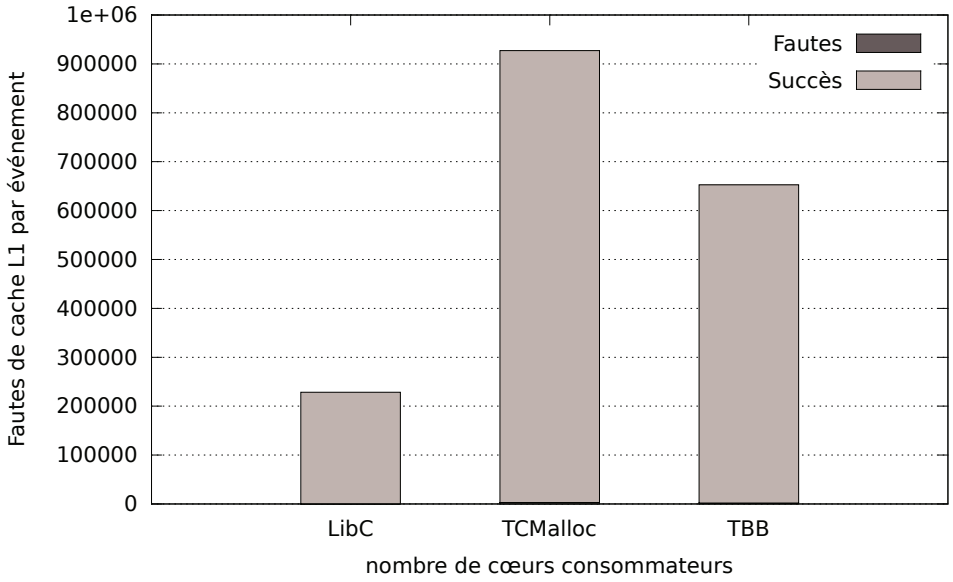

FIGURE 5.9 – Détail des accès aux caches L1 sur producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire.

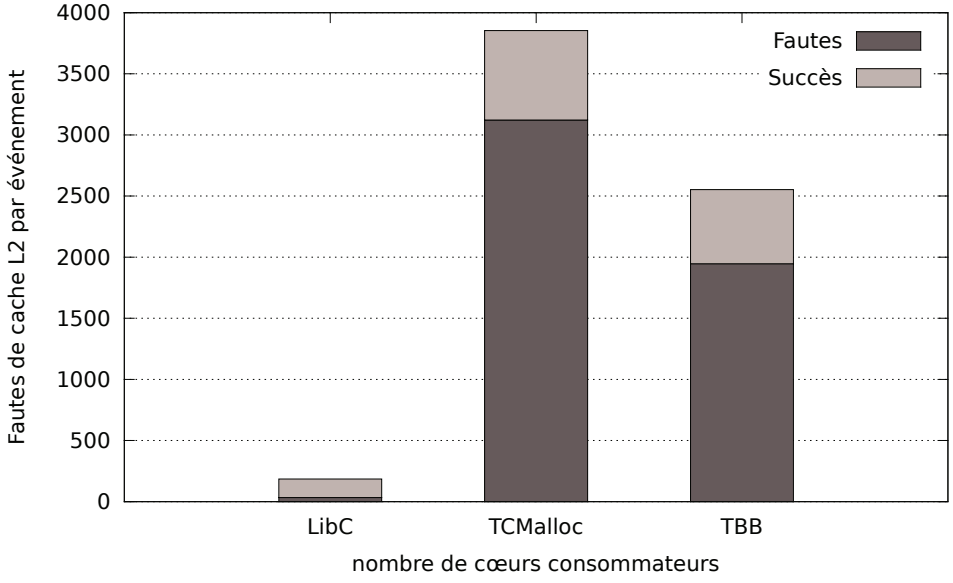

FIGURE 5.10 – Détail des accès aux caches L2 sur producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire.

80% des accès au cache L2 causent des fautes en L2. La situation de faux-partage est donc responsable du faible débit à 4 cœurs.

À 8 cœurs, on voit que TCMalloc a un meilleur débit que TBB et ptmalloc. En analysant le débit et les fautes de caches sur chaque cœur avec TCMalloc, on observe des asymétries entre les cœurs. Les cœurs 0, 1, 2 et 7 ont des débits quatre fois supérieurs à ceux des autres cœurs (3, 4, 5 et 6). De plus, ces cœurs "rapides" (0, 1, 2 et 7) font de l'ordre de 10 fois moins de fautes en cache L2 par événement que les autres. Visiblement, l'implantation de TCMalloc utilise des zones mémoires diminuant le faux-partage entre ces cœurs par rapport aux autres dans notre test à 8 cœurs. Cela résulte dans des performances favorisées dans l'ensemble. Les performances de ptmalloc et TBB à 8 cœurs sont comparables entre elles, et le profilage des caches montre une forte présence de faux-partage (plus de 2K fautes de cache L2 par événement).

Bilan sur les allocateurs testés. Nous montrons donc qu'aucun allocateur mémoire n'a un comportement toujours satisfaisant lors de nos expérimentations sur deux bancs d'essai. TBB et TCMalloc, de par leur architecture, ont de très bonnes performances dans le cas d'une application consistant de flots d'événements indépendants alloués et traités sur le même cœur. Dans ce cas, ptmalloc affiche de mauvaises performances à 8 cœurs. On observe cependant une inversion radicale de ces tendances lorsqu'on considère une application basée sur un modèle producteur-consommateurs, où les événements ne sont pas alloués sur le cœur qui les traite. Dans ce cas, TBB et TCMalloc voient leurs performances passer bien en dessous de celles de ptmalloc à 2 cœurs, à cause d'une situation de faux-partage. De plus, aucun des allocateurs testés n'affiche des performances passant à l'échelle avec le nombre de cœurs sur notre test producteur-consommateurs.

5.2.3 Un nouvel allocateur mémoire pour l'événementiel multi-cœur

Architecture. Au vu des résultats obtenus avec les allocateurs existants, nous proposons un nouvel allocateur mémoire, avec pour objectif de toujours afficher des performances satisfaisantes, et surtout un bon passage à l'échelle en multi-cœur. Nous avons donc réalisé EMA [2], un allocateur mémoire spécifique à Libasync-smp, qui reprend au maximum les bonnes pratiques des autres allocateurs. EMA permet d'éviter le faux-partage presque systématiquement. En effet, EMA tire parti des informations de Libasync-smp pour allouer la mémoire de la manière la plus efficace possible. De plus, EMA, de par son architecture, ne recourt que très rarement à des primitives de synchronisation. Cela lui permet d'allouer rapidement de la mémoire et de passer à l'échelle avec le nombre de cœurs.

L'architecture d'EMA présente des similarités avec celles de TBB et TCMalloc. Tout comme ces derniers, nous nous concentrons sur l'allocation des petits objets, pouvant provoquer du faux-partage, et laissons le système d'exploitation allouer les plus gros. Les objets de petite taille correspondent le plus souvent ici à des allocations d'événements, et ont donc une couleur associée. Nous utilisons également des listes d'objets allouables par tailles, pour réduire les coûts de synchronisation. Ces listes sont alignées sur des lignes de caches pour éviter tout faux-partage provenant de leurs objets. La principale différence d'EMA par rapport aux autres allocateurs est qu'il tire parti du mécanisme de coloration d'événements de Libasync-smp. Cela permet notamment de déterminer quel sera le cœur utilisateur (potentiellement différent du demandeur) d'un résultat d'allocation. On rappelle que, lorsqu'un cœur enregistre un événement à destination d'un autre cœur, c'est le premier qui fait l'allocation.

2. Event-driven Multicore Allocator (allocateur pour environnement d'exécution événementiel multi-cœur).

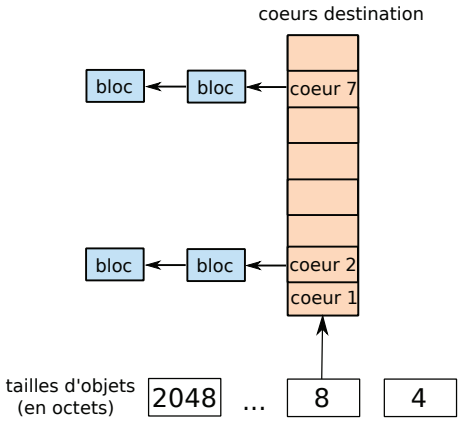

FIGURE 5.11 – Représentation interne d'EMA, privée à un cœur.

Le cœur destinataire est déterminé par EMA en fonction de la couleur qui est associée à l'événement alloué.

La Figure 5.11 présente l'architecture interne d'EMA, vue depuis un cœur. À la manière de TBB, nous choisissons une approche à deux niveaux d'indirections, et chaque bloc contient une liste d'objets allouables. La principale différence avec les allocateurs existants réside dans la table des cœurs destinataires possibles. EMA utilise sur chaque cœur une liste de blocs par taille et par cœur destinataire possible. Les listes d'objets de même taille mais de cœur destination différents sont donc dupliquées. Cela permet d'éviter au maximum l'utilisation d'opérations de synchronisation lors des allocations. Les seules opérations de synchronisation présentes dans EMA sont utilisées d'une part pour les accès au tas global, et d'autre part lors de la libération (car une libération d'un objet peut être faite par un cœur différent du cœur allocateur).

De cette façon, EMA est capable d'éviter le faux-partage dans des situations où un cœur alloue des objets à destination d'autres cœurs. Cette architecture permet aussi de garder une faible complexité dans les fonctions d'allocation et dé-allocation. Allouer un objet consiste simplement à retrouver la liste d'objets allouables, en fonction de la taille demandée et de la couleur de l'événement, puis à choisir le premier élément. La dé-allocation dans son cas le plus simple consiste à remettre l'objet libéré dans la liste correspondante à sa taille. Cependant, ce cas n'est valide que lorsque l'objet a été alloué par le cœur destinataire. Si le cœur destinataire n'a pas alloué l'objet, alors il incrémente le compteur des objets libérés pour ce bloc mémoire. Lorsque ce compteur montre que le bloc ne contient plus d'objets alloués (égal au nombre total d'objets), alors le cœur destinataire recycle ce bloc, le rendant réutilisable. Il est intéressant de noter qu'aucune de ces opérations ne nécessite l'utilisation de primitives de synchronisation, que ce soit des verrous ou des opérations atomiques. En effet, les structures internes d'EMA sont organisées de manière à minimiser le partage. Chaque thread est en charge de gérer ses listes d'objets, d'où l'absence d'accès concurrents, et donc de synchronisation, dans le cas commun (*fastpath*).

Il arrive toutefois que le cœur qui dé-alloue un objet ne soit pas celui qui l'ai alloué en premier lieu. Dans ce cas, l'objet est inséré dans une autre liste spéciale, celle des miettes. Les listes de miettes sont aussi organisées par cœur : il y a un ensemble de listes par cœur, chacune correspondant à un cœur destinataire différent. Lorsque cette liste atteint un seuil de consommation mémoire, EMA met alors en action son ramasse-miettes. Il s'agit simplement d'envoyer un événement au cœur ayant alloué ces objets (le cœur destinataire de la liste),

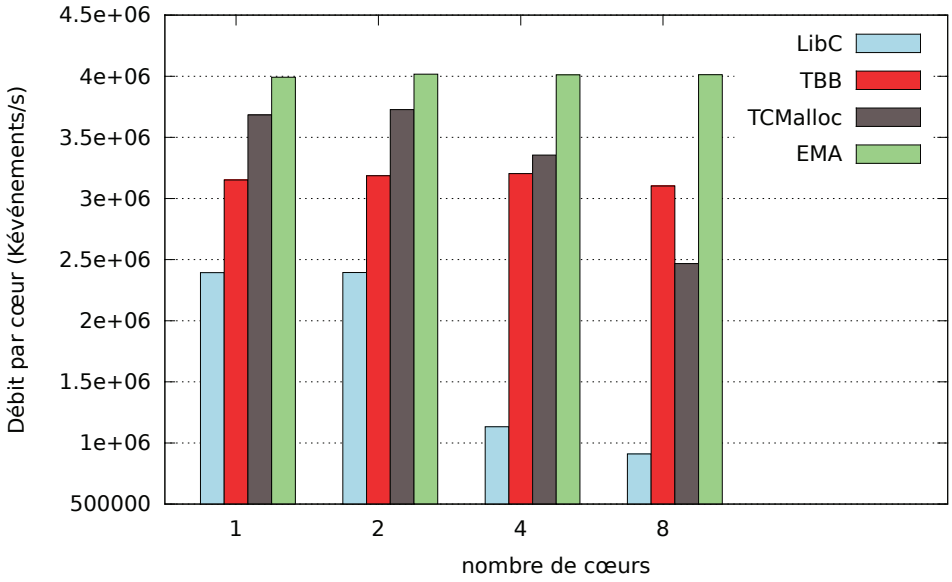

FIGURE 5.12 – Passage à l'échelle d'EMA et TBB sur *local ping pong*.

Allocateur	Temps d'allocation	Augmentation / 1 cœur
TBB	113 cycles	7%
EMA	69 cycles	0%

TABLE 5.2 – Temps d'allocation d'EMA et TBB à 8 cœurs sur `ping pong local`, avec l'augmentation correspondant au passage de 1 à 8 cœurs.

pour lui notifier de libérer les objets contenus dans notre liste. Les événements envoyés par EMA sont de haute priorité, pour éviter de gaspiller et permettre de réutiliser au plus tôt la mémoire.

Résultats. La Figure 5.12 montre le passage à l'échelle d'EMA comparé à celui de TBB sur *local ping pong*. Rappelons que TBB est le meilleur allocateur observé lors de notre précédente comparaison. On observe sur cette figure que le débit d'EMA reste constant quelque soit le nombre de cœurs utilisés. Ce débit est 30% meilleur que celui observé avec TBB. Pour corroborer ce résultat, le Tableau 5.2 présente le temps d'allocation à 8 cœurs, ainsi que l'augmentation de ce temps entre 1 et 8 cœurs. Le temps d'allocation d'EMA est 64% plus faible que celui observé avec TBB. Le fait qu'EMA n'utilise pas de primitive de synchronisation dans le cas normal, en plus d'éviter le faux-partage, lui permet d'avoir un passage à l'échelle idéal sur le test *local ping pong*.

Les résultats sur le test producteur-consommateurs sont présentés à la Figure 5.13. EMA a un débit similaire à ptmalloc à 2 cœurs sur ce test. De plus, EMA est ici le seul allocateur offrant des performances proches d'un passage à l'échelle idéal avec le nombre de cœurs.

Les Figures 5.14 et 5.15 présentent respectivement les accès en cache L1 et L2 selon l'allocateur mémoire, à 2 cœurs. On y observe qu'EMA a le même comportement en cache ici que ptmalloc. C'est à dire que contrairement à TCMalloc et TBB, EMA est capable d'éviter les situations de faux-partage, même quand les événements sont alloués par un cœur différent de celui qui les traite. Contrairement à ptmalloc cependant, l'absence de faux-partage n'est pas dûe à une coïncidence du placement mémoire, mais bien à l'architecture d'EMA. Cela

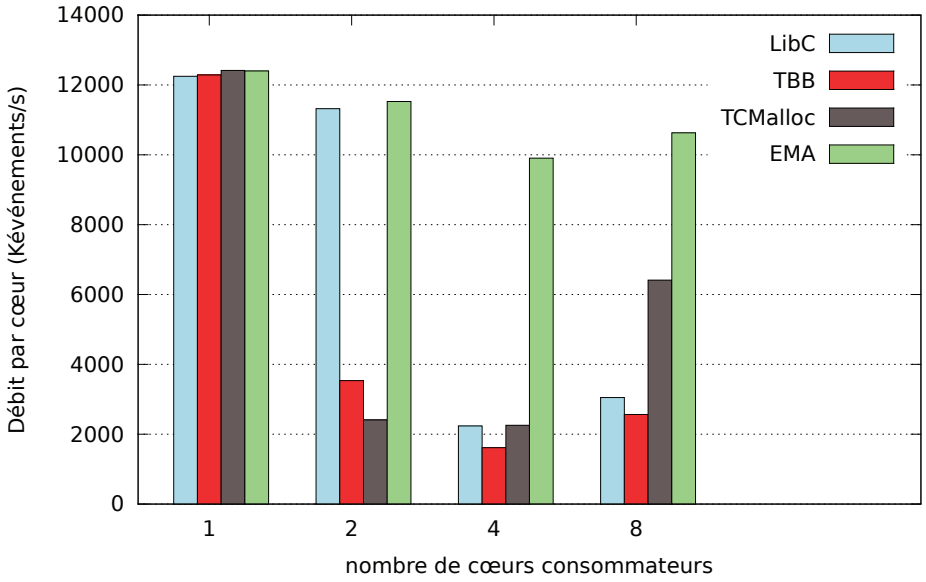

FIGURE 5.13 – Passage à l'échelle d'EMA et TBB sur l'application producteur-consommateurs.

montre que la source des performances d'EMA vient de la résolution des problèmes de faux-partage.

Nous avons donc réalisé un allocateur permettant d'éviter les situations de faux-partage observées initialement, et dont les performances restent toujours meilleures que les autres allocateurs testés dans les deux contextes étudiés ici.

5.2.4 Bilan

Dans cette section, nous avons mené une étude sur l'utilisation mémoire de notre environnement d'exécution. Cette étude part du constat qu'en l'état, Libasync-smp ne se comporte pas comme on l'attendrait. En effet, nous mettons en avant la présence de faux-partage sur un banc d'essai simple (`ping pong local`).

Pour contrer ces effets néfastes sur les performances de l'environnement d'exécution, nous réalisons tout d'abord un bourrage des structures statiques internes à Libasync-smp. Cette technique permet d'augmenter le débit de notre banc d'essai, avec notamment une multiplication par 2.6 du débit à 8 cœurs par rapport à la version originale. Cette amélioration n'est cependant pas satisfaisante, car notre application `ping pong local` n'affiche pas un passage à l'échelle idéal comme attendu. Nous nous penchons donc sur l'allocation dynamique. Comme aucun des allocateurs mémoire testés ne comble nos attentes, nous en imaginons un nouveau.

Notre allocateur mémoire EMA est étroitement lié à Libasync-smp. En effet, il tire parti des informations de l'environnement d'exécution, et notamment la coloration des événements, pour allouer la mémoire de façon à éviter tout faux-partage. La coloration des événements permet à EMA de déterminer quel sera le cœur réellement utilisateur de la zone mémoire allouée. Nos deux bancs d'essai affichent deux comportements en mémoire radicalement différents, et nous montrons qu'EMA se comporte toujours de façon satisfaisante. Plus précisément, le débit avec EMA est au moins toujours aussi bon que celui avec le meilleur allocateur (TBB sur l'application producteur-consommateurs), voire 30% meilleur que TBB sur `ping pong`

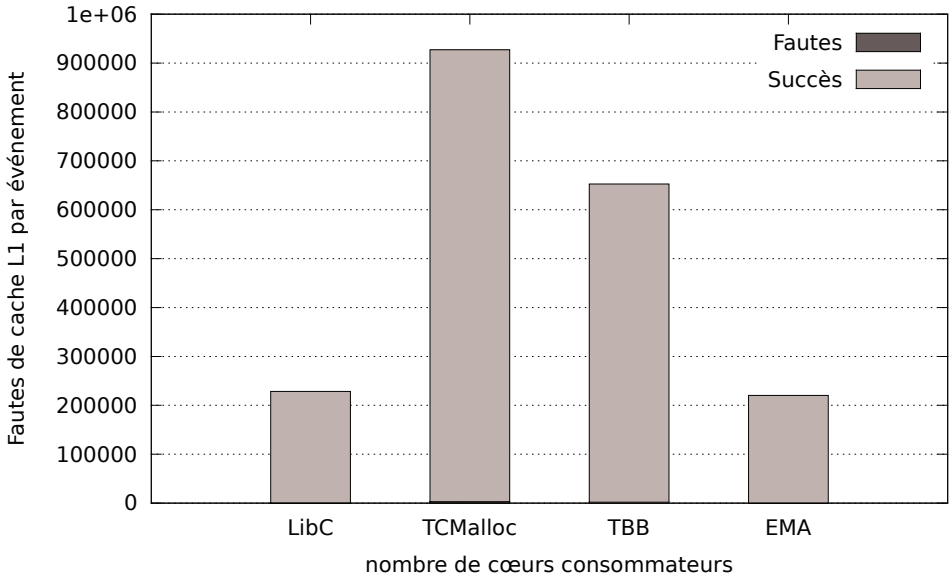

FIGURE 5.14 – Détail des accès aux caches L1 sur l'application producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire.

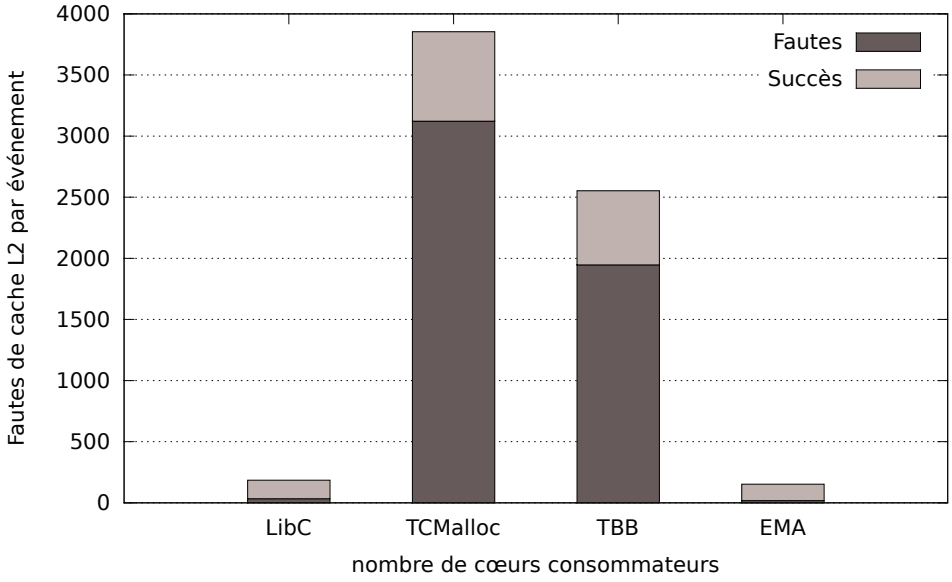

FIGURE 5.15 – Détail des accès aux caches L2 sur l'application producteur-consommateurs à 2 cœurs, selon l'allocateur mémoire.

`local`.

Nous retiendrons de ces expériences que le phénomène de faux-partage est extrêmement néfaste sur les performances des applications où il apparaît. De plus, éviter ce phénomène requiert très souvent d'augmenter la consommation mémoire de l'application : que ce soit pour mettre en place du bourrage, ou bien de la duplication de données. Il est donc essentiel de trouver un compromis viable entre une surconsommation mémoire et le faux-partage observé.

5.3 Optimisations des communications inter-cœurs

Dans cette section, nous nous intéressons aux coûts de synchronisation. Dans notre banc d'essai producteur-consommateurs, nous observons qu'à 8 cœurs, avec des traitants vides, le producteur passe 69% de son temps dans des primitives de synchronisation. À nombre égal d'événements enregistrés, le producteur passe à 1 cœur 7% de son temps dans les mêmes primitives. Ce banc d'essai expose donc un problème de passage à l'échelle des opérations de synchronisation de Libasync-smp.

Notre profilage nous permet de savoir que les primitives de synchronisation mises en cause ici portent sur deux aspects différents. Le premier aspect concerne le réveil d'autres cœurs. On rappelle que chaque cœur s'endort sur un tube UNIX lorsqu'il n'a plus d'événements à traiter. Le deuxième aspect consiste en l'acquisition de verrous. Comme notre modèle permet à un événement de couleur A d'enregistrer un nouvel événement de couleur B quelconque, il arrive qu'un cœur doivent accéder à une autre file d'événements que la sienne. Des verrous sont utilisés au sein de l'environnement d'exécution pour garder ces files d'événements cohérentes.

5.3.1 Latence du réveil

Le modèle producteur-consommateurs est très sensible à la latence du réveil des cœurs. En effet, au début d'une phase, chaque cœur consommateur est endormi. Lorsque le producteur enregistre de nouveaux événements, il doit alors réveiller chacun des consommateurs. Ensuite, ceux-ci se rendorment lorsqu'ils ont fini leurs traitements, et une nouvelle phase peut alors commencer.

Coûts des tubes

Dans la version originale de Libasync-smp, un cœur s'endort au moyen d'un appel à `poll` sur un tube UNIX qui lui est propre. Le réveil s'effectue donc par un appel à `write` d'un octet dans ce tube. Après cela, ce cœur réveillé lit tout le contenu du tube, pour assurer que le prochain appel à poll sera bloquant, puis se met à traiter les nouveaux événements de sa file.

Le seul appel à `write` dure environ 10K cycles. Le modèle de programmation événementiel encourage l'utilisation de traitants courts, pour augmenter la réactivité globale de l'application. Dans la pratique, il est possible de trouver des événements dont le traitement est bien plus court que 10K cycles. Nous avons par exemple des événements dont le traitement requiert entre 2K et 4K cycles dans un serveur Web présenté plus loin à la Section 5.4.1. Nous estimons la latence totale d'un réveil à 22K cycles. Par latence totale nous entendons le temps entre l'appel à `write` et le retour de `poll`.

Méthode de réveil	Débit (KÉvénements / s)	temps du réveil (%)	latence totale
tubes UNIX	198	73%	22K cycles
conditions pthread	229	61%	20K cycles
attente active	1850	0%	532 cycles

TABLE 5.3 – Profilage du banc d'essai producteur-consommateurs à 8 cœurs selon la méthode de réveil.

Nouvelle approche

Dans le but d'améliorer ce comportement, nous proposons une nouvelle solution pour l'endormissement et le réveil des cœurs dans Libasync-smp. Nous nous basons ici sur une technique d'attente active raisonnée. L'idée est de permettre à l'environnement d'exécution d'être plus réactif à haute charge, sans toutefois gaspiller du temps CPU inutilement.

Le Tableau 5.3 présente les résultats obtenus selon différentes méthodes de réveil sur notre banc d'essai producteur-consommateurs à 8 cœur. Nous comparons trois techniques : l'attente active, les conditions pthread et les tubes UNIX (utilisés dans la version originale de Libasync-smp).

L'attente active pure et simple signifie l'absence d'endormissement lorsqu'un cœur n'a plus d'événements à exécuter. Dans ce cas, tous les cœurs sont utilisés à 100% du temps. C'est l'approche qui donne de loin le meilleur débit (plus de 8 fois celui obtenu avec les tubes). Cependant, cette solution est mauvaise à deux points de vue. Premièrement, l'ordonnancement des autres processus sur le système se trouve impacté par le fait que les threads de l'environnement événementiel ne s'endorment jamais. Cela rend la progression des autres processus bien plus lente. Deuxièmement, les cœurs peuvent rester actifs inutilement pendant de longues périodes, ce qui va consommer bien plus d'énergie que nécessaire. Cette consommation énergétique a un fort impact économique. Elle détermine notamment le dimensionnement des *datacenters* (centres de traitement de données). En effet, il s'agit pour ces infrastructures de faire un compromis entre le nombre de cœurs de chaque machine et le nombre global de machines. Cette décision se base sur le rendement d'une machine, calculé en unité de traitements par unité d'énergie consommée. Notre objectif principal est toujours ici d'augmenter le rendement de chaque machine. Pour cela, nous augmentons la capacité de traitement de données en multi-cœur, mais il ne faut pas pour autant augmenter la consommation énergétique, sinon le rendement n'augmente pas.

Parmi les types d'attentes possibles, nous testons également les conditions fournies par la bibliothèque standard Pthread. Ces primitives de synchronisation permettent notamment à un thread de s'endormir selon une condition. Nous observons un débit légèrement meilleur (+15%) avec ce mécanisme, par rapport au débit obtenu avec les tubes UNIX présents dans la version originale de Libasync-smp. Ce gain peut s'expliquer par le fait que les conditions sont un mécanisme prévu pour gérer finement le réveil des threads. Les tubes constituent à l'origine un mécanisme de communication, que Libasync-smp utilise pour gérer le réveil. La différence de performance observée entre les conditions et les tubes provient très probablement du coût de la gestion des tampons de communication associés aux tubes.

Notre approche, baptisée `attente active raisonnée`, est un compromis entre les deux meilleures techniques observées : l'attente active et les conditions Pthread. L'idée est de ne pas endormir immédiatement un cœur lorsqu'on est à haute charge, pour permettre une meilleure réactivité dans le cas où un nouvel événement devrait arriver rapidement. Notre solution est implantée comme suit :

Initialement, chaque cœur s'endort immédiatement sur une condition lorsqu'il n'a plus d'événements à traiter. Lors de son réveil, il mesure le temps passé endormi. À partir de cette donnée, il va déterminer la meilleure politique à adopter pour la prochaine attente. Si le temps passé endormi se révèle être trop court (en dessous d'un certain seuil), alors ce cœur utilisera une attente active pendant un temps limité, avant de s'endormir sur une condition si aucun événement n'est reçu. Si le temps passé endormi est au dessus du seuil, alors l'environnement d'exécution n'est pas soumis à une haute charge, et le cœur s'endormira directement sur une condition la prochaine fois.

Cette solution est donc adaptative, dans le sens où elle détecte si la charge est suffisante pour activer une attente active ou non, en fonction du temps passé endormi. Ce temps est recalculé à chaque fois qu'un cœur doit s'endormir, ce qui permet une adaptation continuelle. Le seuil est fixé à dix fois la latence totale des conditions pthread, soit 200K cycles. C'est à dire le temps écoulé entre l'appel à `pthread_cond_signal` et le retour de `pthread_cond_wait`. Le temps d'attente active est limité à ce même seuil.

Nous avons effectué ce choix selon l'idée suivante. Lorsqu'un cœur est en attente sur une condition pthread, et que cette attente est plus courte que notre seuil, cela veut dire que plus de 10% du temps d'attente était incompressible (du fait de la latence des conditions). Nous pensons qu'à partir du moment où nous pouvons réduire le temps de réveil de plus de 10%, alors il devient rentable de le faire. Le cœur dans cette situation choisira donc une attente active raisonnée plutôt que la condition pthread lors de son prochain endormissement. Une piste d'amélioration lors de travaux futurs est d'adapter la valeur de ce seuil en fonction de l'application.

Pour mettre en évidence les performances des différentes techniques, nous modifions notre banc d'essai producteur-consommateurs pour contrôler la durée des traitements (les événements produits ne sont plus nécessairement vides). La Figure 5.16 présente les résultats obtenus selon différents types d'attente. On y observe notamment que les coûts liés au réveil des threads sont plus présents pour les traitants courts. Le temps de réveil devient négligeable lorsque les traitants exécutés durent plus de 10^5 cycles. La durée d'un traitant d'un serveur de données réaliste se situe généralement entre 10^3 et 10^4 cycles. Les coûts de réveils ont donc un impact sur ces applications, comme notre évaluation le confirme à la section 5.4.

Notre solution a des performances comparables à celle de l'attente active classique (-3% dans le pire cas). Elle permet toutefois aux autres processus de progresser normalement car les cœurs s'endorment lorsqu'ils n'ont plus d'événements à traiter. L'endormissement permet également de gagner sur des aspects d'économie d'énergie.

5.3.2 Enregistrement d'événements par lots

Avec les solutions précédentes, le débit de traitement des événements est fortement augmenté, et on observe une augmentation de la contention sur les files d'événements. Sur le banc d'essai producteur-consommateurs, le temps passé dans l'acquisition de verrous est passé de 10% à 17%. L'augmentation de ces coûts s'explique par le fait que les événements sont enregistrés individuellement. Le producteur va donc payer le coût de synchronisation pour chaque événement enregistré. De plus, l'entrelacement des enregistrements vers des cœurs différents a tendance à générer des fautes de cache.

Pour amortir les coûts de verrouillage et améliorer la localité de cache, nous mettons en place un enregistrement d'événements par lots (batch). L'idée est d'enregistrer plusieurs événements en une seule prise de verrou. Chaque cœur se voit donc attribué un tampon d'événements sortants par cœur distant. Lorsqu'un cœur veut enregistrer un événement sur

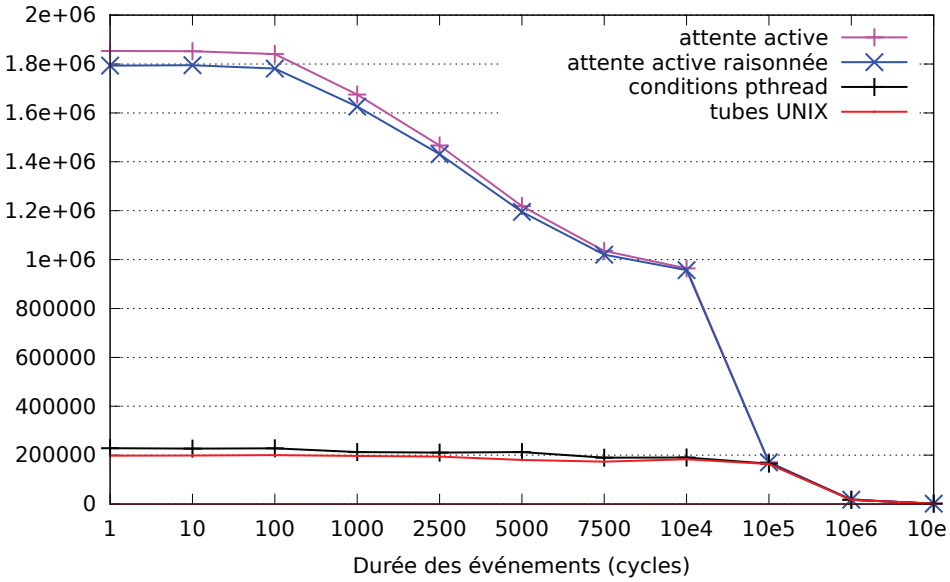

FIGURE 5.16 – Débit du banc d'essai producteur-consommateurs en fonction de la durée des traitants.

un cœur distant, il le place dans le tampon correspondant. Lorsque le tampon est plein, le cœur acquiert le verrou du cœur distant et enregistre tous les événements de son tampon, puis relâche le verrou.

Deux considérations sont à prendre en compte. Premièrement, il est nécessaire de forcer périodiquement l'enregistrement des événements contenus dans un tampon. Il est autrement impossible de garantir le bon progrès de l'application. Pour déterminer cette période, nous observons régulièrement le temps moyen entre deux vidages "naturels" du tampon. Comme cette période est une sécurité pour prévenir la famine, il faut éviter de déclencher des vidages prématurés, pour ne pas réduire l'efficacité globale de notre mécanisme. Nous fixons donc la période à une valeur légèrement supérieure à la moyenne observée, soit 110% du temps moyen entre deux vidages. Deuxièmement, il faut fixer la taille du tampon de manière à trouver un compromis entre maintenir une charge suffisante sur le cœur distant, et amortir les coûts de verrouillage. Il faut pour cela tenir compte du temps de traitement des événements enregistrés, du temps nécessaire pour acquérir le verrou, et de la fréquence des enregistrements d'événements sur des cœurs distants. Ces paramètres sont dépendants de l'application considérée, aussi nous fixons la taille du tampon en fonction de l'application, de façon à toujours maximiser les performances. Nous prenons ainsi une taille de 50 pour le producteur-consommateurs, et une taille de 20 pour l'évaluation sur notre serveur Web et MapReduce.

La Figure 5.17 présente le débit du banc d'essai producteur-consommateurs à 8 cœurs, avec et sans enregistrement par lots. On observe que l'enregistrement par lots peut aller jusqu'à doubler les performances de notre banc d'essai. Les Tableaux 5.4 et 5.5 permettent d'expliquer la provenance de ces gains. Ils présentent les gains apportés par l'enregistrement par lots sur les coûts de verrouillage et l'efficacité des caches L2, respectivement pour le producteur et les consommateurs. On observe une diminution par un facteur 2.6 du temps passé à acquérir des verrous sur le producteur. Plus généralement le nombre d'accès aux caches L2 est divisé par 3, et le nombre de fautes de caches en L2 se voit divisé par au moins 3.5. L'enregistrement par lots augmente donc significativement la localité en cache.

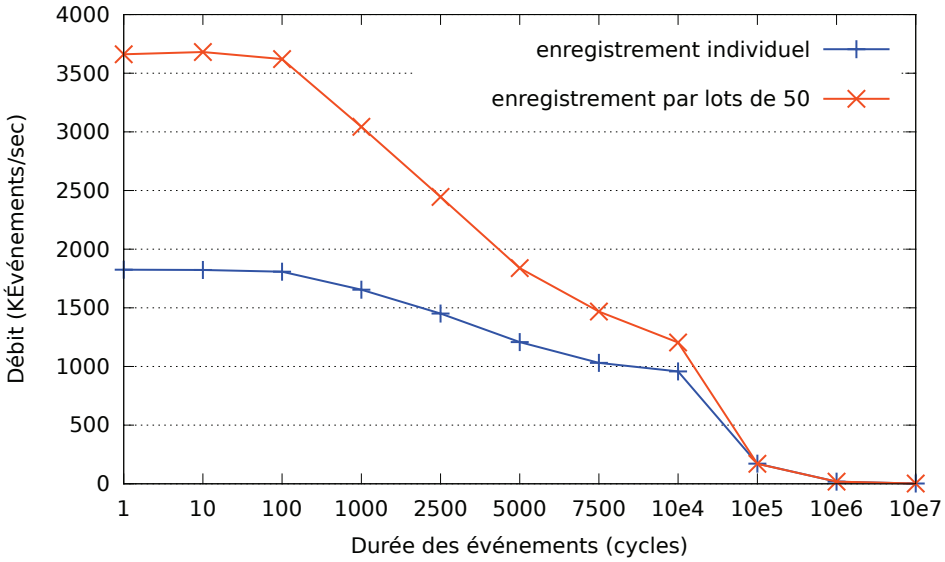

FIGURE 5.17 – Débit du banc d'essai producteur-consommateurs à 8 cœurs avec et sans enregistrement d'événements par lots.

	Producteur		
	coûts de verrouillage	accès L2 / événement	fautes L2 / événement
Sans lots	17.2 %	89	41
Avec lots	6.6 %	28	11

TABLE 5.4 – Effets de l'enregistrement par lots sur les coûts de verrouillage et les caches à 8 cœurs, sur le cœur producteur.

	Consommateurs		
	coûts de verrouillage	accès L2 / événement	fautes L2 / événement
Sans lots	2.3 %	10	7
Avec lots	1.9 %	3	2

TABLE 5.5 – Effets de l'enregistrement par lots sur les coûts de verrouillage et les caches à 8 cœurs, sur les cœurs Consommateurs.

Notre enregistrement par lots permet donc de rentabiliser les coûts de verrouillages. Nous nous assurons toutefois de ne pas perdre la correction de l'application en forçant un vidage périodique des tampons au besoin. Déterminer la valeur optimale de la taille des tampons demeure une des perspectives de ces travaux. Nous souhaitons trouver une solution viable et adaptative en ligne pour ce calcul.

5.3.3 Bilan

Dans cette section, nous avons étudié les coûts de synchronisation au sein de notre environnement d'exécution. Notre banc d'essai producteur-consommateurs pointe vers deux aspects demandant des optimisations potentielles : les primitives de réveil et celles de synchronisation.

Nous avons tout d'abord montré l'impact sur les performances des primitives de réveil sur Libasync-smp. Nous proposons une nouvelle approche baptisée `attente active raisonnée`. Cette approche est un compromis entre de l'attente active pure et l'utilisation de primitives d'endormissement classiques. Elle donne une réactivité permettant de ne pas dégrader les performances à haute charge, tout en gardant un mécanisme d'endormissement pour permettre un ordonnancement équitable vis à vis des autres processus.

Nous avons deuxièmement étudié les coûts des primitives de synchronisation. Notre banc d'essai producteur-consommateurs montre une forte augmentation du temps passé dans ces primitives lorsqu'on passe de 1 à 8 cœurs (de 7% à 69%). Pour pallier ce problème, nous adoptons une approche de traitement par lots. C'est à dire qu'un cœur n'enregistre plus que des lots d'événements, et évite d'enregistrer des événements seuls. Cela permet d'amortir les coûts de synchronisation, d'augmenter la localité en cache de l'environnement d'exécution, et ainsi d'aller jusqu'à doubler le débit de notre producteur-consommateurs à 8 cœurs.

Deux questions restent ouvertes suite à ces travaux. La première est de décider comment choisir une valeur de seuil optimale pour la durée d'attente active raisonnée. Nous pensons qu'étudier des approches à base d'apprentissage par la machine (machine learning) pourraient permettre de converger vers une solution optimale liée à une application. En effet, cela devrait permettre de reconnaître des motifs dans les délais d'arrivée des événements, et donc de mieux cerner quelle technique d'endormissement adopter. Ce type d'approche est sûrement coûteux et il sera difficile de la rendre efficace pour un calcul en ligne de la valeur de seuil. Nous pensons que ce type de calcul doit être fait en ligne. En effet, dans le domaine des serveurs de données il est très souvent impossible de prédire les changements éventuels de charge entrante, et une analyse statique ne permet donc pas d'en tenir compte. Une telle approche pourrait donc constituer une base pour mettre en œuvre des algorithmes d'approximation de la valeur optimale qui soient suffisamment simples pour être implantés en ligne. Deuxièmement, nous espérons trouver un moyen fiable pour déterminer la taille optimale des lots d'événements à enregistrer. Nous pensons qu'un profilage fin du temps de traitement de chaque événement, ainsi que des temps d'acquisition des verrous, couplés aux motifs des délais d'arrivée des événements, devraient permettre de faire un tel calcul. Sachant que ces données recouvrent l'ensemble des facteurs influant sur cette taille, une modélisation de la taille optimale devrait être possible à partir d'exécutions de différentes applications simples.

5.4 Évaluation sur des applications réelles

Jusqu'ici nous avons validé nos optimisations sur nos deux bancs d'essai. Nous évaluons dans cette section nos optimisations sur deux cas d'études réels. Nous commençons par détailler le contexte expérimental. Nous commençons notre analyse avec un serveur Web. Puis

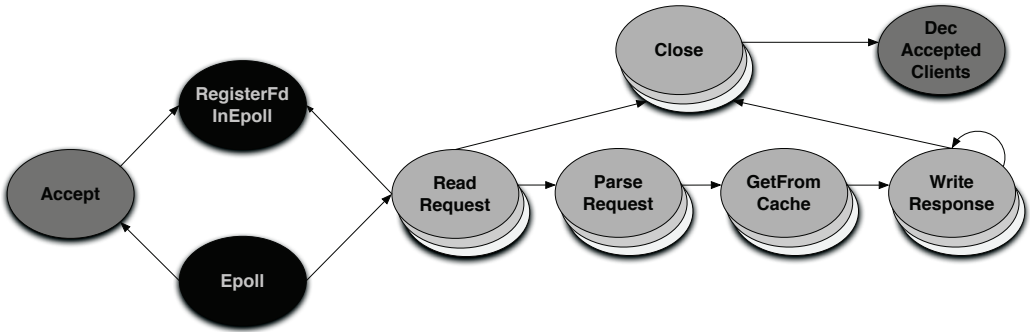

FIGURE 5.18 – Architecture de SWS.

nous évaluons nos optimisations sur la suite de tests de l'environnement d'exécution MapReduce Phœnix.

De façon intéressante, ces deux types d'applications exposent des profils d'exécution très différents. En effet, le serveur Web est une application intensive en termes d'E/S. De plus, il utilise un grand nombre d'événements qui ont un faible temps d'exécution. D'un autre côté, les applications de la suite Phœnix n'utilisent pas ou très peu d'E/S, et leur création d'événements est plus irrégulière que celle observée sur le serveur Web.

Nous présentons les performances de la version originale de Libasync-smp d'une part, puis celles de notre environnement d'exécution modifié, en rajoutant les optimisations les unes après les autres. Cela nous permet de connaître la part de gain liée à chaque optimisation.

5.4.1 Serveur Web

Nous évaluons un serveur Web événementiel que nous avons réalisé sur l'environnement d'exécution Libasync-smp. Nous avons baptisé cette implantation `SWS` (pour `Simple Web Server`).

Architecture de SWS. Pour la mise en œuvre de SWS, nous avons suivi l'architecture décrite dans le papier de Libasync-smp [93] pour servir du contenu statique. Notre serveur est capable de servir du contenu statique via la méthode GET, et de traiter correctement les différents cas d'erreurs.

La Figure 5.18 présente l'architecture de SWS. On peut y voir les différents traitants d'événements utilisés, ainsi qu'un exemple représentatif de leur coloration. Par souci de simplicité, les traitants d'erreurs ne sont pas représentés. Nous détaillons plus avant chaque traitant :

- Le traitant `Epoll` est en charge de surveiller les descripteurs de fichiers enregistrés. Ce traitant fait partie de l'environnement d'exécution, et est donc caché du programmeur. Dans le cas de SWS, ce traitant surveille l'arrivée de nouvelles connexions, ainsi que l'arrivée de nouvelles requêtes.
- Le traitant `RegisterFdInEpoll` correspond à la fonction `fdcb` de l'API de Libasync-smp. Il fait donc lui aussi partie du fonctionnement interne de l'environnement d'exécution. Il est appelé lorsqu'une application veut enregistrer de nouveaux descripteurs de fichiers à surveiller par `Epoll`. `RegisterFdInEpoll` et `Epoll` sont tous deux coloriés avec la même couleur, car ils travaillent tous deux sur les mêmes données, notamment le tableau des descripteurs de fichiers surveillés.
- Le traitant `Accept` est en charge d'accepter les nouvelles connexions entrantes. Chaque nouvelle connexion ainsi créée est suivie d'un événement de type `RegisterFdInEpoll`,

pour guetter les requêtes entrantes. Comme il est intéressant de connaître le nombre de connexions en traitement, ce traitant met à jour un compteur des connexions actives. Cela sert notamment pour ne pas dépasser le nombre maximal de clients à servir simultanément, pour garantir des propriétés de qualité de service (comme le temps de réponse, par exemple). Tous les traitants manipulant ce compteur doivent être exécutés en exclusion mutuelle, pour maintenir la cohérence de ce compteur, et doivent donc avoir la même couleur.

- `ReadRequest` est le traitant associé à la lecture réseau d'une requête. Ce traitant est déclenché par `Epoll` lorsque de l'activité est détectée sur une connexion. Une requête peut être lue en plusieurs fois, donc `ReadRequest` se réenregistre sur `Epoll` via `RegisterFdInEpoll`. Lorsqu'une requête est entièrement lue, `ReadRequest` poste un événement `ParseRequest`. Si au contraire une fermeture de connexion est détectée, alors un événement `Close` est posté. `ReadRequest` est coloré avec le numéro de connexion (numéro de descripteur de socket), pour permettre un parallélisme par flots (par connexions). Cette coloration est conservée pour tous les événements survenant sur la même connexion : `ParseRequest`, `GetFromCache`, `WriteResponse`, et `Close`.
- Le traitant `ParseRequest` est en charge d'analyser syntaxiquement la requête lue. Dans le cas classique (sans erreur) il poste un événement de type `GetFromCache` avec l'URI [3] correspondante.
- `GetFromCache` récupère la ressource demandée dans la requête. Il poste ensuite un événement `WriteResponse` lorsque cette ressource demandée est récupérée.
- Le traitant `WriteResponse` est en charge d'écrire la réponse sur la socket. Cela peut éventuellement s'effectuer en plusieurs fois, si les tampons système d'écriture sont pleins. Pour cela, nous pouvons utiliser `Epoll` pour re-poster `WriteResponse` lorsque les tampons d'écritures sont libérés. Cependant, cela implique des changements de cœurs, coûteux, dûs à la coloration. Comme la saturation des tampons d'écriture est un phénomène que nous observons rarement, nous préférons re-poster `WriteResponse` directement, par souci d'efficacité. Ensuite, lorsque l'intégralité de la réponse à été écrite, ce traitant peut enregistrer un événement `Close` si le serveur choisit de fermer la connexion.
- `Close`, comme son nom l'indique, sert à fermer la connexion pour laquelle il est posté. Nous l'utilisons également pour libérer les tampons associés. Il faut cependant prendre garde à s'assurer qu'il n'y ait pas d'autres traitements en cours sur celle-ci. C'est à dire qu'aucune autre requête ne doit être arrivée entre-temps sur la connexion à fermer. Lorsqu'une connexion est fermée, ce traitant enregistre un événement `DecAcceptedClients`.
- Le traitant `DecAcceptedClients` sert à décrémenter le compteur des connexions simultanées. Il est donc coloré de la même couleur qu'`Accept`, qui est chargé d'incrémenter le même compteur. Cette décrémentation peut mener à l'acceptation de nouvelles connexions.

Injection de charge. Pour nos expérimentations sur SWS, nous avons développé un injecteur de charge. Son architecture est basée sur celle présentée par Banga et Druschel [4]. Il est constitué de deux threads, fonctionnant chacun de manière événementielle. Le premier est dédié à l'envoi de requêtes, tandis que le deuxième réceptionne les réponses. Nous adoptons un système maître-esclave pour distribuer l'injection sur plusieurs machines. Le maître est donc en charge de synchroniser les esclaves lors des différentes phases d'injection, ainsi que d'agréger leurs résultats. L'injection se fait en boucle fermée : (i) pour chaque phase d'in-

3. Universal Resource Identifier

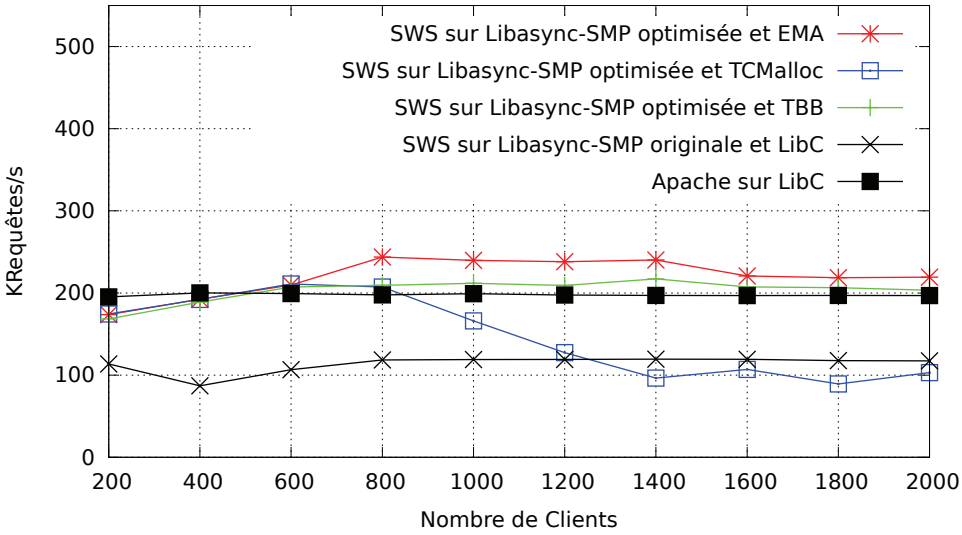

FIGURE 5.19 – Débit de SWS avec différents allocateurs, comparé à Apache

jection, un esclave est responsable d'un certain nombre fixe de clients et (ii) lorsqu'un client envoie une requête, il attend de recevoir la réponse avant de renvoyer une requête.

Notre charge a plus pour but de stresser l'environnement d'exécution que d'exposer un fort degré de réalisme. Pour cela, nous utilisons de petits fichiers de 1Ko. Cela permet notamment de s'affranchir des E/S disque. De plus, la petite taille de fichier lui permet de tenir en cache. Ainsi, la majorité des accès mémoires proviennent de l'environnement d'exécution. 8 machines double-cœurs sont en charge d'injecter la charge sur les 8 cartes réseau de notre serveur. Nous faisons varier le nombre de clients de 200 à 2000. Chaque client virtuel ouvre une connexion, envoie 150 requêtes, et ferme la connexion. Ce comportement est répété pendant toute la phase d'injection. Chaque phase dure 30 secondes et est répétée 3 fois. Nous observons un écart-type toujours inférieur à 2%.

Évaluation. La Figure 5.19 montre le débit à 8 cœurs de SWS et du serveur Web Apache en fonction du nombre de clients, selon différentes configurations. Le serveur Web Apache est testé avec la LibC pour refléter une configuration classique. On observe que le meilleur débit est atteint par notre version optimisée de Libasync-smp combinée à l'allocateur mémoire EMA. Par version optimisée, on entend ici version avec les optimisations suivantes activées : le bourrage statique, l'attente active raisonnée, et l'enregistrement par lots. Cette configuration, ainsi que Libasync-smp optimisée combinée à l'allocateur mémoire de TBB ont toutes deux de meilleures performances qu'Apache, qui nous sert ici de référence. La version originale de Libasync-smp, combinée à l'allocateur de la LibC, obtient les moins bonnes performances. Cela reflète les défauts de mise en œuvre constatés à la Section 5.1.4. Le fait que notre version optimisée avec EMA affiche le meilleur débit montre que nous avons correctement pallié ces défauts.

De façon intéressante, le débit de SWS avec TCMalloc s'effondre à partir de 800 clients. Notre hypothèse est que des coûts de verrouillage apparaissent liés à la gestion des les listes internes de TCMalloc. En effet, nous avions déjà remarqué à la section 5.2.2 que le passage à l'échelle avec le nombre de cœurs de TCMalloc était moins bon que celui de TBB sur nos

Optimisation	**Amélioration par rapport à l'original**	**Amélioration par rapport à la version précédente**
+ bourrage statique	17%	-
+ EMA	55%	32%
+ attente active raisonnée	70%	10%
+ enregistrement par lots	111%	24%

TABLE 5.6 – Détail de l'amélioration du débit relativement à chaque optimisation sur SWS. Chaque optimisation est rajoutée à la précédente (la dernière ligne présente la version finale avec toutes les optimisations).

bancs d'essais.

Le tableau 5.6 présente l'impact de chaque optimisation sur le débit de SWS à 8 cœurs. Nous partons de la version originale de Libasync-smp, et nous rajoutons chaque optimisation à la version de la ligne précédente. On observe tout d'abord que chaque optimisation a un rôle important dans le débit de SWS, et fait gagner entre 10% et 32% de performances.

L'amélioration des placements mémoire augmente à elle seule le débit de SWS de 55% par rapport à la version d'origine de Libasync-smp. Ce gain significatif s'explique par le fait que SWS fonctionne avec un parallélisme par flots. C'est à dire que chaque client est traité de manière indépendante. Ces traitements ne partagent pas naturellement de données entre eux. Ici la version originale de Libasync-smp introduit du partage de données inutile, sous la forme de faux-partage au niveau des structures internes de l'environnement d'exécution. Le fait d'enlever ce faux-partage permet aux requêtes des différents clients d'être traitées de façon réellement indépendante, et donc d'augmenter significativement les performances. Nos optimisations permettent de réduire les fautes de caches L2 par événement de 63%.

Enfin, la réduction des coûts des primitives de communications inter-cœurs a également un fort impact sur les performances de SWS. Cela s'explique notamment par le fait que son architecture suit le modèle producteur-consommateurs. En effet, le traitant `Epoll` a ici un rôle de producteur, autant pour `Accept` que `ReadRequest`. C'est ce dernier qui 'consomme' les requêtes entrantes. Nos optimisations ont donc du sens dans un tel contexte, et on observe une baisse du temps passé dans des primitives de verrouillage de 81% sur le cœur producteur.

5.4.2 MapReduce

Nous évaluons maintenant nos optimisations de Libasync-smp au travers de la suite de bancs d'essais de l'environnement d'exécution MapReduce Phœnix 2.0 [67].

Présentation de Phœnix. Phœnix est un environnement d'exécution MapReduce écrit en C. Il vise à exécuter efficacement des applications écrites selon le modèle MapReduce sur des architectures multi-cœur. Pour cela, Phœnix utilise un pool de threads, des files pour gérer les tâches, ainsi que des tampons pour stocker les données de l'application et les résultats intermédiaires entre les étapes de Map et de Reduce. Les opérations sur les files de gestion des tâches sont atomiques, et impliquent la prise de verrous. L'implantation de Phœnix testée créée d'abord toutes les tâches avant de commencer leur traitement effectif.

Nous modifions cette implantation pour qu'elle tire parti de Libasync-smp. C'est à dire que nous supprimons le pool de threads et les files de gestion de tâches. À la place, nous

utilisons le modèle événementiel et les files optimisées de Libasync-smp. Les tampons intermédiaires, alloués dans des structures passées en paramètres des threads par Phœnix, sont donc maintenant stockés dans les paramètres passés aux traitants. Nous allouons ces tampons avec EMA pour les tests mentionnant qu'EMA est actif (en plus des événements Libasync-smp comme précédemment). De plus, nous n'avons plus la restriction de devoir attendre d'avoir une file de tâches pleine pour commencer le traitement des tâches. Cela dit, l'interface de programmation de Phœnix reste inchangée, et nous pouvons lancer les bancs d'essai sans les modifier. Nos seules modifications portent sur la gestion de l'exécution des tâches. Cela signifie notamment que nous ne modifions pas la façon de Phœnix de découper les tâches pour les différentes phases de calcul (Map, Reduce, et Merge).

La suite d'applications de Phœnix [67] comporte des applications provenant de domaines différents. On notera que `Word Count`, `Reverse Index` et `String Match` sont représentatives des applications d'entreprise. `Matrix Multiply` est directement issu du domaine du calcul scientifique. `Kmeans`, `PCA` et `Linear Regression` représentent des patrons de charge classique dans le domaine intelligence artificielle, tandis que `Histogram` représente le domaine du traitement d'images. Nous détaillons plus avant l'architecture de chaque application :

- **Word Count** compte le nombre d'occurrences de chaque mot d'un ensemble de fichiers d'entrée. La phase de Map parcours une sous-partie d'un fichier et renvoie une paire contenant le mot et la valeur 1 quand le mot a été trouvé. La phase Reduce va additionner les valeurs pour chaque mot.
- **Reverse Index** construit un index d'un ensemble de fichiers HTML, associant les liens vers les fichiers. L'étape de Map analyse la collection de fichiers, et renvoie une paire <lien, fichier> pour chaque lien trouvé. L'étape de Reduce fusionne tous les fichiers référencés par le même lien dans une liste chaînée.
- **String Match** travaille sur deux fichiers : un fichier contenant des mots cryptés et l'autre des mots en clair (clés). Chaque tâche de l'étape Map analyse une partie du fichier des clés, tente de les crypter, et renvoie une paire <clé, drapeau>, où le drapeau indique si la clé correspond à un mot crypté. L'étape de Reduce est la fonction identité (c'est à dire qu'elle ne modifie pas les résultats).
- **Matrix Multiply** calcule un produit de matrices. La phase de Map découpe la matrice résultat en lignes, et chaque tâche renvoie une paire $< (x, y), V >$, où (x,y) sont les coordonnées dans la matrice résultat, et V la valeur résultat. Ici également, la phase de Reduce est la fonction identité.
- **KMeans** est une implantation de l'algorithme kmeans, qui regroupe un ensemble de points en amas. Le calcul se fait en plusieurs itérations de calculs MapReduce. A chaque itération, la phase de Map explore le vecteur des moyennes et un sous-ensemble de points. A partir de ça, elle chaque tâche calcule la distance entre un point et chaque moyenne, et assigne le point à l'amas le plus proche. Pour chaque point, elle renvoie une paire contenant le numéro d'amas et le vecteur de données. La phase de Reduce rassemble tous les points d'un même amas et calcule leur barycentre. Elle renvoie donc la paire contenant le numéro d'amas et le barycentre associé. L'algorithme s'arrête lorsque les barycentres convergent.
- **PCA** calcule le vecteur moyen et la matrice de covariance d'une matrice de points. L'algorithme utilise deux itérations de MapReduce. Dans la première itération, chaque tâche Map calcule la moyenne sur une ligne et renvoie une paire associant les numéros de lignes aux valeurs moyennes. Le calcul de la matrice de covariance se fait lors de la seconde itération. L'étape Map fait ce calcul et renvoie une paire associant les coordonnées dans la matrice de covariance à la valeur contenue à cet endroit. Les phases de

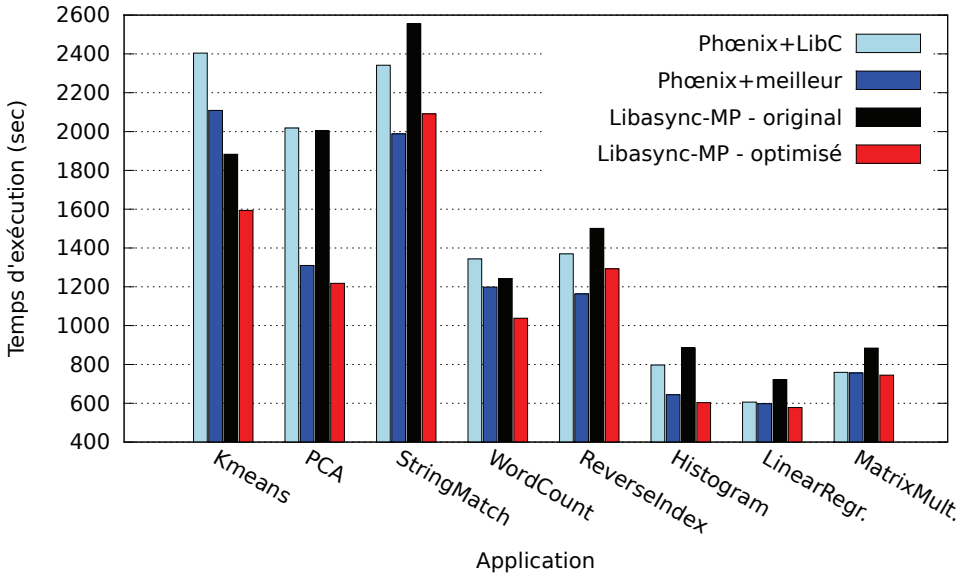

FIGURE 5.20 – Temps d'exécution des différentes applications avec différentes versions de Phœnix. Le temps le plus court représente les meilleures performances. Phœnix+meilleur correspond aux performances de Phœnix lié au meilleur allocateur mémoire (TCMalloc ou TBB).

Reduce sont la fonction identité dans les deux itérations.

- **Linear Regression** calcule la droite correspondante à un ensemble de points situés dans un fichier. L'étape de Map calcule différentes métriques, notamment la somme des carrés, sur un sous-ensemble du fichier. L'étape de Reduce fusionne ces résultats sur l'ensemble du fichier et calcule la meilleure droite.
- **Histogram** calcule la fréquence d'occurrence des couleurs dans une image bitmap RGB 256 couleurs. Chaque tâche Map est associée à une partie du fichier, et calcule la fréquence des couleurs dans leur partie, puis renvoie un tableau des fréquences. La phase de Reduce fait la somme de ces fréquences sur l'ensemble du fichier.

Évaluation. Pour chaque application de la suite de tests, nous évaluons son temps d'exécution à 8 cœurs. Nous évaluons ces temps avec notre version optimisée de l'environnement d'exécution, que nous comparons à la version originale de Libasync-smp. Par souci de comparaison, nous testons également la version originale de Phœnix, avec l'allocateur standard de la LibC, ainsi que les allocateurs multi-cœur étudiés en Section 5.2.2. Chaque expérience est répétée au moins trois fois, l'écart-type ne dépasse jamais 6% de la valeur moyenne présentée.

La Figure 5.20 présente les temps d'exécution observés sur les différentes applications de la suite Phœnix. Quatre configurations sont représentées. La première, nommée "Phœnix+LibC", montre les performances de Phœnix avec l'allocateur standard de la LibC. La deuxième, "Phœnix+meilleur" correspond aux performances de Phœnix avec l'allocateur multi-cœur donnant les meilleures performances pour l'application (TCMalloc ou TBB selon les cas, le détail se trouve au Tableau 5.8). Viennent ensuite la version originale de Libasync-smp puis notre version optimisée.

Le Tableau 5.7 présente le détail des gains apportés par chaque optimisation par rapport à la version originale de Libasync-smp. Les optimisations sont rajoutées les unes aux autres,

	Word Count	Reverse Index	Histo-gram	Linear regr.	Kmeans	PCA	String Match	Matrix Mult.
+ Bourrage	1%	-2%	-2%	-1%	1%	1%	0%	-1%
+ EMA	20% (19%)	14% (16%)	24% (27%)	-1% (0%)	14% (13%)	65% (63%)	13% (13%)	-6% (-5%)
+ Attente active raisonnée	19% (-1%)	16% (2%)	46% (18%)	24% (25%)	18% (4%)	64% (0%)	24% (10%)	18% (25%)
+ Enregistrement par lots	20% (1%)	16% (0%)	47% (1%)	25% (1%)	19% (1%)	64% (0%)	22% (-1%)	19% (0%)

TABLE 5.7 – Amélioration par rapport à la version originale de Libasync-smp. Le gain de chaque optimisation est entre parenthèses. Les optimisations sont rajoutées les unes aux autres.

et la dernière ligne représente notre environnement d'exécution avec toutes les optimisations activées. Le chiffre entre parenthèses correspond à l'amélioration du temps d'exécution par l'optimisation elle-même (par rapport à la ligne du dessus), l'autre chiffre étant l'amélioration totale par rapport à la version originale de Libasync-smp. Contrairement au serveur Web, on observe ici que les gains viennent essentiellement d'EMA et de l'attente active raisonnée ; le bourrage des données statiques et l'enregistrement par lots ne jouant qu'un rôle minime dans les performances sur ces applications.

Le rôle d'EMA dans l'amélioration des performances vient surtout de sa vitesse d'allocation, qui lui confère des temps de réponse intéressants, en plus de la localité apportée par les placements mémoire. En effet, l'environnement d'exécution Phœnix fait beaucoup d'allocations, donc réduire le temps d'allocation est important. De plus, ces allocations (concernant souvent des objets itérateurs) sont accédées intensément (en boucle, typiquement). Une bonne localité des données en cache peut donc permettre un certain gain de performance dans ce cas. Un profilage nous montre que l'utilisation d'EMA réduit de 37% en moyenne les fautes de caches L2 sur l'ensemble de la suite de tests.

L'attente active raisonnée, quant à elle, a un rôle important sur le temps de réactivité des threads. Comme les applications de type MapReduce utilisent un schéma producteur-consommateurs, il est important que les consommateurs gardent une bonne réactivité lors de l'initialisation et des changements de phases (Map vers Reduce, et Reduce vers Merge). Nous observons une réduction de 38% du temps passé en attente de tâches par chaque thread.

Le bourrage, ainsi que l'enregistrement par lots, s'ils ne sont pas utiles pour les applications MapReduce présentées ici, ne heurtent pas les performances pour autant. Nous avons vu qu'ils ont leur utilité sur d'autres types d'applications, comme le serveur Web. Nos optimisations apportent globalement de 16% à 66% de gain sur les applications de type MapReduce par rapport à la version originale de Libasync-smp.

Le Tableau 5.8 présente les améliorations de performance de notre environnement d'exécution optimisé par rapport à la version originale de Phœnix (sans Libasync-smp). Par souci d'équité, nous comparons Phœnix avec l'allocateur standard de la LibC (1ère ligne), ainsi qu'avec les autres allocateurs multi-cœur étudiés précédemment. La 2ème ligne du tableau représente l'amélioration de notre environnement d'exécution par rapport à Phœnix avec le meilleur allocateur pour le test (dont le nom est entre parenthèses).

On observe tout d'abord que notre environnement d'exécution est toujours meilleure que Phœnix sur l'allocateur de la LibC (ptmalloc), avec un gain de 2% à 66%. Cela montre que notre environnement d'exécution est a priori meilleur que l'existant dans une configuration

	Word Count	Reverse Index	Histo-gram	Linear regr.	Kmeans	PCA	String Match	Matrix Mult.
Amélioration sur Phœnix avec LibC	29%	6%	32%	5%	51%	66%	12%	2%
Amélioration sur Phœnix avec un allocateur multi-cœur	15% (TBB)	-10% (TBB)	7% (TBB)	3% (TBB)	32% (TC-Mal-loc)	8% (TBB)	-5% (TC-Mal-loc)	2% (TBB)

TABLE 5.8 – Amélioration de notre environnement d'exécution par rapport à Phœnix 2.0.

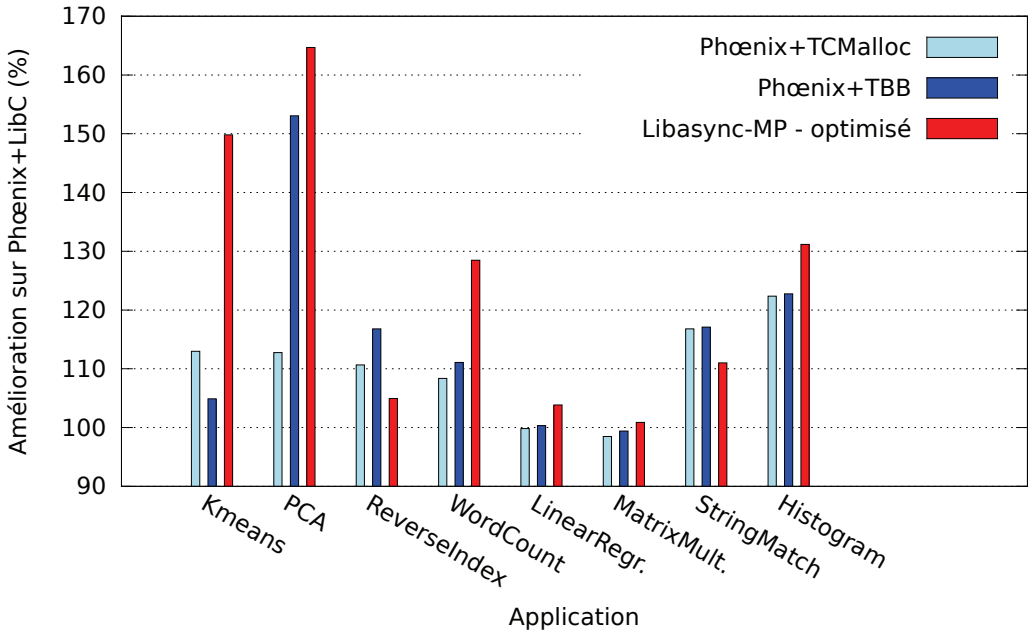

FIGURE 5.21 – Améliorations des différents allocateurs par rapport à Phœnix sur LibC (ptmalloc).

standard. De plus, notre environnement d'exécution a des performances le plus souvent comparables à celles de Phœnix utilisant un allocateur multi-cœur (de -10% à 15%), sauf pour Kmeans, pour lequel notre environnement d'exécution augmente les performances de 32%.

Nous avons vu qu'EMA joue un rôle clé dans les gains qu'apportent nos optimisations (cf. Tableau 5.7). Partant de cette observation, nous voulons maintenant comparer les performances des différents allocateurs mémoire sur la suite d'applications MapReduce de Phœnix. La Figure 5.21 présente l'amélioration des performances de Phœnix, successivement avec TC-Malloc puis TBB, et notre environnement d'exécution (avec EMA), par rapport à Phœnix avec la LibC (ptmalloc). Cela nous permet de comparer les comportements des différents allocateurs mémoire multi-cœur étudiés.

Il ressort principalement deux remarques de la Figure 5.21. Tout d'abord, pour les applications Reverse Index, Kmeans et PCA, on observe des variations de performances avec les différents allocateurs multi-cœur significatives (supérieures à l'erreur de mesure). C'est no-

tamment intéressant dans le cas de PCA, où les performances avec TBB sont 35% supérieures à celles avec TCMalloc. Kmeans présente la tendance inverse, et TCMalloc devient meilleur que TBB. Notre argument ici est que dans les cas où l'un des allocateurs multi-cœur pourrait être mis en défaut par un autre, EMA affiche des performances toujours au moins proches du meilleur allocateur, et souvent meilleures.

La deuxième remarque se situe au niveau des applications Linear Regression, String Match et Histogram. Sur ces applications, au contraire des autres, TCMalloc et TBB ont des performances très similaires. On observe qu'EMA a toujours de meilleures performances que TBB et TCMalloc, sauf avec String Match, où notre environnement d'exécution est 5% moins performant que Phœnix avec TBB. Nous en déduisons qu'EMA a un bon comportement dans tous les cas.

Nous avons donc montré que nos optimisations sont efficaces pour les applications suivant un schéma MapReduce. Les gains sont notamment dûs à EMA, notre allocateur mémoire. Nous montrons qu'à la différence des allocateurs multi-cœur TCMalloc et TBB, EMA a des performances stables, souvent meilleures que les autres allocateurs étudiés, et toujours au pire proches du meilleur cas.

5.5 Conclusion

La programmation événementielle est une bonne approche pour la conception de serveurs de données efficaces. D'un autre côté, les architectures matérielles sont aujourd'hui multi-cœur, et le nombre de cœurs tend à augmenter. La coloration d'événements est une solution intéressante pour paralléliser les applications événementielle, et ainsi leur faire tirer parti des architectures multi-cœur.

Dans ce chapitre, nous avons étudié les performances de Libasync-smp, un environnement d'exécution événementiel avec coloration d'événements. Nous avons notamment identifié plusieurs limitations de performances lors de l'augmentation du nombre de cœurs. Ces limitations portent d'une part sur la gestion de la mémoire, et d'autre part sur les communications inter-cœurs.

Nous proposons quatre optimisations que nous implantons au sein de l'environnement d'exécution Libasync-smp. Deux de celles-ci visent à améliorer la gestion de la mémoire, et notamment à éviter le phénomène de faux-partage observé. Pour cela, nous utilisons tout d'abord le bourrage des données internes statiques. De plus, nous proposons EMA, un nouvel allocateur mémoire spécifique aux systèmes multi-cœur à coloration d'événements. Les deux autres optimisations que nous proposons visent à réduire les coûts de communications inter-cœurs. Plus précisément, la troisième, baptisée attente active raisonnée, a pour but d'augmenter la réactivité de l'environnement d'exécution en réduisant les coûts de réveil des threads. La quatrième optimisation vise à amortir les coûts de synchronisation, en enregistrant les événements par lots plutôt qu'individuellement. Nous évaluons l'impact de ces optimisations sur des bancs d'essai ainsi que des applications réalistes, à savoir un serveur Web, et la suite d'applications MapReduce de la bibliothèque Phœnix. Notre environnement d'exécution optimisé améliore les performances du serveur Web jusqu'à 111% par rapport à la version originale de Libasync-smp, et jusqu'à 66% par rapport à la bibliothèque Phœnix.

Bien que ces optimisations apportent un gain intéressant, les résultats obtenus jusqu'ici nous laissent penser que ce gain est plus important pour des petits événements, et pourrait être moindre pour des applications utilisant des événements à plus gros grain. Cela est notamment vrai pour les optimisations relatives aux communications inter-cœurs, comme on l'observe sur les Figures 5.16 et 5.17. La durée d'exécution des traitants de notre serveur Web est de l'ordre

de 10^3 cycles, donc nos optimisations ont du sens dans ce cas. Il se peut que l'impact de nos optimisations diminue si notre serveur supportait des opérations de traitement plus lourdes, comme du chiffrement de données par exemple. Ceci dit nous pensons que même dans ce cas les optimisations présentées ici ne réduisent jamais les performances des applications.

Les optimisations proposées dans ce chapitre sont basées d'une part sur des résultats de profilage, et d'autre part sur des bonnes pratiques de programmation multi-cœur. Elles ont été mises en œuvre et évaluées au sein d'un environnement d'exécution événementiel coloré : Libasync-smp. Cependant, nous pensons que les phénomènes auxquels nous avons fait face, tels que le faux-partage, se retrouvent dans bien d'autres cas de programmation multi-cœur. Notamment, si l'implantation d'EMA est spécifique à Libasync-smp, nous pensons en revanche que les principes utilisés peuvent être réutilisés dans des contextes différents.

Nous avons porté une attention toute particulière aux bonnes pratiques de programmation pour multi-cœur, comme par exemple le bourrage de structures de données, ou encore le traitement par lots. Si certaines de ces pratiques ne sont pas nouvelles, nous en introduisons d'autres, comme notamment l'allocation mémoire sensible à la coloration d'événements. Nous avons prouvé ici l'efficacité de telles techniques en multi-cœur. Comme les architectures multi-cœur se répandent de plus en plus, nous pensons donc que ces bonnes pratiques gagnent à être étendues à plusieurs domaines. Dans ce cadre, nous avons étudié leur intégration au sein d'un environnement d'exécution événementiel générique, Libasync-smp. Il est envisageable comme travaux futurs d'étudier leur intégration et les possibilités d'automatisation de telles pratiques au sein d'autres langages ou environnements d'exécution.

Une autre piste de travail envisagée pour la suite de ces travaux concerne les valeurs de seuils utilisées dans certaines des optimisations présentées (attente active raisonnée et enregistrement par lots). Il serait intéressant de décharger le programmeur de la recherche de la valeur optimale. Nous prévoyons donc d'automatiser le calcul des valeurs optimales (ou proches de l'optimal). Dans l'optique d'un environnement d'exécution adaptatif, il serait judicieux d'effectuer ce calcul en ligne.

Chapitre 6

Analyse comparative d'architectures de serveurs Web en multi-cœur

Sommaire

Dans ce chapitre, nous étudions les trois modèles de programmation les plus utilisés pour concevoir des serveurs Web, à savoir les modèles de thread par connexion, événementiel, et à étages. Plus précisément, nous comparons les performances d'implantations efficaces de serveurs Web représentatives de ces trois modèles. Le but de cette étude est de mieux comprendre les différences entre les comportements de ces modèles en multi-cœur, et de déterminer lequel est à même de donner les meilleures performances.

Ce chapitre est organisé comme suit. Tout d'abord, nous présentons l'état de l'art de l'évaluation de performances de serveurs Web à la section 6.1. Ensuite, la section 6.2 discute les différents mécanismes d'injection de charge existants, ainsi que nos choix. La section 6.3 présente les aspects importants pour une comparaison pertinente, ainsi que notre environnement d'expérimentation et les réglages apportés. Nous exposons les résultats obtenus à la section 6.4. Les causes sous-jacentes à ces résultats sont explicitées à la section 6.5, nous nous intéressons notamment au passage à l'échelle limité commun à toutes les implantations étudiées. Nous présentons en section 6.6 une approche simple pour contourner les limitations observées, qui augmente les performances de 13% à 4 cœurs. Les sections 6.7 et 6.8 discutent les résultats obtenus et leur portée, ainsi que les travaux futurs à envisager.

6.1 État de l'art

Nous présentons dans cette section l'état de l'art relatif à notre étude. Nous commençons par étudier les travaux de comparaison de performances de modèles de serveurs Web dans un contexte mono-cœur. Puis nous nous intéressons aux travaux d'évaluation de performances de serveurs Web en multi-cœur.

Des serveurs Web représentatifs des modèles événementiel, à étages et thread par connexion ont été comparés par Pariag *et al.* [64], dans l'optique de déterminer quel modèle était le plus adapté pour construire des serveurs Web efficaces. Ces travaux conduisent à deux conclusions. Premièrement, le réglage des paramètres des serveurs Web, notamment le nombre de connexions concurrentes, a un impact significatif sur leurs performances. Deuxièmement, le modèle "thread par connexion" a globalement de moins bonnes performances que ses homologues événementiel et à étages.

Il y a cependant plusieurs limitations à ces travaux. Tout d'abord cette étude se concentre sur un contexte mono-cœur, et donc les conclusions tirées restent à montrer dans un contexte multi-cœur. Ensuite, le choix d'utiliser Knot avec la bibliothèque Capriccio de Von Behren *et al.* [85] pour représenter le modèle de thread par connexion est discutable. En effet, Capriccio est une bibliothèque de threads utilisateurs basée sur un moteur événementiel, ce qui est d'ailleurs la principale raison pour laquelle elle ne tire pas parti des architectures multi-cœur.

Pour faire suite à ces travaux, Harji *et al.* [43, 42] ont conduit des expériences sur une petite machine multi-cœur à 4 cœurs. Ils concluent qu'en multi-cœur, leurs implantations de serveurs Web suivant les modèles événementiel et à étages ont de meilleures performances que les implantations plus largement utilisées, comme Apache, NGinx ou ligthttpd. Ces performances sont comparables avec ou sans partage d'état global : les modèles événementiel et à étages permettent de maintenir un état global, leurs équivalents N-COPY ne le permettent pas. Un état global partagé facilite l'ajout d'autres aspects, comme par exemple le contrôle d'admission. On peut cependant se poser la question de savoir pourquoi ils choisissent de limiter le nombre de cœurs à 4 depuis leur machine 8 cœurs. De plus, leur comparaison se limite aux modèles événementiel et à étages. Ces travaux éliminent le modèle "thread par connexion" en attribuant à ce modèle les mauvaises performances d'Apache. Nous argumentons que la différence de performances entre Apache et les autres serveurs étudiés vient principalement des différence de fonctionnalités et non du modèle de programmation. Nos expérimentations sur Knot (modèle "thread par connexion") montrent d'ailleurs des performances comparables aux modèles événementiel et à étages.

Les serveurs Web multi-cœur ont été étudiés par simulation auparavant. Choi *et al.* [20] simulent un matériel simple pour comparer différentes architectures de serveurs Web. Ils utilisent une charge basée sur des traces réelles. Cependant, le fort taux d'E/S présent dans cette

étude nous semble peu adapté aux situations modernes. En effet, avec les quantités actuelles de mémoire centrale, il est courant d'avoir une distribution de fichiers en mémoire pour limiter les E/S. De plus, leur simulation ne prend pas en compte les hiérarchies mémoires liées aux architectures multi-cœur. Or ce facteur est primordial dans les performances des serveurs Web, notamment de par les partages de cache matériels entre les cœurs. Bien que nous ne dénions pas les intérêts de la simulation, nous pensons que l'expérimentation en situation réelle, avec un matériel complexe, peut mener à des résultats sensiblement différents, rendant les deux approches complémentaires.

Veal et Foong [83] ont également étudié le passage à l'échelle du serveur Web Apache sur une machine 8 cœurs très similaire à la nôtre. En utilisant la charge dynamique SpecWeb2005, ils montrent qu'Apache ne passe pas à l'échelle idéalement avec le nombre de cœurs. Ils concluent qu'une contention d'origine matérielle, le bus d'adresses, est à l'origine de la baisse de performances. La principale différence avec notre analyse est que nous comparons trois modèles de programmation différents, via leur implantation respective. Une autre différence notable réside dans la différence d'injection de charge, discutée plus avant à la section 6.2.

D'autres travaux récents, comme ceux de Boyd *et al.* [14] ou Bauman *et al.* [5], ont mené des expérimentations d'implantations ad hoc de serveurs Web en multi-cœur. Des outils comme PK, de Boyd *et al.* [13] et DPROF, de Pesterev *et al.* [65] ont aussi testé les performances du serveur Web Apache dans un contexte multi-cœur. Tous ces travaux visent à stresser principalement le noyau et de ce fait utilisent une charge spécifiquement dédiée à cela. Cette charge est peu réaliste, surtout du point le vue de la distribution de fichiers utilisée. Celle-ci est souvent réduite à un seul fichier, suffisamment petit pour tenir dans les caches processeurs et ainsi stresser le système sous-jacent au maximum. De plus, notre objectif est de comparer différents modèles de serveurs Web tout en utilisant un noyau et un matériel communément utilisés, or ces études n'utilisent pas un noyau standard, excepté DPROF.

Enfin, Voras *et al.* [86] comparent différents modèles de programmation au sein d'un serveur de cache de mémoire distribuée de type memcached, sur une machine multi-cœur. Bien qu'ils comparent les modèles à étage et événementiel, il n'y a pas d'étude du modèle de thread par connexion. Ils font également état d'un biais dans leur méthodologie, car leur injecteur de charge tourne sur la même machine que le serveur. De fait, il n'y a pas vraiment d'E/S réseau, et surtout cette situation impacte l'ordonnancement sur le système. Ces résultats sont donc difficilement applicables au domaine des serveurs Web visé ici.

En somme, plusieurs études de performances de serveurs Web ont déjà été réalisées en contexte multi-cœur. Il y a peu de comparaisons de modèles de programmation, et aucune sur des machines multi-cœur supérieures à 4 cœurs (cf. Harji *et al.* [42]). De plus, nous élargissons cette comparaison en rajoutant le modèle threadé à notre étude. Nous étudions également le passage à l'échelle de chaque modèle considéré.

6.2 Injection de charge

Nous dressons dans cette section un panorama des différentes techniques d'injection de charge sur les serveurs Web ainsi que les choix effectués pour notre étude.

6.2.1 État de l'art de l'injection de charge Web

Une injection de charge Web se divise en quatre parties distinctes.

Distribution de fichiers. Nous isolons la partie *distribution de fichiers* du reste. Il s'agit de l'ensemble des fichiers demandés au serveur. Cela représente la partie statique d'une charge Web, les requêtes dynamiques étant générées à la volée par de l'exécution de code sur le serveur. On compte quatre paramètres principaux servant à définir une distribution de fichiers. Premièrement, il faut prendre en compte le nombre total de fichiers. Ces fichiers se spécifient en différentes classes. Chaque classe de fichier définit une taille (ou un intervalle de tailles) pour chaque fichier lui appartenant. Lorsqu'une classe définit un intervalle de tailles possibles, chaque fichier se voit usuellement attribuer une taille selon une distribution probabiliste. Par exemple, SpecWeb utilise une distribution Zipf pour calculer la taille d'un fichier. Il faut enfin tenir compte de la taille totale de la distribution de fichiers, ce qui va déterminer la présence d'E/S disque si cette taille est supérieure à la quantité de RAM sur la machine.

Modèle d'accès. Une fois la partie statique déterminée, il faut ensuite choisir un modèle d'accès aux ressources du serveur. Cela concerne à la fois les fichiers statiques ainsi que les pages calculées dynamiquement. Les charges les plus simples demandent une ressource au hasard parmi celles disponibles. Ceci n'est pas très réaliste, car la répartition des requêtes dépend de la structure du site (par exemple, la page d'accueil sera très probablement la plus demandée). Une amélioration consiste à réaliser une modélisation du site, souvent basée sur une chaîne de Markov, qui va déterminer la probabilité d'accès à chaque ressource. Les injecteurs SpecWeb sont basés sur une telle modélisation. Enfin, la dernière possibilité consiste à rejouer une trace des accès à un site Web réel. Cette trace peut être obtenue via les journaux d'un serveur Web existant.

Modèle d'injection. Nous nous intéressons aussi à la simulation des clients proprement dite. Il convient de distinguer trois modèles d'injection de charge. Le modèle le plus simple, et communément utilisé (présent notamment dans les différentes versions de SpecWeb [73, 75, 74], TPC-W de [36] et ApacheBench) est celui de la *boucle fermée*. Dans ce modèle, chaque client exécute une boucle qui consiste à envoyer une requête au serveur, attendre la réponse et l'analyser. Cependant, Banga *et al.* [4] ont montré qu'une injection en boucle fermée ne permet pas de saturer complètement un serveur, puisque la charge se cale sur le débit du serveur lorsque le serveur est à son plein régime. Pour pallier ce problème, il existe le modèle de *boucle ouverte*. Dans ce modèle, un client se connecte et envoie des requêtes au serveur selon un taux de charge défini, sans attendre de réponse du serveur, les réponses du serveur sont traitées à part, en parallèle. Plusieurs injecteurs de charge fonctionnent selon de modèle de boucle ouverte, notamment httperf de Mosberger *et al.* [61]. Cette approche n'est cependant pas très réaliste, car elle ne permet d'étudier qu'une surcharge brutale du serveur, de nouveaux clients arrivant continuellement et à intervalles constant. Dans un souci d'apporter du réalisme à cette situation, Schroeder *et al.* [70] ont introduit le modèle de *boucle semi-ouverte*. Il s'agit d'un compromis entre les deux modèles précédents. Ainsi, de nouveaux clients arrivent selon un taux défini, et les clients actifs peuvent partir avec une certaine probabilité après chaque requête. Tout client qui reste dans le système continue à envoyer des requêtes au serveur.

Modélisation des temporisations. Tous ces modèles d'injection supportent l'ajout de temporisations entre deux envois de requêtes. Ces temporisations sont introduites pour modéliser les temps d'attente côté client. Elles sont de deux types. Premièrement, les temporisations dites "inactives" représentent les temps d'attentes pendant lesquels l'utilisateur lit la page Web reçue. Un deuxième type de temporisation dite "active" permet de modéliser

les latences du navigateur Web entre deux requêtes. Typiquement, lorsqu'une première page est demandée, le temps d'attente actif représente le temps d'analyse de la page reçue avant l'envoi des requêtes liées aux ressources contenues dans cette page, par exemple pour des images.

6.2.2 Description de la charge retenue

Pour notre étude, nous utilisons une charge très similaire à celle utilisée par Pariag *et al.* [64]. Nous choisissons une distribution de fichiers similaire à la partie statique utilisée dans SpecWeb99. Cette distribution est constituée de 24480 fichiers, totalise 3.3Go de données. Les machines récentes présentent en effet une grande capacité mémoire (plusieurs dizaines de Go), et nous pensons que la partie statique des fichiers servis par un serveur Web doit pouvoir tenir en mémoire centrale dans la majeure partie des cas. Cela permet notamment d'éviter les E/S vers le disque dur, et augmente ainsi les performances des serveurs. Nous pré-chargeons donc en mémoire centrale (via le tampon du système de fichiers) notre distribution de fichiers avant chaque test. Nous nous concentrons sur une charge statique principalement pour deux raisons.

Premièrement, nous remarquons que la partie statique de SpecWeb représente une part importante de la charge, même dans les dernières versions (SpecWeb2009). Plus important encore, les contenus statiques restent très sollicités, on peut citer par exemple les transferts de photos, de musiques, ou de logiciels. Malgré la dynamicité grandissante du Web, beaucoup de sites choisissent d'avoir plusieurs serveurs dédiés aux différents types de contenus (statiques et dynamiques), par souci d'efficacité. On voit donc apparaître des serveurs spécialisés pour traiter les contenus statiques. Beaver *et al.* [6] ont notamment montré que le service Facebook, lors des pics de charge, sert jusqu'à 1 million d'images par seconde. Nous concluons donc que même à l'heure du Web 2.0, l'étude d'une charge statique a toujours du sens.

Deuxièmement, les charges dynamiques ajoutent des processus externes aux serveurs Web pour traiter les requêtes. La synergie de ces processus avec le serveur dépend fortement du modèle de programmation utilisé pour le serveur. Par exemple, un serveur Web événementiel n'utilise typiquement qu'un processus par cœur. Rajouter des processus externes pour traiter des requêtes dynamiques aura donc un fort impact sur l'ordonnancement : les processus du serveur Web se verront alloués moins de quanta de temps CPU. Cela peut mener à différents problèmes, comme une baisse du taux d'acceptation des connections par unité de temps, par exemple. Ce phénomène changera drastiquement si on considère cette fois un serveur Web basé sur le modèle thread-par-connection. Dans ce cas, la compétition pour le temps CPU sera dépendante du nombre de threads du serveur Web, ainsi que du nombre de processus externes. Nous intuitons donc une coopération difficile au niveau de l'ordonnancement entre le serveur et les processus de traitement de requêtes dynamiques. Une récente étude de Gaud *et al.* [38] a montré que les coûts de communication entre le serveur Web et les processus étaient également à prendre en compte. Nous laissons donc l'introduction de tels processus à de futurs travaux.

Nous utilisons l'injecteur httperf pour rejouer des traces de requêtes HTTP, en boucle semi-ouverte. Ces traces sont générées selon le modèle d'accès de SpecWeb99. Nous utilisons les mêmes temporisations que Pariag *et al.* [64], à savoir 3.0 secondes pour les attentes inactives, et 0.343 secondes pour les attentes actives.

6.3 Méthodologie

Dans le but de faire une comparaison équitable et d'en obtenir des résultats pertinents, plusieurs aspects additionnels doivent être pris en compte. Nous les présentons dans cette section.

Reproductibilité des résultats. Pour chaque résultat présenté dans ce chapitre, nous nous assurons que l'écart-type vaut moins de 4% sur au moins 10 répétitions pour les serveurs Web, et 100 répétitions pour les bancs d'essai mémoire. Par la suite, toutes les discussions sont basées sur des différences de performances significativement plus grandes que l'écart-type observé.

6.3.1 Choix d'implantations

Comme nous allons comparer des implantations représentatives de chaque modèle, il nous faut les choisir avec précaution. Nous nous assurons de deux choses ici. Premièrement que chaque implantation retenue soit bien représentative de son modèle de programmation. Deuxièmement nous prenons soin à ce que ces implantations soient comparables en termes de fonctionnalités, pour que notre comparaison de performances aie du sens.

Knot. Nous choisissons le serveur Web Knot pour représenter les implantations de serveurs à base de threads. Notre choix est dirigé par une étude précédente de ce type menée par Pariag *et al.* [64]. Knot est une implantation rapide en C construite pour démontrer l'efficacité de la bibliothèque de threads coopératifs Capriccio [85]. Comme Capriccio n'utilise qu'un seul cœur, nous avons modifié Knot pour qu'il tire parti des architectures multi-cœur. Pour cela, nous l'avons recompilé en utilisant la bibliothèque standard de threads pour Linux, la NPTL [30]. Dans l'implantation NPTL, à chaque thread utilisateur correspond un thread noyau. Cela donne au noyau la capacité de décision du placement des threads sur les cœurs, et donc permet de tirer parti des multi-cœurs. La politique d'ordonnancement est également différente de Capriccio car elle est préemptive. Bien que cela soit l'objectif de Capriccio, la NPTL, plus récente, peut aussi gérer plusieurs milliers de threads, tout en tirant parti, elle, des multi-cœurs. Nous avons également ajouté le support de l'appel système à zéro-copie `sendfile`. Cette primitive permet d'envoyer un fichier sur le réseau sans faire de copie du contenu en espace utilisateur. Si le contenu du fichier est déjà présent dans le cache du système de fichiers, alors aucune E/S disque n'est nécessaire. De fait, comme aucune copie n'est faite en espace utilisateur lors de la copie, ce mécanisme permet de tirer parti au maximum du cache du système de fichiers, situé en espace privilégié.

Userver. Userver est une implantation efficace de serveur Web événementiel. Elle a été écrite en C. μserverimplante trois modes de fonctionnement distincts : SPED (Single Process Event-Driven), SYMPED (SYmetric MultiProcess Event-Driven) et shared-SYMPED. Le premier mode SPED correspond exactement au modèle événementiel. La principale différence entre les modes SYMPED et SPED est qu'en mode SYMPED, plusieurs processus sont présents. Chaque processus exécute alors une boucle de contrôle et se comporte comme un processus SPED indépendant, à la façon d'une architecture N-COPY. Il y a cependant une différence majeure entre N-COPY et SYMPED. En effet, en mode SYMPED, les processus partagent tous la même socket d'acceptation de connexion. Cela évite notamment de gérer différents ports d'écoute. De plus l'équilibrage de charge externe, nécessaire en N-COPY,

devient facultatif. La variante shared-SYMPED étend le mode SYMPED avec un cache des descripteurs de fichiers ouverts partagé entre les processus. Par la suite nous utilisons le mode SYMPED.

Les traitants d'événements de μserver sont découpés en fonction des E/S du serveur. C'est à dire que la boucle principale accepte tout d'abord d'éventuelles nouvelles connexions, puis appelle une primitive de scrutation sur l'ensemble des sockets ouvertes. μserver traite d'abord les écritures possibles, pour terminer le plus vite possible les connexions en cours. Puis les nouvelles requêtes sont lues sur les sockets. Les lectures et écritures se font de manière non-bloquante, et losrqu'une E/S devrait bloquer, ce traitement s'interrompt pour être repris lorsque l'E/S sera non-bloquante, après une scrutation future. μserver permet d'utiliser différentes méthodes de scrutation, notamment `select`, `poll` ou `epoll`. Plusieurs méthodes sont également possibles pour l'envoi de fichiers : `write` et `sendfile`. Dans ce chapitre nous utilisons `epoll` et `sendfile`, car elles mènent à de meilleures performances que leurs alternatives dans notre environnement.

Watpipe. Watpipe [43] est une implantation en C++ d'un serveur Web à étages, basée sur le code de μserver. Le découpage en étages du serveur suit globalement celui de son homologue événementiel en traitants. Dans la pratique, il y a deux étages de scrutation : une boucle de scrutation en lecture, et une en écriture. Pour ces étages de scrutation, un thread est dédié par étage. L'étage d'acceptation de connexion crée autant de threads qu'il y a de cartes réseau sur la machine, pour permettre autant que possible d'accepter des connexions en parallèle. Viennent ensuite les étages de lecture et d'écriture. À chacun de ces étages est attribué un nombre de threads paramétrable. Chaque étage communique de façon asynchrone via des files de messages.

Le modèle à étages permet d'intégrer des réglages dynamiques du serveur, comme montré par Welsh *et al.* avec SEDA [89]. On peut notamment introduire du contrôle d'admission, ou bien un contrôle du nombre de threads par étage en fonction de différents critères. L'implantation de Watpipe ne permet pas de régler dynamiquement le serveur en fonction de la charge en entrée, comme ce qui est fait dans Haboob [88] [1]. Cependant, comme la charge que nous utilisons n'évolue pas au cours du temps, et surtout que nous réglons finement chaque serveur, nous n'avons pas besoin de réglage automatique dans nos expérimentations.

Fonctionnalités équivalentes. Les implantations choisies doivent être comparables en termes de quantité et de nature des calculs liés à chaque requête. Des fonctionnalités présentes dans Apache telles que les hôtes virtuels, des vérifications de sécurité ou encore des points d'entrée pour des modules optionnels rendraient une comparaison avec μserver inégale. En choisissant Knot, μserver et Watpipe nous nous assurons un ensemble de fonctionnalités très similaires. Pour se rapprocher encore plus, nous avons implémenté dans Knot le support de la primitive `sendfile` que nous utilisons dans μserver et Watpipe. Une comparaison de ces implantations avec la même charge a donc du sens.

6.3.2 Optimisation de la pile réseau

Nous cherchons à effectuer les réglages nécessaires pour atteindre les performances maximales avec les serveurs Web choisis. Nous commençons donc par régler la pile réseau. En termes d'équipements réseau, les machines de Grid5000 utilisées dans cette étude disposent

1. Haboob est l'implantation de serveur Web sur la plateforme SEDA.

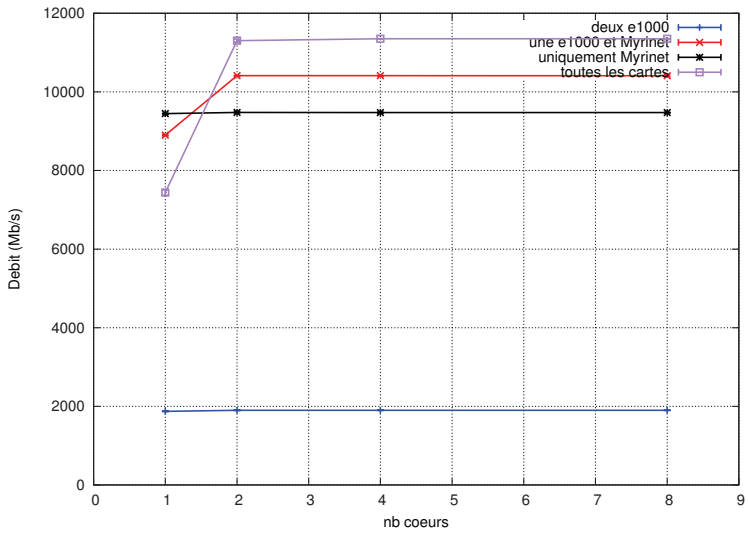

FIGURE 6.1 – Comparaison du débit maximal selon les cartes réseau utilisées.

d'une carte Myrinet à 10Gb/s ainsi que de 2 cartes Ethernet à 1Gb/s. En outre, elles sont dotées de 8 cœurs. Nous étudions donc ici la répartition des différents flots réseaux sur les cœurs, dans le souci d'augmenter au maximum les possibilités de parallélisme.

Nombre de cartes réseau. La première chose à faire est de déterminer quelles cartes réseau utiliser pour notre comparaison. En effet, la carte 10Gb/s semble incontournable pour maintenir une charge suffisante sur le serveur. En revanche, la question de savoir s'il convient d'utiliser des cartes 1Gb/s additionnelles reste ouverte. Pour cela, nous étudions le débit et la consommation CPU des différentes configurations possibles de notre matériel à l'aide du banc d'essai netperf [68].

Les Figures 6.1 et 6.2 montrent respectivement le débit et l'utilisation CPU d'une machine soumise au banc d'essai réseau netperf [68]. Les messages netperf sont envoyés par 9 machines clientes, et nous étudions le comportement du serveur en réception. Nous utilisons ici des messages de 30Ko, ce qui correspond à la taille moyenne d'un fichier accédé dans une distribution SPECWeb99. Le débit présenté ici correspond donc au débit maximal pouvant être soutenu par un serveur Web dans notre environnement avec notre charge.

Notre expérience montre qu'une carte réseau 10Gbps demande bien plus de CPU qu'une carte 1Gbps. Plus précisément, notre carte réseau 10Gbps ne peut être utilisée à 100% par un seul cœur, et n'atteint ses performances maximales qu'à partir de 2 cœurs. On remarque une faible différence d'utilisation CPU entre la carte 10Gb/s seule et les trois cartes, à 4 et 8 cœurs. Tandis que le débit avec toutes les cartes atteint 2Gb/s de plus. Nous choisissons donc par la suite d'utiliser les trois cartes réseau de nos machines. Nous retiendrons donc le débit maximal atteignable par nos serveurs Web dans cette configuration, qui est de 11,3Gb/s.

Nous pensons que cette question vient d'un besoin plus général que notre étude. En effet, nous avons vu que le nombre de cœurs avait tendance à augmenter, et que cette tendance est confirmée également pour les générations futures de processeurs. Du point de vue des cartes réseau en revanche, la tendance est différente, dans le sens où la capacité augmente par paliers plus importants (de 1Gb/s à 10Gb/s), et plus lentement. Cela mène à des situations où le nombre de cartes réseau présentes sur une machine est nettement inférieur au nombre de cœurs. C'est dans de telles situations que se posent les questions de comment utiliser la

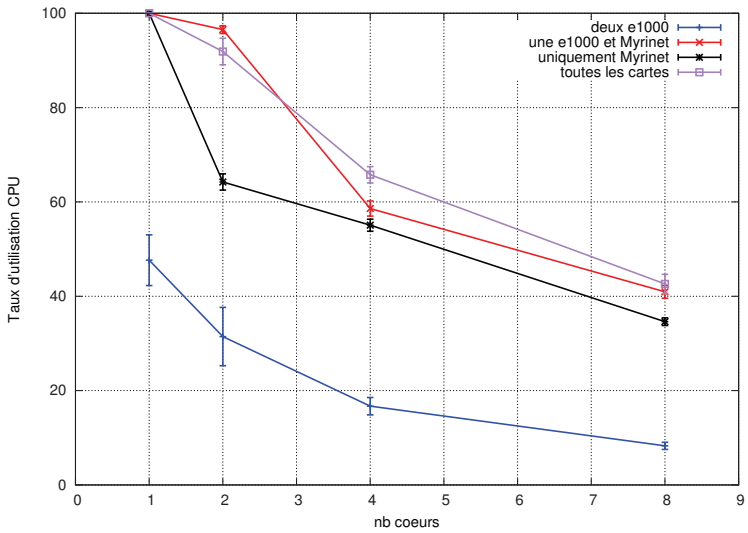

FIGURE 6.2 – Comparaison de l'utilisation CPU dédiée aux traitements réseau selon les cartes utilisées.

capacité réseau en maintenant un bon degré de parallélisme.

Affinités des interruptions. Lorsqu'on considère des flots réseaux indépendants, Dobrescu *et al.* [28] ont montré que les meilleures performances sont atteintes lorsque le traitement des paquets et des files associées à une connexion se font sur le même cœur. Notre configuration matérielle ne nous permet pas de virtualiser nos cartes réseau, nous ne pouvons donc pas créer une carte virtuelle par cœur. Nous étudions donc ici les différentes possibilités de placement des flots réseaux sur les cœurs. Pour cela nous définissons cinq stratégies de placement d'interruptions sur les différents cœurs. La configuration *one_per_core* consiste à réserver un cœur pour chaque carte réseau. Comme nous n'avons que trois cartes réseaux ici cette configuration n'utilise que trois cœurs pour la couche réseau. La courbe nommée *all_on_first_core* représente le cas dans lequel un cœur est dédié aux traitements réseaux. La configuration intitulée *allcores* est le choix par défaut du système. Il s'agit ici d'autoriser tous les cœurs à traiter des interruptions réseau. Le contrôleur d'interruption est alors en charge de choisir un cœur parmi ceux autorisés.

Nous introduisons également deux nouvelles stratégies spécifiques à notre configuration : *3_nics_myri_allcores* et *3_nics_myri_one-die*. L'idée est d'attribuer un cœur indépendant à chaque carte e1000 1Gbps, et tout un ensemble d'autres cœurs à la carte Myrinet 10Gbps. Nous distinguons deux choix pour l'ensemble de cœurs dédiés à la carte 10Gbps. La première variante est simple et utilise tous les cœurs non encore utilisés par les cartes 1Gbps. La deuxième solution ne prend pour la carte Myrinet que les cœurs ne partageant pas de cache avec ceux dédiés aux cartes e1000, pour tenter d'éviter tout perturbation dans les caches L2 partagés. Ces perturbations pourraient être du faux-partage, ou bien plus simplement une compétition pour l'espace disponible en cache.

La Figure 6.3 montre les performances de μserver soumis à une charge SpecWeb99 injectée en boucle semi-ouverte. On voit tout de suite que dédier un cœur aux traitements réseaux est la stratégie la moins efficace. A l'inverse, les stratégies les plus efficaces se révèlent être celles qui ne restreignent pas le nombre de cœurs utilisés pour les traitements réseaux. Nous adoptons donc par la suite le réglage standard appelé ici *allcores*.

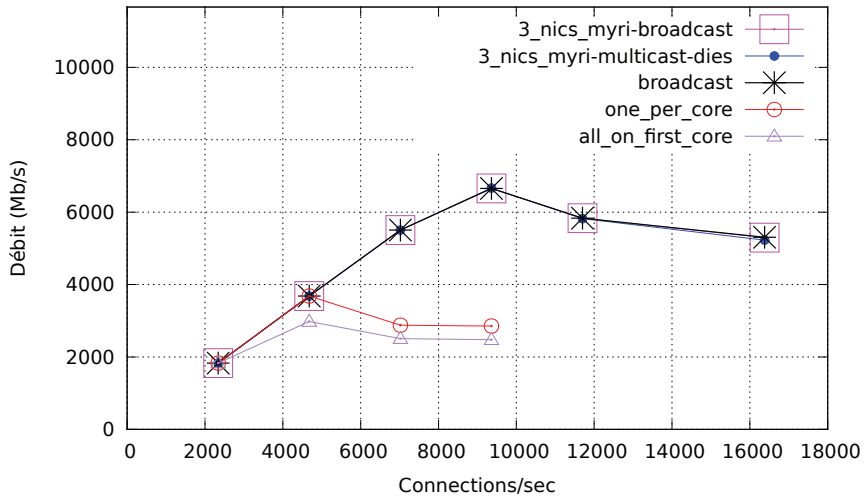

FIGURE 6.3 – Comparaison du débit en fonction des affinités d'interruptions réseau à 8 cœurs sur μserver avec une charge SpecWeb99 demandée en boucle semi-ouverte et 39 machines clientes.

Répartition logicielle des flots sur les cœurs. Étant donné que nous laissons au système le soin de répartir les interruptions réseau sur les cœurs, nous pouvons étudier les mécanismes de gestion d'affinités de plus haut niveau. Google [45] a mis au point un système de distribution logicielle des paquets réseaux. Le principe est simple : chaque paquet entrant doit pouvoir être potentiellement traité sur un nouveau cœur. Ce mécanisme est appelé *Receive Packet Steering*[2], que nous abrégerons RPS par la suite. L'idée es d'associer une clé (hash) à chaque paquet entrant, en utilisant des données du protocole, notamment l'adresse IP et les numéros de ports utilisés. A partir de cette clé, un cœur est choisi pour traiter le paquet. Par défaut, les paquets entrants sont répartis équitablement sur tous les cœurs. Ce comportement peut toutefois être altéré, et l'administrateur peut alors choisir quels cœurs associer à quelle interface réseau.

Un mécanisme de *Receive Flow Steering*[3] [46], que nous abrégerons RFS par la suite, est implanté au dessus du mécanisme de RPS. Le but est pour le noyau de détecter le cœur le plus adapté pour la réception des paquets (c'est à dire le cœur sur lequel l'application va traiter le paquet). L'implantation du RFS fonctionne actuellement uniquement pour TCP, mais le principe peut s'appliquer à tous les protocoles orientés connexion. Le RFS détecte sur quel cœur chaque application effectue des E/S réseau (via les primitives `sendmsg` et `recvmsg` du noyau). Il interprète cette information comme étant le cœur "désiré" par l'application. Il combine ensuite cette information avec le cœur calculé par le RPS, pour déterminer le meilleur choix de cœur destinataire. Ce choix tient compte du cœur désiré, et garantit que les paquets seront délivrés dans le bon ordre (des changements de cœur intempestifs peuvent conduire à un traitement des paquets dans le désordre).

Nous étudions ici l'impact des mécanismes RPS et RFS sur notre environnement. Pour cela, nous analysons le débit de μserver à 8 cœurs, en comparant les performances avec RPS, RPS et RFS, sans RPS. Nous utilisons tous les réglages présentés précédemment, c'est à dire 3 cartes réseau, et une répartition d'interruptions standard. L'implantation de RPS propose

2. Redirection de paquets en réception
3. Redirection de flots en réception

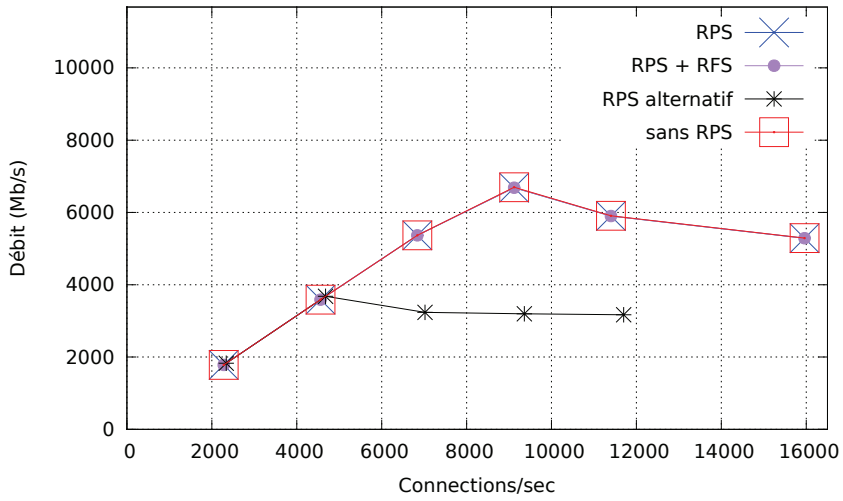

FIGURE 6.4 – Impact de la distribution logicielle de flots réseaux sur μserver avec une charge SpecWeb99 en boucle semi-ouverte. 38 machines clientes, 3 interfaces réseau actives.

deux façons différentes pour calculer le hash de chaque paquet, nous étudions donc le choix par défaut, ainsi que l'algorithme alternatif.

La Figure 6.4 montre les performances de μserver induites par l'activation des mécanismes de RPS et RFS à 8 cœurs, en fonction de la charge. On remarque que l'activation de la distribution des paquets n'a aucun effet, et les courbes avec et sans RPS sont confondues. Nous attribuons cela au fait que les interruptions sont déjà réparties intelligemment sur les cœurs par les pilotes de cartes réseaux. Cela dit le RPS n'est pas sans effet, et on observe une baisse de performances avec le calcul de hash alternatif.

Dans la suite du document, nous désactivons les mécanismes de RPS et RFS. En effet, comme nous observons les mêmes performances avec et sans ces mécanismes, nous adoptons la solution la plus simple.

6.3.3 Réglages additionnels du système nécessaires

Les réglages précédents nous assurent que le réseau fonctionne au maximum de ses capacités. La prochaine étape est de vérifier que les réglages du système ne sont pas limitants. Notre expérience nous a montré l'importance de ces réglages au sein du noyau.

Le tableau 6.3.3 présente les réglages effectués sur le système. Il s'agit notamment de spécifier les limites maximales du nombre de connexions simultanées présentes dans le système (paramètres backlog et somaxconn). Il est en effet impossible aux différents serveurs de fonctionner à pleine capacité si l'on omet ces réglages. Le cas le plus flagrant que nous avons rencontré ici est avec Knot, où l'utilisation CPU moyenne à 8 cœurs stagne à 33%, et le débit reste plat lorsque la charge varie de 5000 connexions/sec à 20000 connexions/sec. Nous observons des comportements similaires avec μserver et Watpipe. Une fois les réglages du système effectués, l'utilisation CPU et le débit des trois serveurs augmentent naturellement avec la charge.

net.ipv4.tcp_rmem	4096 87380 4194304
net.ipv4.tcp_wmem	4096 65536 4194304
net.core.rmem_default	87380
net.core.wmem_default	65536
net.core.rmem_max	4194304
net.core.wmem_max	4194304
net.ipv4.conf.all.arp_filter	1
net.ipv4.conf.eth0.arp_filter	1
net.ipv4.conf.eth1.arp_filter	1
net.ipv4.conf.eth2.arp_filter	1
net.ipv4.tcp_syncookies	0
net.ipv4.tcp_tw_recycle	1
net.ipv4.tcp_fin_timeout	1
net.ipv4.ip_local_port_range	1024 65000
net.core.somaxconn	200000
net.core.netdev_max_backlog	400000
net.ipv4.tcp_max_syn_backlog	200000
kernel.pid_max	786762
fs.file-max	1048576
kernel.randomize_va_space	0
vm.max_map_count	786762

TABLE 6.1 – Tableau récapitulatif des modifications apportées aux réglages du système.

6.3.4 Configuration de chaque serveur

Pour renforcer la pertinence de notre comparaison nous devons nous assurer que chaque serveur fonctionne au maximum de ses capacités. Pour cela, nous réglons finement et indépendamment chaque serveur de manière à atteindre son débit maximum. Nous détaillons la procédure de réglage utilisée pour chaque serveur ici.

Comme notre charge HTTP est en boucle semi-ouverte, ces réglages consistent à trouver le nombre approprié de connexions concurrentes [4] auquel un serveur atteint son débit maximum. Nous procédons à cette phase de réglages pour 1, 2, 4 et 8 cœurs.

Lors des réglages à 2 cœurs, nous remarquons pour les trois serveurs que le débit change en fonction des cœurs choisis. Nous testons 2 cœurs partageant un cache L2, puis 2 cœurs sur le même processeur mais ne partageant pas de cache L2, et enfin 2 cœurs sur deux dies différentes. Le meilleur débit est toujours atteint lorsque les cœurs choisis partagent un cache L2. C'est donc cette configuration qui est retenue par la suite.

Pour les réglages à 4 cœurs, nous soulevons la même question. Nous testons donc quatre cœurs d'un même processeur, puis deux fois 2 cœurs partageant un cache L2 sur des processeurs différents, puis 4 cœurs ne partageant aucun cache L2 entre eux. Le meilleur débit est atteint lorsqu'on utilise quatre cœurs d'un même processeur. Nous retenons donc cette configuration pour la suite.

Les légendes des figures présentant les résultats de chaque réglage sont toutes ordonnées. Le réglage apparaissant plus haut dans la légende est le meilleur que nous retenons.

4. Dans le modèle SpecWeb99, le nombre de requêtes par seconde est directement lié au nombre de connexion par seconde.

Userver. Dans le modèle événementiel, le nombre de connexions concurrentes est déterminé directement par la taille de la file d'événements. Pour μserver, nous faisons donc varier la taille de la file d'événements, représentée par le paramètre maxconn. La valeur du paramètre maxconn est fixée pour chaque instance de processus. Il est donc naturel que cette valeur n'augmente pas avec le nombre de cœurs (l'architecture SYMPED utilise un processus par cœur). La légende des Figures 6.5 6.6 6.7 6.8 présente la valeur du paramètre maxconn. Par exemple, `20K` désigne la configuration où chaque processus de μserver accepte au maximum 20 000 connexions simultanées, soit 80 000 connexions au total avec 4 processus de μserver (pour un réglage à 4 cœurs).

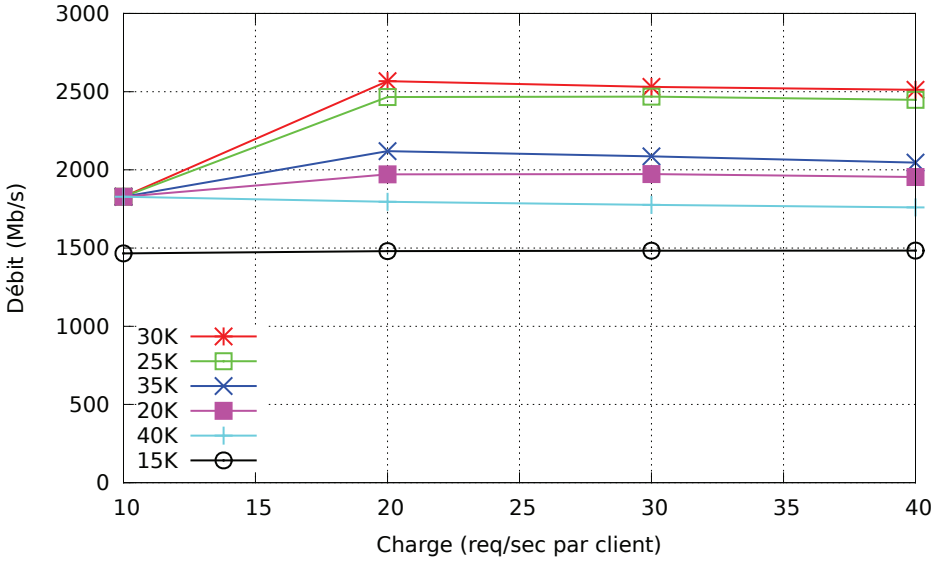

FIGURE 6.5 – Performances des différents réglages pour μserver à 1 cœur.

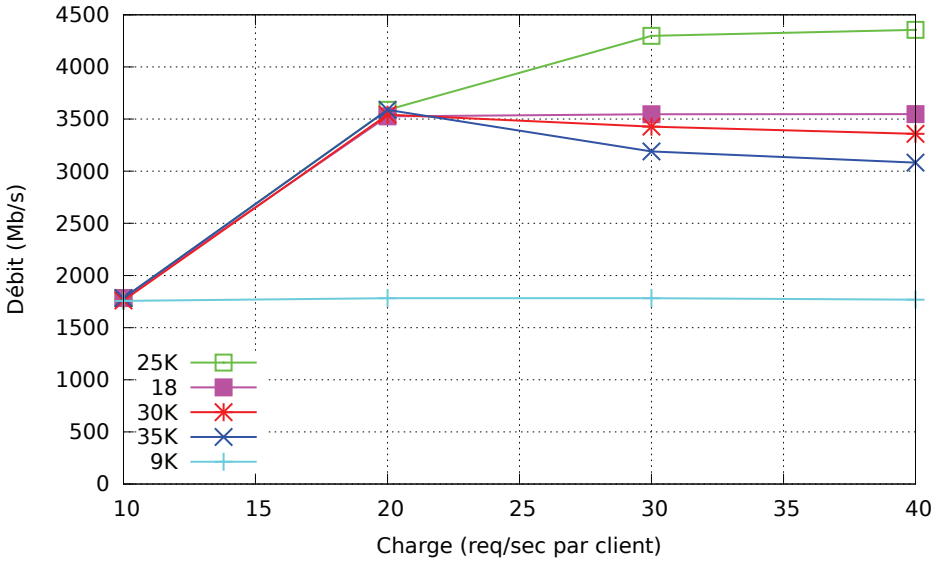

FIGURE 6.6 – Performances des différents réglages pour μserver à 2 cœurs.

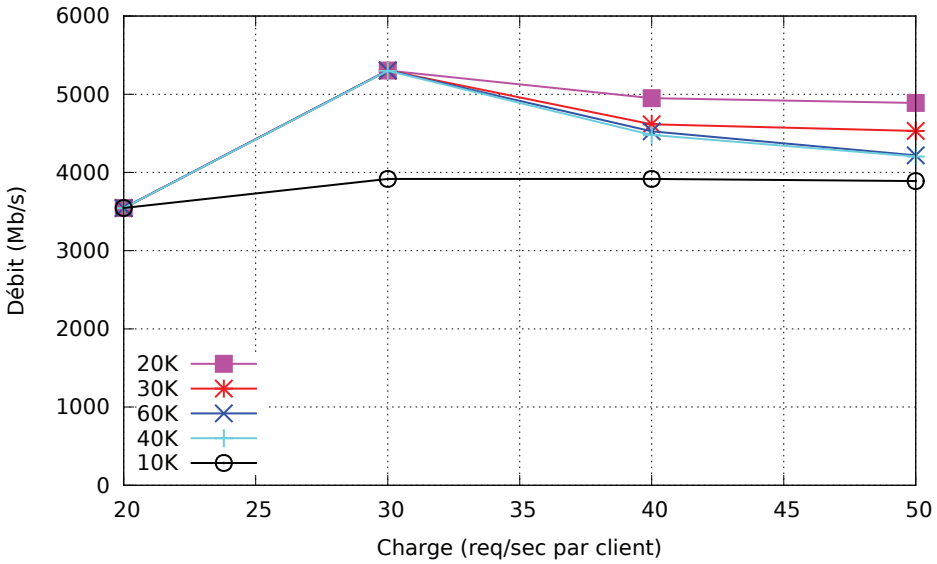

FIGURE 6.7 – Performances des différents réglages pour μserver à 4 cœurs.

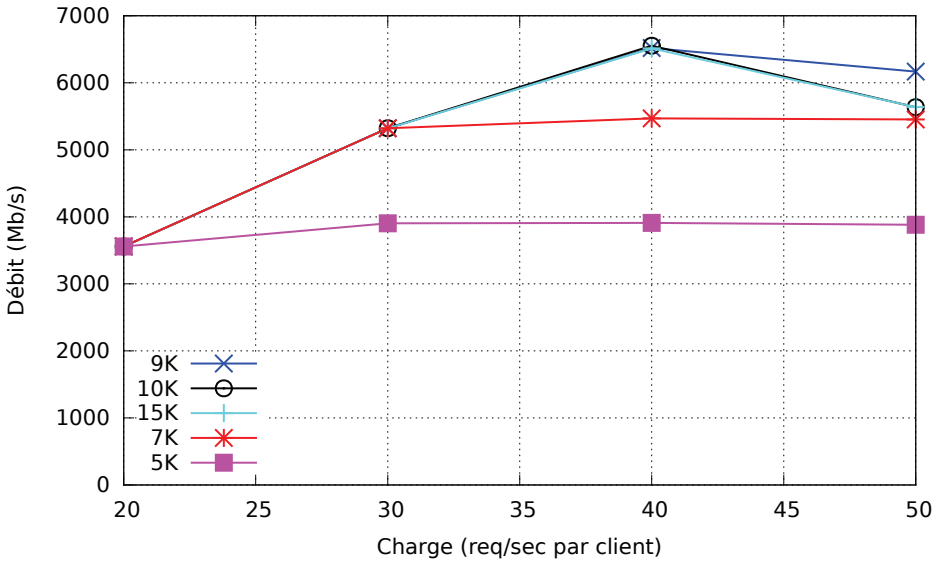

FIGURE 6.8 – Performances des différents réglages pour μserver à 8 cœurs.

Watpipe. Dans le modèle à étages, le nombre de connexions concurrentes est déterminé par le nombre de threads alloué à chaque étage, ainsi que par le nombre maximum d'éléments dans les files de ceux-ci. Des résultats préliminaires sur Watpipe nous montrent que si le nombre maximum d'éléments dans les files des étages (paramètre maxconn) n'est pas limitant, alors seul le nombre de threads associé aux étages de lecture et écriture a un impact sur les performances. Nous fixons donc le paramètre maxconn à la valeur maximum possible pour Watpipe, 480000, pour nous assurer qu'il ne soit pas limitant. Nous varions ensuite le nombre de threads associés aux étages de lecture et écriture. Nous testons également deux configurations d'E/S : en utilisant la primitive `sendfile` tantôt en mode bloquant et tantôt en mode non-bloquant. La légende des Figures 6.9 6.10 6.11 6.12 présente le nombre de threads associé aux étages de lecture et écriture, ainsi que le mode d'utilisation de `sendfile`. Par exemple, `10 - non-bloquant` désigne la configuration avec 10 threads associés aux étages de lecture et écriture, et utilisant `sendfile` en mode non-bloquant.

Nature des E/S. On remarque ici que l'utilisation de `sendfile` en mode bloquant mène le plus souvent à des performances moins bonnes que le mode non-bloquant pour Watpipe. À 4 cœurs notamment, à la Figure 6.11, toutes les configurations bloquantes ont de moins bonnes performances que toutes les configurations non-bloquantes testées.

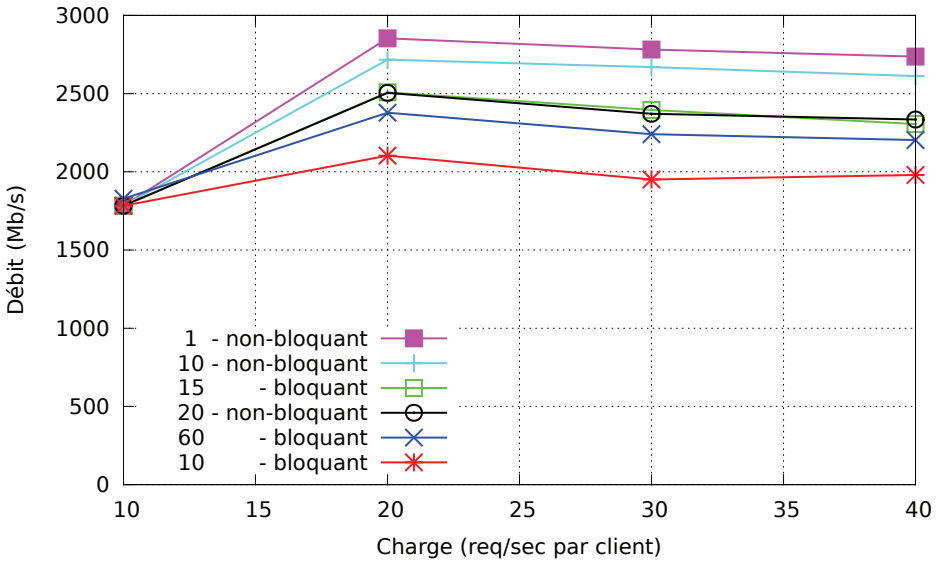

FIGURE 6.9 – Performances des différentes configurations pour Watpipe à 1 cœur.

Knot. Dans le modèle de thread par connexion, le nombre maximal de connexions concurrent est directement lié au nombre de threads dans le serveur. Pour régler Knot face à notre charge, nous faisons donc varier le nombre de threads. La légende des Figures 6.13 6.14 6.15 6.16 présente le nombre de threads associés à Knot. Par exemple, `20K` désigne un réglage avec 20 000 threads.

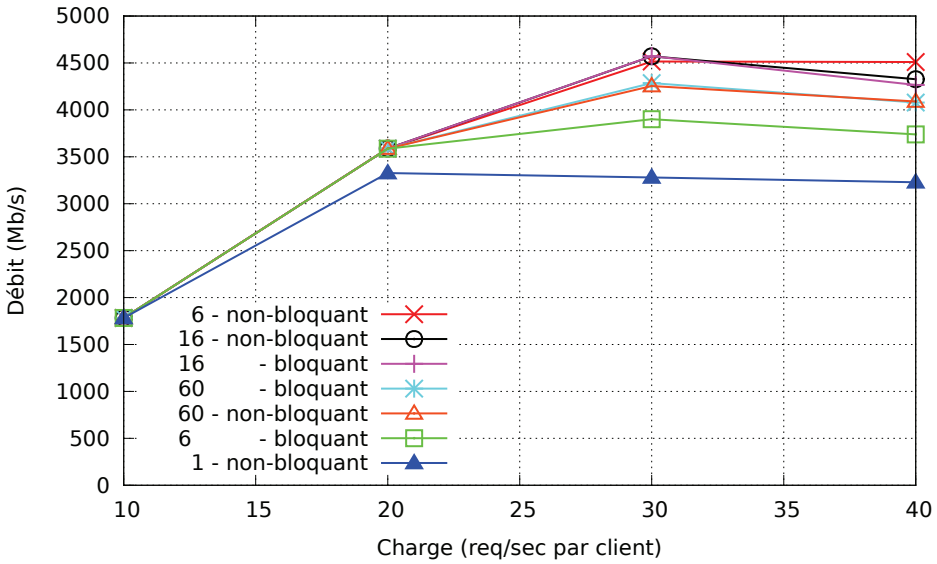

FIGURE 6.10 – Performances des différentes configurations pour Watpipe à 2 cœurs.

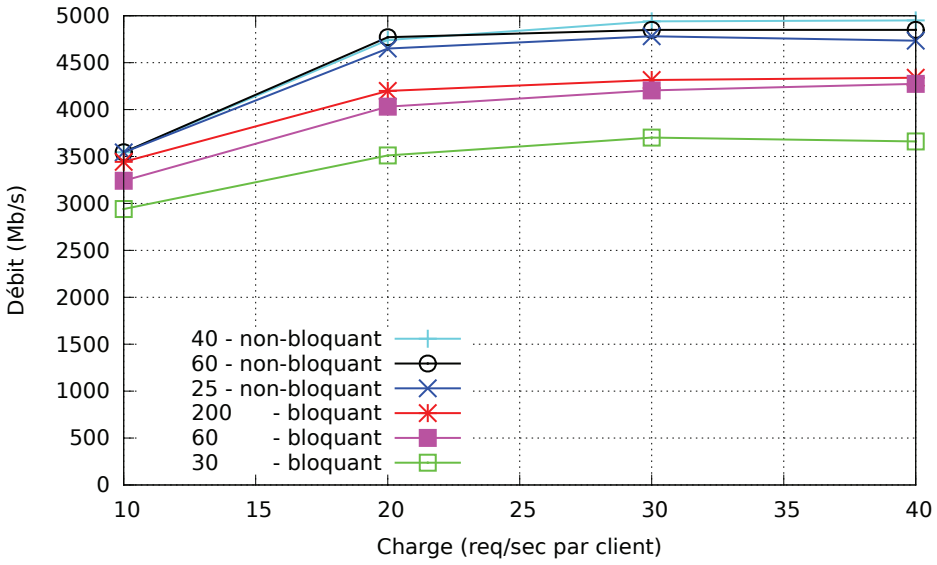

FIGURE 6.11 – Performances des différentes configurations pour Watpipe à 4 cœurs.

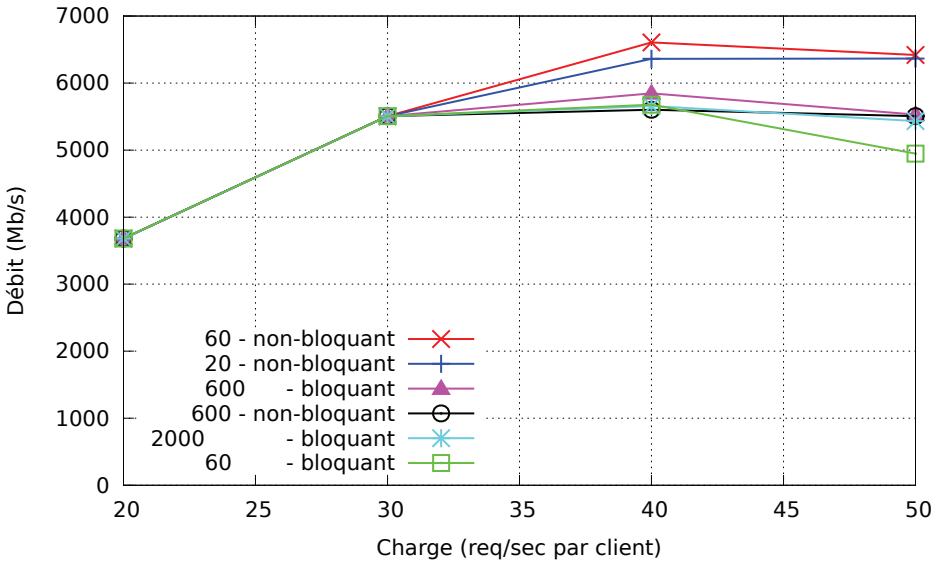

FIGURE 6.12 – Performances des différentes configurations pour Watpipe à 8 cœurs.

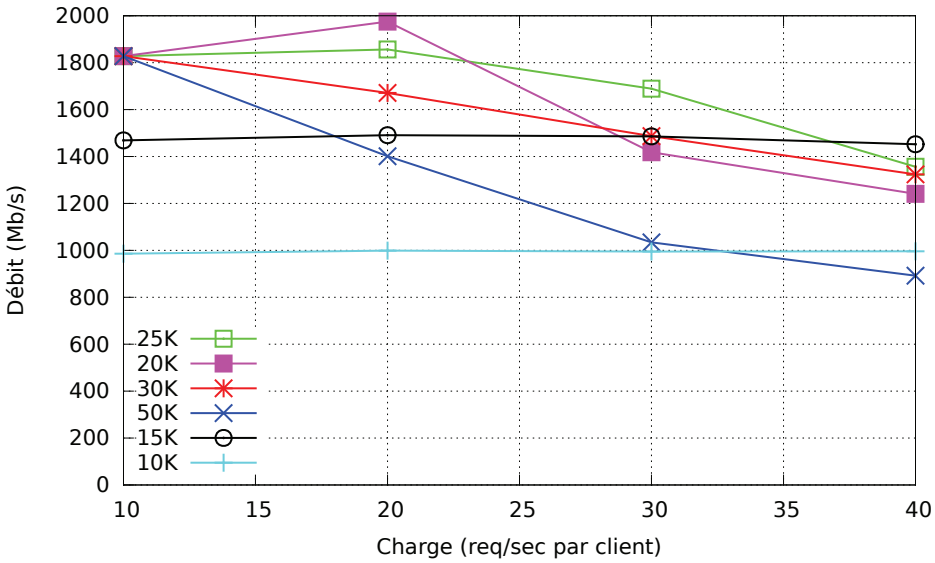

FIGURE 6.13 – Performances des différentes configurations pour Knot à 1 cœur.

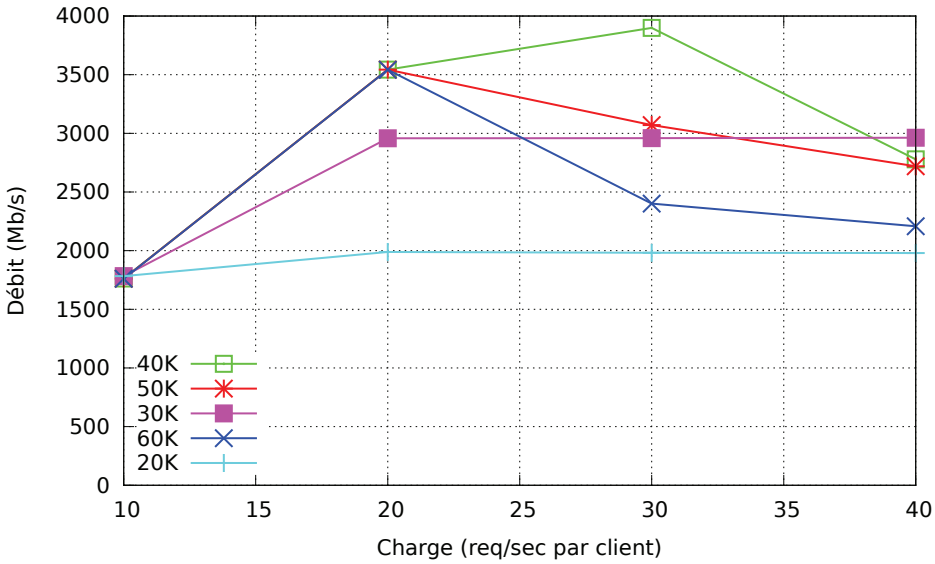

FIGURE 6.14 – Performances des différentes configurations pour Knot à 2 cœurs.

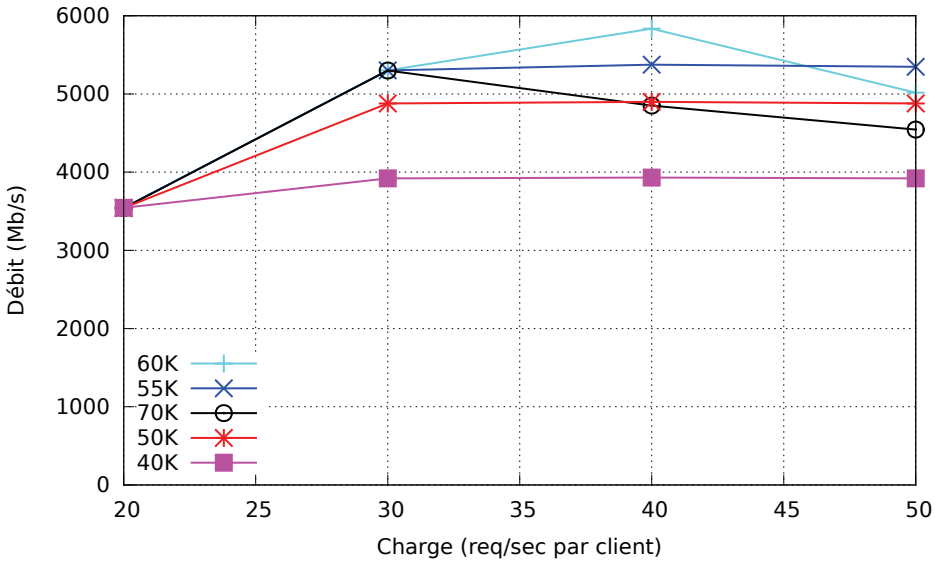

FIGURE 6.15 – Performances des différentes configurations pour Knot à 4 cœurs.

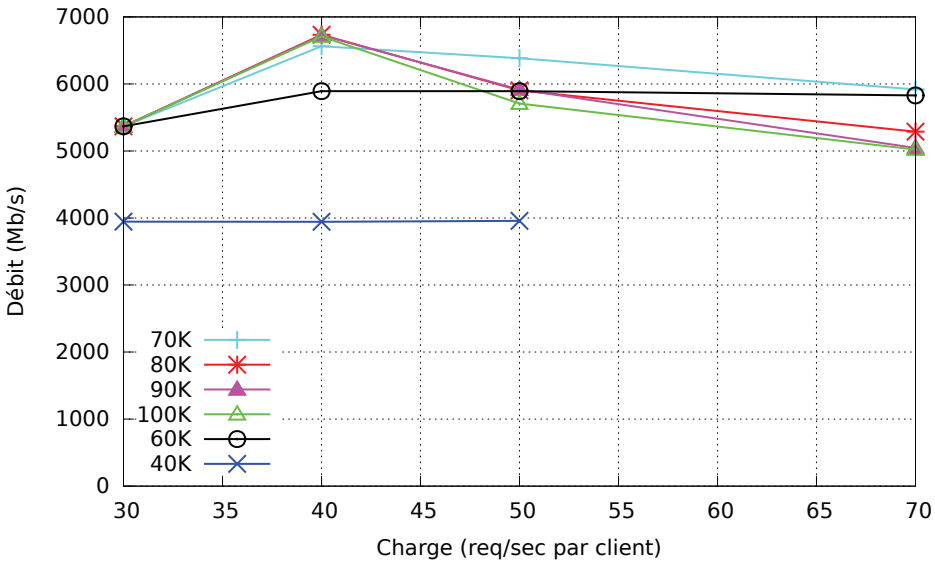

FIGURE 6.16 – Performances des différentes configurations pour Knot à 8 cœurs.

6.4 Comparaison de performances

Cette section présente les résultats de notre étude, accompagnés de nos analyses et de nos conclusions.

6.4.1 Observation du débit

À un cœur. La Figure 6.17 présente le débit des trois serveurs étudiés en fonction de la charge, à 1 cœur. On retrouve ici les mêmes tendances que celles observées précédemment par Pariag *et al.* [64] et Harji *et al.* [43], à savoir que μserver et Watpipe ont un meilleur débit que Knot.

La différence de débit entre Knot et Watpipe observée ici est de 35% au maximum. Elle est bien plus importante ici que dans les études précédentes, où la différence maximum de débit était de 18%. Cela peut s'expliquer par plusieurs paramètres. Premièrement, les études précédentes ne pré-chargent pas toute la distribution de fichiers en mémoire, comme nous le faisons. Elles tiennent donc compte des E/S qui peuvent ralentir les serveurs et donc atténuer les différences de performances entre eux. Deuxièmement, le matériel utilisé n'est pas le même. On remarquera surtout les différences de cartes réseau utilisées. En effet, nous utilisons une carte 10Gb/s et deux cartes 1Gb/s, tandis que Harji *et al.* utilisent huit cartes 1Gb/s. Cela mène à des différences de configurations qui peuvent influer grandement sur ce type d'expérimentations. Enfin, nous utilisons une bibliothèque de threads différente pour Knot (NPTL au lieu de Capriccio), et les réglages systèmes sont très certainement différents [5]. Il est intéressant de remarquer que toutes ces différences nous permettent d'atteindre un débit maximum presque deux fois supérieur à ceux observés lors des précédentes études : 2.8Gb/s contre 1.5Gb/s au maximum pour Pariag *et al.* [64] et Harji *et al.* [43]. Cela nous permet de penser que nous étudions les serveurs Web dans de meilleures conditions, dans le sens où

5. Nous n'avons pas de moyen de vérifier cela, sachant qu'aucune des études de Pariag *et al.* [64] et Harji *et al.* [43] ne divulgue ses réglages systèmes, mais notre hypothèse se base sur les différences de matériel menant à des réglages différents.

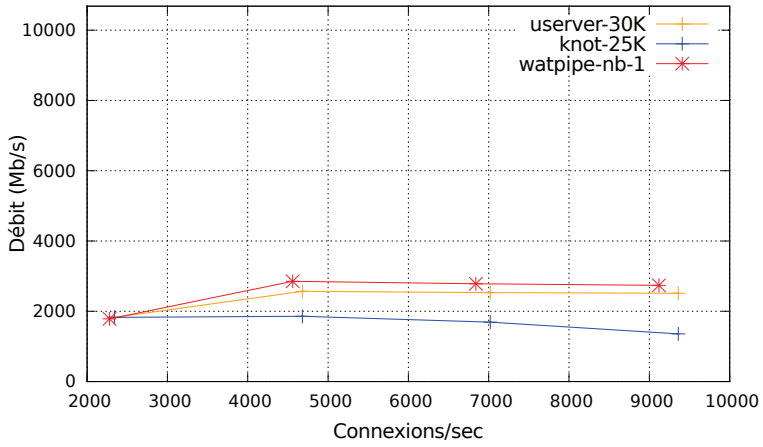

FIGURE 6.17 – Débit en fonction la charge des trois serveurs à 1 cœur.

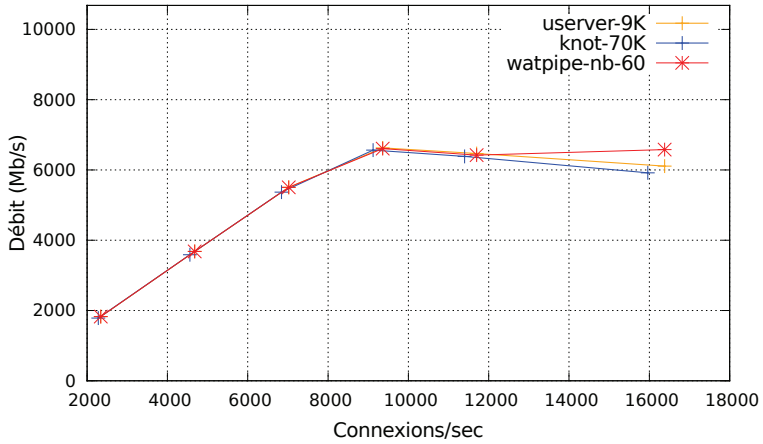

FIGURE 6.18 – Débit en fonction la charge des trois serveurs à 8 cœurs.

notre environnement nous permet d'obtenir de meilleures performances.

La Figure 6.17 suggère que le modèle à étages est le plus propice à exhiber un meilleur débit que ses homologues événementiels, et surtout threadés. En effet, ici Watpipe surpasse μserver d'au plus 10% et Knot d'au plus 35%. Nos conclusions restent donc similaires à celles de Pariag *et al.* [64] et Harji *et al.* [43].

À huit cœurs. La Figure 6.18 présente le débit des serveurs Web en fonction de la charge à 8 cœurs. Il est important de remarquer le débit maximum est atteint simultanément par tous les serveurs, en ce point les débits coïncident pour atteindre 6.6Gb/s.

Après ce point (9000 connexions/sec et delà), tous les serveurs sont saturés, et le débit diminue. Le réseau n'est pas limitant ici, car on a vu qu'il était possible d'atteindre 11Gb/s avec notre environnement, et la même taille moyenne des messages. Le chiffre du débit maximum atteint à 8 cœurs est surprenant car il correspond à seulement 2,3 fois celui à 1 cœur. Comme les trois modèles choisis peuvent pleinement tirer parti des architectures multi-cœur, et que les performances des serveurs sont limitées par le CPU, nous nous attendons à un meilleur passage à l'échelle, au moins jusqu'aux limites du réseau. Cependant, les données

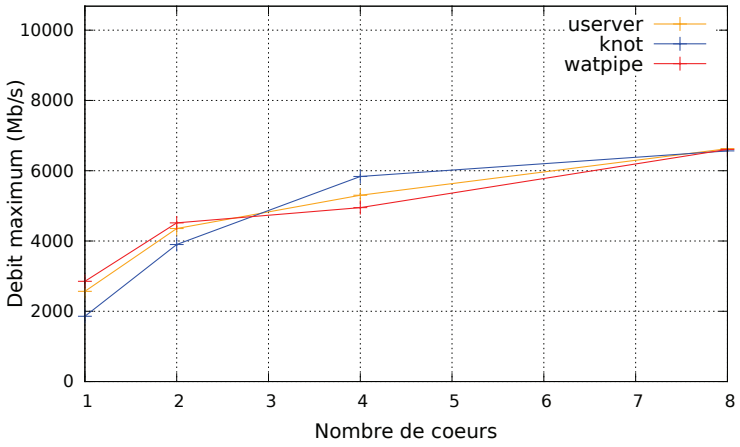

FIGURE 6.19 – Meilleur débit de chaque serveur en fonction du nombre de cœurs.

expérimentales nous montrent que le réseau n'est pas aux limites de ses capacités, et pourtant les serveurs présentent un mauvais passage à l'échelle.

6.4.2 Analyse du passage à l'échelle

Nous menons une analyse du passage à l'échelle afin de mieux comprendre les performances des trois serveurs.

La Figure 6.19 présente le passage à l'échelle observé pour chaque serveur. Cette courbe est obtenue en prenant uniquement le meilleur point de chaque serveur pour chaque configuration de cœurs. Le fait le plus évident sur cette Figure est le problème de passage à l'échelle apparaissant à partir de 4 cœurs. Ce problème est tellement important que tous les serveurs ont les mêmes performances à 8 cœurs.

Remarque. Une inversion de tendance apparaît à 4 cœurs. C'est à dire que Knot devient meilleur que Watpipe et μserver. Nous observons ici que Knot souffre moins du problème de passage à l'échelle que les deux autres serveurs, et profite ainsi d'un répit avant que cette saturation impose à chaque implantation la même limitation de performances aperçue à 8 cœurs. Nous expliquons ce comportement avec un profilage fin à la section 6.5.

6.5 Profilage

Dans cette section, nous présentons les résultats d'un profilage à grain fin, qui nous permet de comprendre les limitations auxquelles font face les serveurs Web lors de nos expériences.

6.5.1 Origine de la limitation

Généralement, pour comprendre les performances d'une application, une bonne métrique est le nombre de *Cycles Par Instructions*, ou CPI. Comme son nom l'indique, cette métrique mesure le temps processeur nécessaire à l'exécution complète d'une instruction. Sur des processeurs multi-scalaires, le nombre de CPI optimal est inférieur à 1, car le processeur peut

nb cœurs	CPI μserver	CPI Knot	CPI Watpipe
1	2.1	2.0	1.8
2	2.5	2.1	2.2
4	3.4	3.0	3.3
8	3.9	3.4	4.1

TABLE 6.2 – Évolution des CPI avec le nombre de cœurs

exécuter plusieurs instructions simultanément. Dans le cas idéal, cette métrique ne varie pas lorsqu'on augmente le nombre de cœurs.

Dans la pratique cependant, on observe souvent des variations avec le nombre de cœurs actifs. Le nombre de CPI peut en effet varier selon l'efficacité des caches, des accès mémoire, et des communications inter-cœurs. Lorsque le nombre de cœurs change, l'utilisation des caches partagés change, causant des variations sur les CPI. En effet, les accès à la mémoire deviennent bien plus lents lorsque la donnée n'est pas présente en cache, et les CPI augmentent avec la latence des instructions. Il peut également diminuer en cas de contention sur un `spinlock` (verrou à attente active), car beaucoup d'instructions sont alors exécutées à intervalles de temps très rapprochés. Or chaque tentative d'acquisition de spinlock tente d'écrire dans la zone partagée contenant le verrou. Cela est fait via des instructions atomiques, qui bloquent tout accès au cache contenant cette zone, même les accès en lecture provenant des autres cœurs. Étant donné les multiples variables qui influent sur les CPI, il convient de raffiner ensuite les résultats obtenus à partir des CPI avec d'autres, relatifs aux causes possibles.

Le tableau 6.5.1 présente les CPI observés pour chaque serveur Web selon le nombre de cœurs actifs. On remarque que ce nombre double entre 1 et 8 cœurs. Nous sommes donc face à une augmentation pathologique des CPI. Cela correspond généralement à des temps de blocages CPU pendant lesquels le processeur ne peut exécuter les instructions présentes dans le pipeline. Nous choisissons donc par la suite d'étudier en détails les différentes causes de blocages CPU.

Une compétition pour des verrous ferait soit baisser le nombre de CPI dans le cas de verrous à attente active, soit endormirait les processus compétiteurs et ainsi diminuerait l'utilisation du processeur vue par le système d'exploitation (au moins pour le cas événementiel, où il n'y a qu'un seul processus par cœur). Or le processeur est pleinement utilisé et le nombre de CPI augmente. Nous éliminons donc la possibilité d'un problème lié à l'acquisition de verrous.

Pour étudier plus en détails les causes possibles de l'augmentation du nombre de CPI, nous utilisons des compteurs de performances spécifiques à la micro-architecture de nos processeurs. Plus précisément les compteurs `RESOURCE_STALL_*` permettent d'identifier les causes de blocages au niveau du processeur. La Figure 6.20 présente une décomposition du nombre de CPI selon les compteurs disponibles. Les cycles présentés ici sont ramenés à des composantes des CPI et classifiés comme suit :

- "Branch" : correspond aux cycles bloqués à cause d'une erreur de prédiction de branchement.
- "Load/Store" : compte les cycles durant lesquels tous les tampons associés aux opérations Load/Store sont pleins.
- "ROB" : représente le temps durant lequel le tampon de ré-ordonnancement d'instructions (Re-Order Buffer) est plein. Le tampon de ré-ordonnancement contient les instructions en cours d'exécution. Il est plein lorsque beaucoup d'instructions dans le

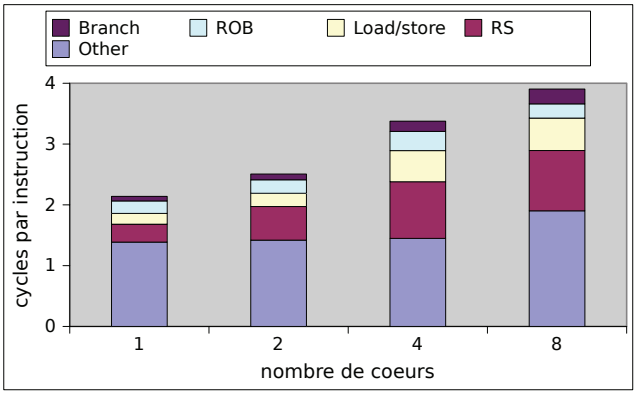

FIGURE 6.20 – Décomposition des CPI sur μserver

pipeline ont une grande latence. Aucune nouvelle instruction ne peut entrer dans le pipeline tant que ce tampon est plein.

– "RS" : compte les cycles durant lesquels toutes les Stations de Réservation sont pleines. Les stations de réservation sont des tampons intermédiaires pour s'abstraire des registres. En effet, les architectures multi-scalaires faisant du renommage de registres, ces stations servent pour stocker les opérandes d'instructions ré-ordonnées. Ici aussi, aucune nouvelle instruction ne peut entrer dans le pipeline tant que ces stations de réservation sont pleines.
– "Autres" : sont les cycles restants. Cela correspond aux cycles utiles, passés à exécuter les instructions du programme, ainsi qu'à d'autres blocages potentiels non comptés dans cette architecture.

On observe dans la Figure 6.20 une hausse significative des événements "Load/Store" et "RS". Les stations de réservation sont des tampons contenant les opérandes et les résultats des instructions, utilisées dans l'algorithme d'ordonnancement dynamique d'instructions de Tomasulo pour les processeurs super-scalaires présenté Hennessy et Patterson [44]. Lorsque celles-ci sont pleines, le pipeline bloque l'exécution de toute instruction supplémentaire, jusqu'à ce qu'une place se libère pour stocker les opérandes et le résultat d'une nouvelle opération. Il en va de même lorsque les tampons associés aux Load/Store : le pipeline bloque tout nouvelle instruction jusqu'à ce qu'une opération "load" soit complètement exécutée (et ainsi retirée du pipeline), ou bien que le résultat d'une opération "store" soit visible en cache ou en mémoire centrale [6].

A 8 cœurs, les événements "Load/Store" et "RS" représentent à eux seuls 40% du nombre total de CPI. Une telle proportion indique la présence d'opérations à grandes latences dans le pipeline. Il s'agit très certainement d'accès mémoires, et d'opérations interdépendantes, qui ne peuvent profiter de l'aspect multi-scalaire du processeur. Nous pouvons donc déduire des chiffres de la Figure 6.20 que la principale source du problème de passage à l'échelle des serveurs Web est due aux latences liées aux accès mémoires, qui augmentent avec le nombre de cœurs.

Remarque. Nous remarquions à la Figure 6.19 que les performances de Knot à 4 cœurs surpassent celles de μserver et Watpipe. Le fait que le nombre de CPI de Knot est plus faible

6. Une troisième possibilité de comptage de cet événement est lié aux instructions de barrière mémoire de type `mfence`. Nous éliminons cette possibilité car aucune barrière n'est présente dans les implantations considérées.

que ceux de μserver et Watpipe suggère que Knot souffre moins des grandes latences d'accès mémoire. Cela peut s'expliquer par le grand nombre de threads présents dans le système, demandant au système plus de temps de calcul pour les gérer. En effet, si le nombre de changements de contexte est similaire entre Knot et Watpipe à 4 cœurs (de l'ordre de 140K changements/sec), le nombre de threads à gérer est en revanche sensiblement différent. À 4 cœurs, Watpipe a 85 threads en tout, tandis que Knot a 60K threads, ce qui fait une différence de 3 ordres de grandeur. Notre hypothèse est que le temps de calcul nécessaire à chaque changement de contexte est sensiblement différent selon le nombre de threads en jeu.

Pour confirmer cette hypothèse, nous relevons les 10 fonctions dans lesquelles les serveurs Web passent le plus de temps à 4 cœurs (chiffres obtenus via l'outil Oprofile). On observe notamment que la fonction `sched_migrate_task` apparaît dans les 10 premières fonctions pour Knot (occupant 2% du temps d'exécution), alors qu'elle est absente chez μserver et presque inexistante chez Watpipe (0.1% du temps d'exécution). Cette fonction est utilisée par le noyau pour migrer l'exécution d'un thread d'un cœur à un autre. Cela confirme notre hypothèse selon laquelle le temps nécessaire au noyau pour effectuer un changement de contexte est plus important pour Knot que pour Watpipe et μserver, car il dépend du nombre de threads. Cet ajout de calculs a pour conséquence de relâcher suffisamment la pression sur la mémoire pour permettre à Knot d'avoir un meilleur débit que Watpipe et μserver. En somme, l'inefficacité de Knot observée à 1 cœur (débit 35% plus faible que Watpipe) lui permet de gagner en performances à 4 cœurs.

6.5.2 Coût de la cohérence mémoire

Dans le but d'analyser l'origine de l'augmentation des latences d'accès à la mémoire, nous raffinons le profilage des serveurs Web.

Observations. Nous observons tout d'abord un comportement très similaire pour les trois serveurs vis à vis des caches L2 et des TLB. Le nombre de fautes de cache L2 par instruction est toujours inférieur à 0.009, tous serveurs et nombre de cœurs confondus. De même, pour les TLB, ce nombre reste toujours en deçà de 0.008 fautes de TLB de données par instruction. Ces chiffres sont d'une part trop faible, et d'autre part n'ont pas une variation suffisante avec le nombre de cœurs pour être tenus responsables de l'augmentation des CPI. Nous éliminons donc les fautes de caches et de TLB comme causes directes de l'augmentation des coûts d'accès à la mémoire. Nous intéressons ensuite à utilisation du bus, présentée Figure 6.21. Le premier fait remarquable sur cette Figure est que les trois serveurs Web exhibent sensiblement le même comportement de bus. Lors de nos expériences, l'utilisation du bus de données augmente de 1 à 4 cœurs, puis stagne autour de 20% à 4 et 8 cœurs. Il en va autrement pour le bus d'adresses, dont l'utilisation est anormalement élevée à partir de 4 cœurs (50%), et qui montre une saturation à 8 cœurs avec plus de 80% d'utilisation. Cette situation suggère une grande contention du bus d'adresses, limitant le débit du bus de données.

Mise en œuvre de la cohérence mémoire. Le bus d'adresses est ici bien plus utilisé que le bus de données car il sert à transmettre les messages du protocole de cohérence mémoire, appelés "snoops". Sur notre architecture, ainsi que sur la plupart des architectures standard x86, les protocole de cohérence mémoire couramment utilisés sont MESI [22] et MOESI [27]. Ces deux protocoles fonctionnent sur le même principe. Il s'agit d'associer à chaque ligne de cache un état parmi les suivants possibles : modifié, exclusif, partagé ou invalide[7]. Cela sert

7. M(odified), E(xclusive), S(hared), I(nvalid)

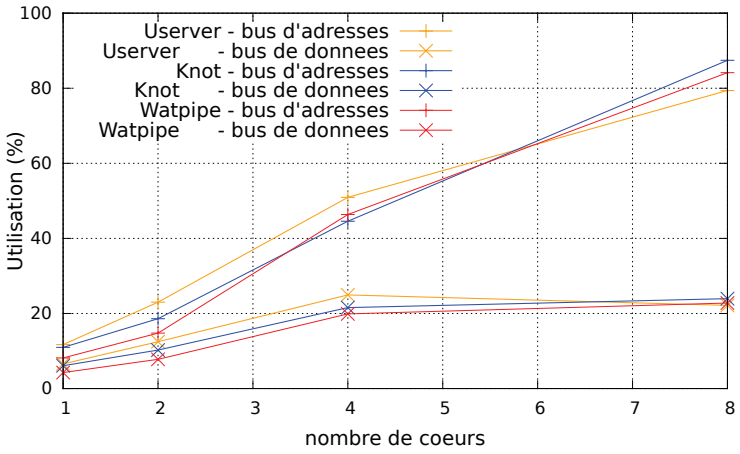

FIGURE 6.21 – Utilisation des bus en fonction du nombre de cœurs.

	M	E	S	I
M	X	X	X	OK
E	X	X	X	OK
S	X	X	OK	OK
I	OK	OK	OK	OK

TABLE 6.3 – Compatibilité des états MESI.

à représenter l'état des données d'une ligne de cache par rapport à leurs copies présentes en mémoire centrale ou dans les caches voisins. Pour garder la mémoire cohérente, il faut s'assurer qu'il existe au plus une copie modifiée parmi toutes celles existantes. Le Tableau 6.3 présente les compatibilités pour une même ligne de cache vues de deux caches dans le système. Pour maintenir la cohérence, à chaque opération de lecture ou d'écriture sur une ligne de cache, le protocole spécifie que selon l'état de cette ligne des messages de contrôle (snoops) seront envoyés ou non aux autres caches, afin de maintenir chaque copie dans un état cohérent. Ces snoops transitent sur le bus d'adresses.

La Figure 6.22 présente le nombre de snoops par instruction lorsque le nombre de cœurs varie. On peut y observer que pour un nombre d'instructions exécutées fixe, le nombre de messages envoyés augmente avec le nombre de cœurs. Nous en déduisons que ce sont ces snoops qui provoquent une saturation du bus d'adresses lorsque le nombre de cœurs utilisés augmente.

Nous rencontrons donc une situation où le bus d'adresses est saturé pour chacune des implantations de serveur Web. Nous pouvons donc conclure que pour les architectures multi-cœurs UMA, le modèle de programmation n'impacte pas ou peu le débit des serveurs Web.

6.6 Réduction du partage de données via N-COPY

6.6.1 Approche

Comme le protocole de cohérence mémoire est la cause de la baisse de performances observée, nous cherchons ici à réduire le partage de données entre leurs cœurs. Pour cela, nous utilisons différents déploiements basés sur la stratégie N-COPY. L'idée ici est de réduire

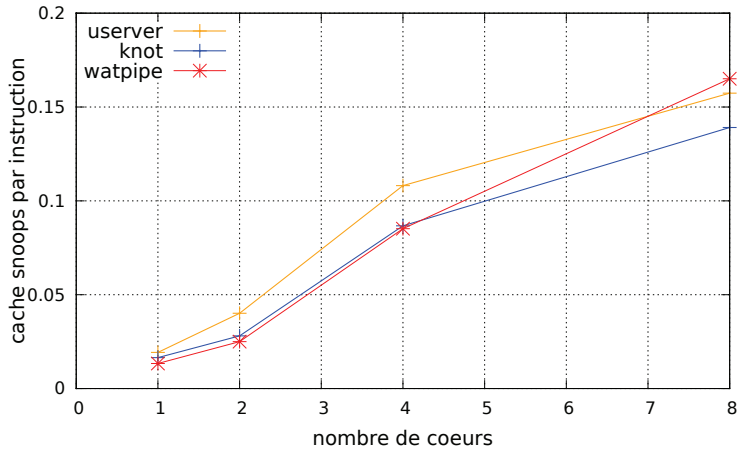

FIGURE 6.22 – Nombre de snoops par instruction, en fonction du nombre de cœurs.

le partage de données entre les cœurs en utilisant des processus indépendants.

En effet, si chaque requête était traitée de façon indépendante jusqu'alors, la socket principale d'acceptation de connexion était partagée entre les processus. En N-COPY, chaque copie est un processus indépendant, qui écoute sur une socket différente, et donc un port TCP différent. Cela implique que le traitement des connexions acceptées sur un port TCP ne partage plus aucune donnée avec les connexions acceptées sur un port différent. Nous espérons ainsi limiter les effets néfastes du partage de données, et augmenter les performances de nos serveurs Web. Cette stratégie requiert l'utilisation d'un mécanisme d'équilibrage de charge externe, pour assurer que chaque processus utilisera au maximum les cœurs qui lui sont attribués.

Nous avons remarqué jusqu'ici qu'à 2 cœurs les serveurs Web tiraient parti du partage de données en cache. En effet, nos expérimentations à 2 cœurs montrent que choisir 2 cœurs partageant un cache L2 mène à de meilleurs débits que 2 cœurs ne partageant pas de cache. Cela nous mène à tester différentes manières de répartir les copies selon le nombre de cœurs, que nous appelleront `partitionnement N-COPY` par la suite. L'idée étant alors non pas d'abolir tout partage de données, mais de l'utiliser de façon contrôlée. Un partitionnement N-COPY peut se définir comme suit : N copies indépendantes de M processus partageant une même socket d'acceptation de connexions. Nous utilisons nos précédentes observations pour faire en sorte que les M processus partageant des données partagent également un maximum de la hiérarchie mémoire. Typiquement lorsque M vaut 2, nous plaçons ces 2 processus sur deux cœurs partageant un cache L2. De fait, nous testons dans cette section les différents partitionnements N-COPY possibles. Par exemple, à 8 cœurs, nous testons 8 copies de 1 processus, puis 4 copies de 2 processus (les processus étant disposés de telle façon que chaque copie partage un cache L2), puis 2 copies de 4 processus (les processus partageant deux à deux un cache L2).

6.6.2 Résultats

Le Tableau 6.4 présente le débit maximum obtenu pour chaque configuration de chaque serveur Web. Pour chaque nombre de cœur, nous spécifions le nombre de copies indépendantes utilisées. Le nombre M de processus de chaque copie (partageant des données) se déduit facilement comme suit : $M = \frac{nombre\ de\ copies}{nombre\ de\ c\text{œ}urs}$. Les configurations n'utilisant qu'une seule copie

nombre de cœurs	nombre de copies indépendantes	μserver	Watpipe	Knot
1	1	2567	2853	1856
2	1	4355	4516	4090
	2	4362	4027	4090
4	1	5302	4950	5835
	2	5968	5584	5848
	4	5806	4447	4399
8	1	6626	6607	6563
	2	6569	6552	6399
	4	6564	6561	5598
	8	6566	5634	3327

TABLE 6.4 – Débit maximum pour chaque serveur Web (Mb/s)

sont exactement celles étudiées précédemment, où tous les processus partagent des données.

Les effets de cette réduction de partage sont le plus observables à 4 cœurs. En effet, nous avons vu que les configurations originales de Watpipe et μserver ne sont pas aussi performantes que Knot à la section 6.4.2. En revanche, le N-COPY améliore les performances de μserver et Watpipe de 13% par rapport aux configurations sans N-COPY. Cette augmentation ramène les serveurs événementiel et à étages à des performances similaires à celles de Knot. Cela signifie que la duplication de la socket d'écoute réduit suffisamment la contention sur le bus mémoire pour améliorer les performances de Watpipe et μserver, là où Knot réduit cette contention avec les coûts liés aux changements de contextes.

On remarque toutefois que la stratégie N-COPY 1 copie par cœur (dernières lignes du tableau 6.4) n'est jamais la meilleure. Cette stratégie heurte même significativement les performances à 4 et 8 cœurs. Par exemple, le débit de Watpipe à 8 cœurs lorsque 8 copies sont présentes (5634 Mb/s) est 15% moins bon que le meilleur débit de Watpipe à 8 cœurs (6607 Mb/s). On note que la mémoire consommée augmente avec le nombre de copies indépendantes, notamment à cause de la duplication des données. Pour Watpipe, la consommation mémoire à 2 copies est de 67%, puis devient 79% à 4 copies, et enfin 97% à 8 copies, causant des E/S disque pour accéder au fichier d'échange (swap) et baissant le débit du serveur. Le coût de la duplication des données n'est donc pas négligeable, et n'est de fait pas toujours rentable. Cela corrobore les conclusions de Harji *et al.* [43] et Bathia *et al.* [7], selon lesquelles les performances des serveurs Web dépendent entre autres de leur empreinte mémoire.

La Figure 6.23 montre le meilleur point pour chaque nombre de cœurs, choisi parmi toutes les configurations testées. Un fait remarquable est qu'il n'existe pas de configuration qui soit toujours la meilleure. À 1 et 2 cœurs, les architectures SYMPED et à étages ont de meilleures performances que leurs versions N-COPY, et que toutes les configurations de Knot.

À 4 cœurs, le N-COPY devient rentable pour diminuer le partage de données, et donc la contention sur le bus. Ce n'est cependant pas le cas à 8 cœurs, où la contention sur le bus devient telle que même une stratégie de N-COPY ne suffit pas à améliorer les performances.

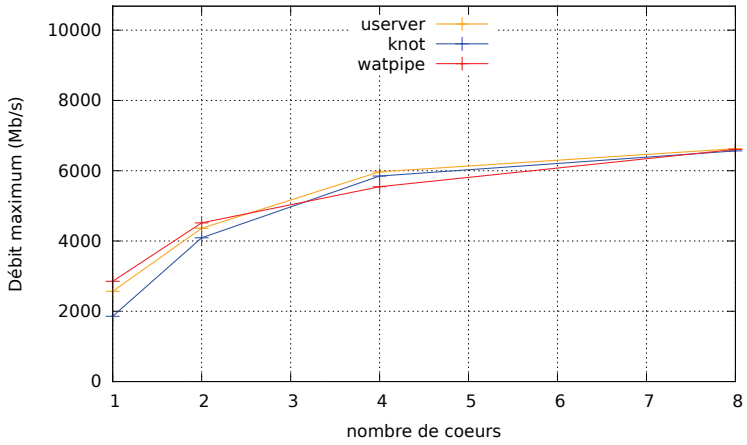

FIGURE 6.23 – Passage à l'échelle des trois serveurs Web.

6.6.3 Bilan

Nous avons montré l'importance de l'architecture matérielle sous-jacente, et plus précisément la hiérarchie mémoire. La prise en compte de celle-ci peut aider à prédire la meilleure configuration. Dans notre cas, à 4 cœurs par exemple, nous savons que le dernier niveau de cache est partagé uniquement par 2 cœurs. Nous dirigeons alors notre stratégie de N-COPY vers 2 copies du serveur, chacune tirant parti d'un cache L2 complet. Nos expériences montrent que cette solution est une bonne piste pour limiter les communications entre les caches de dernier niveau et vers la mémoire.

6.7 Discussion

Nous discutons dans cette section de la portée de nos résultats. Nous nous posons tout d'abord la question de savoir si nos observations seraient valides sur des architectures NUMA. Ce point paraît crucial car ces architectures remplacent le bus sur lequel nous observons une saturation par une interconnexion rapide. Dans un deuxième temps, nous étudions le cas des charges dynamiques. En effet, il est intéressant de savoir si l'ajout de calculs lors des requêtes peut diminuer la pression sur la mémoire, et ainsi montrer un meilleur passage à l'échelle.

6.7.1 Transposition aux architectures NUMA

Les architectures NUMA ne sont pas munies de bus, mais d'une interconnexion entre les nœuds NUMA, ayant bien souvent un plus gros débit. Nous questionnons donc ici la présence sur ce type d'architecture d'une limitation similaire à celle observée avec les serveurs Web sur les architectures UMA. Notre but ici est de déterminer si des expérimentations de serveurs Web similaires à celles conduites dans ce chapitre auraient du sens sur les architectures NUMA, et si cela suffirait à lever la limitation observée sur le bus mémoire en UMA.

Pour cela, nous utilisons le banc d'essai mémoire de Corey [14] comme une analyse au pire cas du passage à l'échelle d'un serveur Web. Dans ce banc d'essai, chaque cœur alloue 1Go dans sa mémoire locale, et le lit séquentiellement. Il n'y a aucun partage de données dans les zones mémoires lues. Vu l'absence de partage entre les cœurs et la grande taille des zones

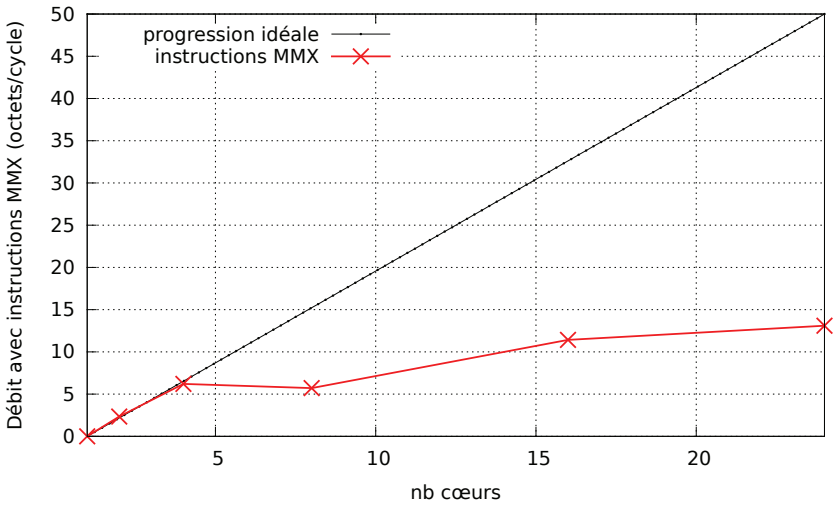

FIGURE 6.24 – Débit du banc d'essai mémoire avec instructions MMX selon le nombre de cœurs

allouées, le débit de ce banc d'essai devrait passer à l'échelle idéalement, du moins jusqu'à ce que le débit maximal vers la mémoire soit atteint.

Nous expérimentons sur des machines NUMA 24 cœurs (2x12 cœurs), et mesurons le débit mémoire en termes du nombre d'octets accédés par cycle processeur. Ces machines sont dotées de 48Go de RAM (2x24Gb), de ce fait nous n'utilisons jamais pleinement la mémoire centrale. Nous testons deux méthodes de lecture mémoire. La première utilise directement l'instruction assembleur `movq` (tirée du jeu d'instructions MMX) pour les lectures mémoire. La deuxième méthode est écrite en C, et laisse au compilateur le soin d'optimiser les accès mémoire au mieux. La première méthode (instruction MMX) correspond au pire cas, car elle est censée stresser plus la mémoire. Elle nous sert ici pour représenter une borne maximum aux capacités mémoires effectives de la machine. A l'inverse, la deuxième méthode est plus représentative des serveurs Web et des applications en général car il est rare qu'une application embarque du code assembleur pour effectuer ses accès mémoire.

Les Figures 6.24 et 6.25 montrent le débit mémoire, respectivement avec et sans instructions assembleur MMX pour les accès mémoire. On remarque trois faits importants. Tout d'abord, le passage à l'échelle du test avec instructions MMX diminue fortement après 4 cœurs, menant à une accélération de seulement 4.8 à 24 cœurs. Cela montre que l'augmentation du nombre de cœurs perturbe les accès mémoire indépendants, même en NUMA. En effet, ce banc d'essai n'atteint pas la limite physique du débit mémoire, car le débit continue d'augmenter avec le nombre de cœurs (il stagnerait si une limite physique était atteinte). Deuxièmement, on note que le débit avec des instructions C est nettement inférieur au débit obtenu avec des instructions MMX (à 24cœurs le débit avec MMX est 20 fois supérieur qu'avec des instructions C). Cela est un comportement attendu, et permet au banc d'essai avec instructions C d'afficher un meilleur passage à l'échelle jusqu'à 12 cœurs. Cependant, et c'est notre troisième remarque, le passage à l'échelle de ce banc d'essai n'est pas optimal, et l'accélération observée est seulement de 12.1 entre 1 et 24 cœurs.

Encore une fois nous sommes loin d'un passage à l'échelle linéaire, alors qu'il s'agit d'une application totalement parallèle, sans partage de données. Nous n'atteignons pas la limite matérielle du débit mémoire, et nous ne dépassons jamais la capacité mémoire (pas d'accès

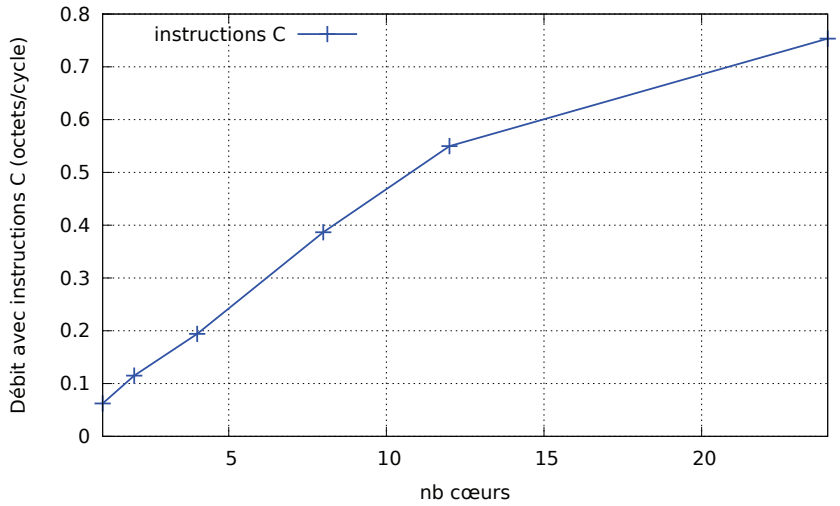

FIGURE 6.25 – Débit du banc d'essai mémoire avec instructions C classiques selon le nombre de cœurs

au fichier d'échange sur disque dur). L'explication la plus plausible est donc une saturation de l'interconnexion due au protocole de cohérence mémoire.

Il semblerait donc que nous faisions face à une limite inhérente aux architectures multi-cœurs en général. L'augmentation des temps d'accès mémoire étant inhérente à toute machine multi-cœur à mémoire cohérente.

6.7.2 Profils de charges dynamiques

Nous avons rencontré dans ce chapitre une limitation dûe à une trop grande pression sur les accès mémoire. Nous voulons déterminer ici si l'ajout de calculs côté serveur pourrait réduire cette pression, et ainsi nous aider à contourner cette limitation. Pour cela, nous expérimentons avec des charges dynamiques.

La Figure 6.26 montre le passage à l'échelle du serveur Web Apache soumis à la charge Support de SpecWeb2005. Nous utilisons ici 8 clients esclaves physiques, et 1 client primaire contenant également le serveur BESIM. Bien que les performances brutes (en termes de débit) augmentent avec le nombre de cœurs utilisés, l'accélération est loin d'être linéaire. En effet, le débit est triplé (x3.04) à 4 cœurs, et seulement multiplié par 5.26 à 8 cœurs. Nous retrouvons ici les performances observées par Veal *et al.* [83], qui font un test similaire avec le même serveur Web, la même charge et un processeur de même famille. La principale différence avec cette étude réside dans le matériel réseau utilisé. Veal et Foong utilisent 4 cartes e1000 1Gb/s, alors que nous utilisons pour cette expérience uniquement nos cartes Myrinet 10Gb/s. Lors de leur étude, Veal avaient conclu que les performances du serveur étaient limitées par le bus d'adresses. Nous nous contentons donc ici de corroborer ce résultat en comparant directement nos résultats aux leurs. Nos résultats présentés à la Figure 6.26 reproduisent presque exactement ceux présentés par Veal. Étant donné la similarité du comportement que nous observons, nous estimons que les mêmes conclusions s'appliquent, et donc que le manque de passage à l'échelle est dû à une saturation du bus d'adresses.

Nous sommes capables de reproduire exactement le même comportement que celui qu'ils ont observé sur la charge Support. Cela nous permet de pousser l'étude plus loin et de remarquer une différence sur les résultats observables sur la variante Ecommerce de la charge

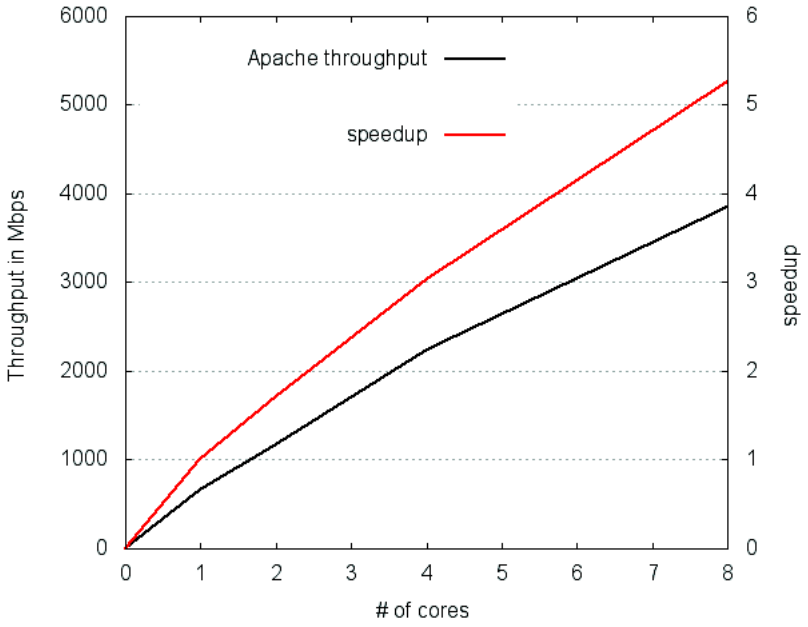

FIGURE 6.26 – Passage à l'échelle d'Apache avec la charge SpecWeb2005 Support.

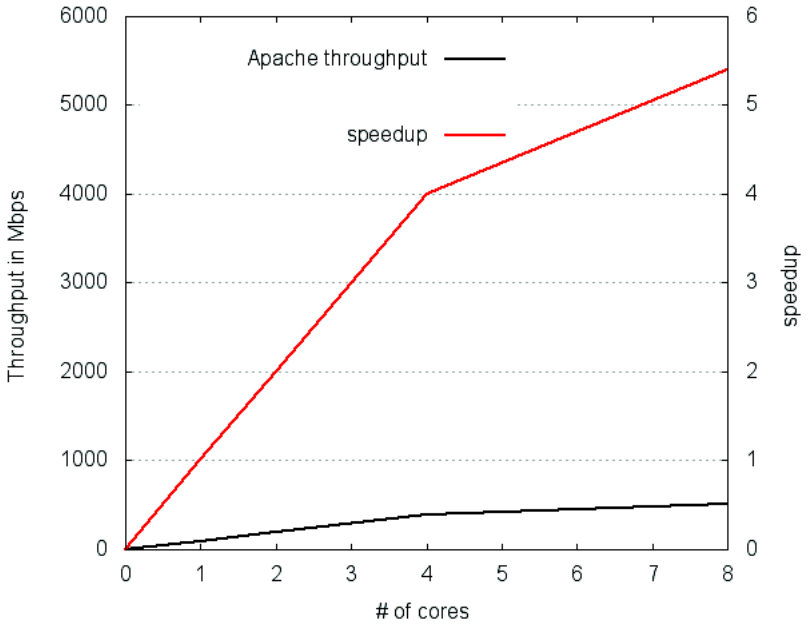

FIGURE 6.27 – Passage à l'échelle d'Apache avec la charge SpecWeb2005 Ecommerce.

SpecWeb2005. On sait que cette variante occasionne bien plus de traitements sur le serveur. Il est donc normal d'observer sur la Figure 6.27 que d'une part le débit est bien plus faible que la variante Support, et que d'autre part le serveur passe à l'échelle parfaitement jusqu'à 4 cœurs. Cela peut s'expliquer par le fait que les traitements supplémentaires nécessitent suffisamment de temps CPU pour masquer et amortir le coût des accès mémoire jusqu'à 4 cœurs. Cependant, malgré cela, un problème de passage à l'échelle réapparaît à 8 cœurs. Comme pour la charge Support, il s'agit ici d'une saturation du bus d'adresses, due aux nombreuses invalidations de données dans les caches. Ces expérimentations nous permettent d'affirmer que l'ajout de calculs côté serveur ne suffit pas à contourner la limitation observée sur le bus d'adresses. Les traitements CPU supplémentaires de la charge Ecommerce permettent un meilleur passage à l'échelle d'Apache jusqu'à 4 cœurs, par rapport à la charge Support. Ces traitements CPU ont donc l'effet escompté dans une certaine mesure. Cependant, la limitation liée au bus d'adresses est toujours présente à 8 cœurs. Cela nous amène à penser que les conclusions tirées dans ce chapitre sont valables autant sur des charges statiques que dynamiques.

6.8 Conclusion

Dans ce chapitre, nous avons comparé les performances de trois implantations de serveurs Web représentatives de trois modèles de programmation, en multi-cœur. Nous avons choisi μserver pour représenter le modèle événementiel, Watpipe pour le modèle à étages, et Knot pour le modèle de thread par connexion. Nous utilisons une charge statique basée sur SpecWeb99, injectée en boucle semi-ouverte. Nous avons montré grâce à cela de grandes variations inattendues dans les écarts de performances entre les différentes implantations. Notamment, à 8 cœurs, toutes les implantations souffrent de la même façon d'un problème de passage à l'échelle. Nous identifions précisément ce problème, qui est une saturation du bus d'adresses, dues aux nombreuses communications entre les cœurs. Pour pallier ce problème et réduire le partage entre les cœurs, nous proposons une approche à base de N-COPY. Nous montrons une amélioration de 13% à 4 cœurs. Cependant malgré cela, le problème de passage à l'échelle persiste à 8 cœurs.

D'autres solutions pourraient être envisagées pour réduire le partage de données entre les cœurs. Premièrement, des expérimentations avec des cartes réseau Ethernet 10Gb/s virtualisables seraient intéressantes. En effet, cela permettrait d'associer un flot réseau par cœur. Ensuite, des expérimentations sur architectures NUMA seraient plus que bienvenues. Cependant, les travaux de Gaud *et al.* [38] sur machine NUMA montrent qu'Apache et PHP ne passent pas non plus idéalement à l'échelle avec le nombre de cœurs. Nous pensons qu'une bonne solution potentielle peut s'implanter au niveau matériel, en modifiant le comportement de la mémoire. Notamment, nous attendons un comportement radicalement différent de la part des nouvelles architectures many-cœurs, telles que le prototype SCC présenté par Intel [82]. On peut toutefois noter que ce type d'architectures demandera des changements radicaux dans les couches logicielles s'il vient à être adopté, du fait du changement des hypothèses sur la cohérence mémoire fournie par le matériel.

Conclusion

Les processeurs ont connu une évolution sans précédent avec l'arrivée des architectures multi-cœur. Ce type d'architectures permet de maintenir l'augmentation des performances connue jusqu'alors, tout en apportant une solution aux difficultés de dissipation thermique et de complexité des processeurs. Pour garder des performances compétitives, les applications doivent donc désormais tirer parti du parallélisme exposé par les architectures multi-cœur. Comme nous l'avons vu au Chapitre 1, la tendance est à l'augmentation du nombre de cœurs présents sur une puce, et donc du degré de parallélisme des processeurs.

Cette évolution matérielle a des répercussions importantes sur les couches logicielles à différents niveaux, car la parallélisation complexifie grandement la programmation. Un premier niveau affecté est celui des modèles de programmation. En effet, ces modèles doivent permettre et faciliter la création d'applications parallèles. Les modèles classiques (événements et threads) suffisent pour exprimer une concurrence de type partage de temps (*time-sharing*) au sein d'un cœur. Toutefois, leurs limitations inhérentes les rendent insuffisants pour exploiter efficacement le parallélisme matériel imposé par les architectures multi-cœur (*space-sharing*). En effet, les threads demandent l'utilisation de primitives de synchronisation coûteuses et complexes pour gérer les accès aux données partagées ; et le modèle événementiel classique ne s'exécute que sur un seul flot d'exécution (donc un seul cœur).

Le second aspect profondément affecté par les multi-cœur est celui des supports d'exécution. Une fois les tâches d'une application définies via un modèle de programmation, il faut les exécuter efficacement sur les cœurs disponibles. Ce point concerne aussi bien les systèmes d'exploitation que les environnements d'exécution utilisateurs. La difficulté est de tenir compte de l'architecture matérielle, et notamment de la hiérarchie mémoire, lors de la répartition des tâches sur les cœurs disponibles.

Bilan des contributions

Les serveurs de données sont des applications intrinsèquement parallèles. En effet, ils servent des requêtes indépendantes de manière concurrente. On s'attend donc à un passage à l'échelle proche de l'idéal en multi-cœur. Il se trouve que ce n'est pas le cas dans la pratique [83]. Nos contributions ont pour objectif de fournir des pistes d'améliorations des performances et surtout du passage à l'échelle avec le nombre de cœurs des serveurs de données. À cette fin, nous avons contribué dans deux directions principales. Tout d'abord nous avons analysé les possibilités de passage à l'échelle d'un environnement d'exécution événementiel pour le multi-cœur. Dans un second temps, nous avons mené une comparaison quantitative des performances en multi-cœur des trois modèles de programmation les plus utilisés pour concevoir des serveurs de données. Nous passons en revue ci-dessous les principales propositions que nous avons apporté au sein de chacune de ces directions.

Optimisations d'une plateforme événementielle multi-cœur

Dans cette partie, nous avons travaillé sur l'environnement événementiel multi-cœur Libasync-smp [93], qui est l'implantation de référence du modèle événementiel avec coloration d'événements. Nous avons tout d'abord créé deux bancs d'essai représentatifs des schémas d'applications événementielles classiques. Ces bancs d'essai nous ont permis d'exposer des limitations dans les performances et le passage à l'échelle de Libasync-smp. Ces limitations sont présentes à deux niveaux : au niveau de la gestion de la mémoire, et des communications inter-cœurs. Le faux-partage dans les données créées et manipulées par Libasync-smp constitue un frein majeur au passage à l'échelle des performances des applications utilisant cet environnement. De plus, nous avons montré les coûts significatifs d'endormissement et de réveil des threads exécutant les traitants, ainsi que les coûts de synchronisation liés à l'enregistrement des événements.

Pour palier ces limitations, nous avons proposé une série de quatre optimisations. Deux de ces optimisations opèrent sur la gestion de la mémoire. En effet, nous avons mis en place un bourrage des structures statiques internes à Libasync-smp. Nous avons ensuite proposé un nouvel allocateur mémoire, EMA. EMA est fortement couplé à l'environnement d'exécution, et s'appuie sur le mécanisme de coloration des événements pour éliminer le faux-partage et limiter les coûts de synchronisation. EMA permet ainsi un passage à l'échelle idéal là où d'autres allocateurs mémoires de l'état de l'art ne le permettent pas.

Nous avons mis en œuvre deux autres optimisations liées aux communications inter-cœurs. Nous avons proposé un mécanisme d'attente active raisonnée pour réduire les coûts de réveil des threads de Libasync-smp. Ce mécanisme s'adapte aux besoins de l'application, et permet une grande réactivité lorsque c'est nécessaire, tout en laissant les threads s'endormir lorsqu'ils sont moins sollicités. Enfin, nous avons amorti les coûts d'enregistrement d'événements au moyen d'un enregistrement par lots. Ce mécanisme permet de rentabiliser les coûts des primitives de synchronisation.

Enfin, nous avons évalué et validé ces optimisations sur des applications réelles, à savoir un serveur Web, et la suite d'applications MapReduce de Phœnix. Nous avons détaillé les gains apportés par chaque optimisation. Notre environnement d'exécution optimisé améliore les performances du serveur Web jusqu'à 111% par rapport à la version originale de Libasync-smp, et jusqu'à 66% par rapport à la bibliothèque Phœnix.

Analyse comparative d'architectures de serveurs Web en multi-cœur

Comme nous l'avons vu, plusieurs modèles de programmation concurrente existent. Nous avons tenté de déterminer lequel est le plus à même de fournir les serveurs de données offrant les meilleures performances et le meilleur passage à l'échelle en multi-cœur. Pour cela, nous avons mené une comparaison quantitative des performances atteintes par différents modèles de programmation, en multi-cœur. Plus précisément, nous avons étudié les trois modèles de programmation les plus utilisés pour mettre en œuvre des serveurs de données : threads, événements et étages. Pour cela, nous avons choisi trois implantations efficaces de serveurs Web : Knot, μserver et Watpipe, chacune représentative d'un modèle de programmation.

Pour mettre en place une comparaison équitable, nous avons d'abord testé chaque serveur indépendamment. De cette manière nous avons obtenu les configurations menant aux meilleures performances de chacun des serveurs étudiés. Nous avons retrouvé à 1 cœur les tendances observées par de précédentes études [64, 43], à savoir que le serveur threadé a de moins bonnes performances que les serveurs événementiel et à étages. Nous avons cependant mis en avant que ces tendances évoluent différemment lorsque le nombre de cœurs augmente.

Plus précisément, le serveur de threads devient meilleur que les deux autres serveurs étudiés à 4 cœurs. De plus, nous avons observé qu'à 8 cœurs toutes les implantations ont les mêmes performances.

Nous avons montré que le multi-cœur apporte de nouveaux éléments dans les performances des serveurs de données, et que la transposition à 8 cœurs des observations faites précédemment à 1 cœur n'est pas valable. Un profilage approfondi nous a permis d'isoler les causes de ce comportement. Il s'agit d'une saturation du bus mémoire. Cette saturation est due à des communications inter-cœurs induites par le protocole de cohérence de caches.

Nous avons alors étudié pour chaque serveur des déploiements N-COPY, pour réduire le partage de données et les effets de cohérence de caches associés. Nous avons montré que les meilleurs placements N-COPY sont ceux qui tiennent compte de la hiérarchie mémoire, et notamment des caches partagés. De tels déploiements permettent notamment d'augmenter de 13% les performances des serveurs événementiel et à étages à 4 cœurs, replaçant ainsi ces implantations au même niveau que le serveur threadé. Toutefois, le phénomène de saturation du bus persiste à 8 cœurs, empêchant le passage à l'échelle des serveurs testés.

Au long de ces deux études, nous avons montré que les performances et le passage à l'échelle des serveurs de données peut être augmentée de deux manières. On peut tout d'abord réduire le partage des données entre les cœurs, pour éviter les communications inter-cœurs coûteuses et indésirables, via une allocation mémoire efficace ou un déploiement N-COPY. Une autre possibilité est de diminuer les coûts de synchronisation et de communication inter-cœurs.

Un aspect important de notre méthodologie se base sur l'observation des compteurs de performances matériels. Cela permet d'une part d'analyser la cause des baisses de performances rencontrées. D'autre part, nous utilisons également ces compteurs pour valider nos optimisations. Ces compteurs jouent un rôle central dans l'analyse de performances et ont été fortement développés ces dernières années.

Enfin nous constatons que les modèles de programmation actuels et leurs implantations ne permettent pas encore un passage à l'échelle idéal en multi-cœur. Nous avons observé que ces difficultés de passage à l'échelle sont majoritairement dues au partage implicite de données. À l'heure actuelle, ce type de partage de données n'est géré par aucun modèle de programmation, et il revient aux programmeurs d'appliquer de bonnes pratiques afin de permettre aux applications de passer à l'échelle de manière satisfaisante.

Perspectives

Les travaux menés dans cette thèse ouvrent des pistes pour de futurs travaux, aussi bien des travaux d'extensions à court terme que des études plus profondes à long terme.

Tout d'abord, nous pouvons citer l'adaptation dynamique des valeurs fixées au sein de notre environnement d'exécution, mentionnées au Chapitre 5. Nous fixons ces pour déterminer la durée du seuil utilisé pour l'attente active raisonnée, ainsi que la taille du tampon lié à l'enregistrement par lots. Notre environnement pourrait bénéficier d'un mécanisme qui adapterait ces valeurs dynamiquement en fonction de l'application.

Une autre piste est d'étudier l'intégration d'autres mécanismes au sein de notre environnement d'exécution. En effet, pour notre étude, nous avons désactivé les mécanismes de ramasse-miettes et de vol de tâches de Libasync-smp car ils dégradent fortement les performances des applications étudiées. Il serait intéressant d'étudier les possibilités de mécanismes

de ramasse-miettes et vol de tâches efficaces en multi-cœur, et leur intégration avec nos travaux.

Nous avons également discuté d'extensions possibles à notre comparaison de modèles de programmation. Dans ce sens, une continuité possible est de renforcer l'isolation que nous proposons via l'utilisation de machines virtuelles. C'est à dire adopter une solution N-COPY où au lieu de processus indépendants, chaque copie consisterait en une machine virtuelle indépendante. Cela pourrait être une piste pour réduire encore plus le partage de données non voulu, sous réserve d'une bonne isolation au sein de l'hyperviseur.

Si l'on considère des perspectives à plus long terme, on peut envisager une généralisation des travaux réalisés dans cette thèse. Une telle généralisation partirait du constat que les bonnes pratiques utilisées ainsi que les défis rencontrés suivent tous une même philosophie, selon laquelle chaque cœur doit pouvoir opérer sur ses propres données indépendamment des autres cœurs. Cela amène des techniques de programmation totalement décentralisées, similaires à celles des systèmes distribués, mais où la mémoire peut tout de même être partagée entre les cœurs (par le biais des caches notamment). Une approche basée sur le modèle de programmation événementiel semble être une bonne piste pour mettre en œuvre ce type d'applications, du fait que le partage est rendu explicite via le passage de messages. Cependant, nous avons vu qu'il reste tout de même des difficultés dues à un partage implicite des données. Nous pensons donc qu'il faut revoir les abstractions fournies aux programmeurs, dans le sens où elles doivent permettre d'éviter le partage implicite de données, de découper facilement un travail en tâches parallèles, tout en restant suffisamment simples pour rester utilisables.

Bibliographie

[1] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young.
Mach : A new kernel foundation for unix development.
In *USENIX Summer*, pages 93–113, 1986.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur.
Cooperative Task Management Without Manual Stack Management.
In *Proceedings of the the 2002 USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002. USENIX Association.

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

[4] Gaurav Banga and Peter Druschel.
Measuring the Capacity of a Web Server.
In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*, Monterey, CA, USA, 1997. USENIX Association.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schuepbach, and Akhilesh Singhania.
The multikernel : a new OS architecture for scalable multicore systems.
In *Proceedings of the 22nd ACM symposium on Operating systems principles (SOSP)*, October 2009.

[6] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel.
Finding a needle in haystack : facebook's photo storage.
In *OSDI'10*, 2010.

[7] Sapan Bhatia, Charles Consel, and Julia L. Lawall.
Memory-Manager/Scheduler Co-Design : Optimizing Event-Driven Servers to Improve Cache Behavior.
In *Proceedings of the 5th ACM International Symposium on Memory Management (ISMM'06)*, Ottawa, Ontario, Canada, June 2006. ACM Press.

[8] Robert D. Blumofe.
Executing multithreaded programs efficiently.
PhD thesis, Cambridge, MA, USA, Cambridge, MA, USA, 1995.

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou.
Cilk : An Efficient Multithreaded Runtime System.
J. Parallel Distrib. Comput., 37(1) :55–69, 1996.

[10] Robert D. Blumofe and Charles E. Leiserson.
Scheduling multithreaded computations by work stealing.
J. ACM, 46(5) :720–748, 1999.

[11] OpenMP Architecture Review Board.
Openmp.
`http://www.openmp.org`.

[12] 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.
Corey : An Operating System for Many Cores.
In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, USA, December 2008. USENIX Association.

[13] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich.
An Analysis of Linux Scalability to Many Cores.
In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[14] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek.
Reinventing scheduling for multicore systems.
In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, Monte Verità, Switzerland, May 2009.

[15] Apple Inc. Technical Breif.
Grand central dispatch :a better way to do multicore, 2009.
`https://libdispatch.macosforge.org/`.

[16] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee.
Software transactional memory : Why is it only a research toy ?
ACM Queue, 6(5) :46–58, 2008.

[17] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin.
Predicting inter-thread cache contention on a chip multi-processor architecture.
High-Performance Computer Architecture, International Symposium on, 0 :340–351, 2005.

[18] David Chase and Yossi Lev.
Dynamic circular work-stealing deque.
In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*, Las Vegas, Nevada, USA, 2005. ACM.

[19] Benjie Chen and Robert Morris.
Flexible Control of Parallelism in a Multiprocessor PC Router.

In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 2001. USENIX Association.

[20] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, and Chita R. Das.
A Multi-Threaded PIPELINED Web Server Architecture for SMP/SoC Machines.
In *WWW*, 2005.

[21] Intel Corporation.
Tera-scale architecture project.
`http://techresearch.intel.com/articles/Tera-Scale/1421.htm`.

[22] Intel Corporation.
Intel 64 and ia-32 architectures software developer's manual, 2011.
`http://download.intel.com/products/processor/manual/325462.pdf`.

[23] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica.
Wide-area cooperative storage with cfs.
In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, Banff, Alberta, Canada, 2001. ACM Press.

[24] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris.
Event-Driven Programming for Robust Software.
In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002. ACM Press.

[25] M. Daga, A.M. Aji, and Wu chun Feng.
On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing.
In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141 –149, july 2011.

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

[27] Advanced Micro Devices.
Amd64 architecture programmer's manual volume 2 : System programming, 2011.
`http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf`.

[28] Mihai Dobrescu, Norbert Egi, Katerina J. Argyraki, Byung-Gon Chun, Kevin R. Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy.
Routebricks : exploiting parallelism to scale software routers.
In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[29] Doug Lea.
malloc.
`http://gee.cs.oswego.edu/dl/html/malloc.html`.

[30] Ulrich Drepper and Ingo Molnar.
The Native POSIX Thread Library for Linux, February 2005.
`http://www.akkadia.org/drepper/nptl-design.pdf`.

[31] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole.
Exokernel : an operating system architecture for application-level resource management.
In *SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266. ACM, 1995.

[32] P. Felber, E. Rivière andre, W.M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Hohmuth, M. Pohlack, A. Cristal, I. Hur, O.S. Unsal, P. Stenström andm, A. Dragojevic, R. Guerraoui, M. Kapalka, V. Gramoli, U. Drepper, S. Tomic and, Y. Afek, G. Korland, N. Shavit, C. Fetzer, M. Nowack, and T. Riegel.
The velox transactional memory stack.
Micro, IEEE, 30(5) :76 –87, sept.-oct. 2010.

[33] Michael J. Freedman, Eric Freudenthal, and David Mazières.
Democratizing Content Publication with Coral.
In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, USA, 2004. USENIX Association.

[34] Kevin Fu, M. Frans Kaashoek, and David Mazières.
Fast and Secure Distributed Read-Only File System.
In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI'00)*, San Diego, California, 2000. USENIX Association.

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

[36] Daniel F. García and Javier García.
Tpc-w e-commerce benchmark evaluation.
Computer, 36(2) :42–48, 2003.

[37] Fabien Gaud, Sylvain Geneves, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma.
Efficient workstealing for multicore event-driven systems.
In *International Conference on Distributed Computing Systems*, pages 516–525, June 2010.

[38] Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Gilles Muller, and Vivien Quéma.
Optimisations applicatives pour multi-cœurs numa : un cas d'étude avec le serveur web apache.
In *Conférence Française en Systèmes d'Exploitation (CFSE'8)*, May 2011.

[39] Google.
Tcmalloc : Thread-caching malloc.
`http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[40] Andreas Gustafsson.
Threads without the pain.
ACM Queue, 3(9) :34–41, 2005.

[41] Philipp Haller and Martin Odersky.
Actors that unify threads and events.
In *COORDINATION'07 : Proceedings of the 9th international conference on Coordination models and languages*, pages 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[42] A. Harji, P. Buhr, and Tim Brecht.
Comparing high-performance multi-core web-server architectures.
In *5th Annual International Systems and Storage Conference (SYSTOR)*, Haifa, Israel, June 2012.

[43] Ashif Harji.
Performance Comparison of Uniprocessor and Multiprocessor Web Server Architectures.
PhD thesis, University of Warterloo, January 2010.

[44] John L. Hennessy and David A. Patterson.
Computer Architecture - A Quantitative Approach (4. ed.).
Morgan Kaufmann, 2007.

[45] Tom Herbert.
Receivent Packet steering, November 2009.
`http://lwn.net/Articles/361440`.

[46] Tom Herbert.
Receive Flow Steering, April 2010.
`http://lwn.net/Articles/382428`.

[47] Intel Corporation.
Single-chip Cloud Computer.
`http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-architecture-brief.pdf`.

[48] John Jannotti and Kiran Pamnany.
Safe at Any Speed : Fast, Safe Parallelism in Servers.
In *Proceedings of the 2nd USENIX Workshop on Hot Topics in System Dependability (HotDep'06)*, Seattle, Washington, USA, November 2006. USENIX Association.

[49] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi.
Analysis and approximation of optimal co-scheduling on chip multiprocessors.
In *PACT '08 : Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 220–229, New York, NY, USA, 2008. ACM.

[50] Dan Kegel.
The c10k problem, 2006.
`http://www.kegel.com/c10k.html`.

[51] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn.
Using os observations to improve performance in multicore systems.
IEEE Micro, 28(3) :54–66, 2008.

[52] Maxwell Krohn.
Building Secure High-Performance Web Services with OKWS.

In *Proceedings of the 2004 USENIX Annual Conference*, Boston, MA, USA, June 2004. USENIX Association.

[53] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek.
Events Can Make Sense.
In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007. USENIX Association.

[54] Alexey Kukanov and Michael J. Voss.
The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.
Intel Technology Journal, 11, 2007.

[55] James Larus and Christos Kozyrakis.
Transactional memory.
Commununication of the ACM, 51(7) :80–88, 2008.

[56] Edward A. Lee.
The Problem with Threads.
IEEE Computer, 39(5), 2006.

[57] J. Liedtke.
On micro-kernel construction.
In *SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250. ACM, 1995.

[58] David Mazières.
A Toolkit for User-Level File Systems.
In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, 2001. USENIX Association.

[59] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel.
Separating Key Management From File System Security.
In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, Kiawah Island, South Carolina, USA, December 1999. ACM Press.

[60] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek.
The click modular router.
In *SOSP '99 : Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231. ACM, 1999.

[61] David Mosberger and Tai Jin.
Httperf, a tool for measuring web server performance.
First Workshop on Internet Server Performance, 26 :31–37, June 1998.

[62] John K. Ousterhout.
Why threads are a bad idea (for most purposes).
Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[63] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel.
Flash : An efficient and portable Web server.
In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, California, USA, June 1999.

[64] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, and Amol Shukla.
Comparing the performance of web server architectures.
In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07)*, Lisbon, Portugal, June 2007. ACM Press.

[65] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris.
Locating Cache Performance Bottlenecks Using Data Profiling.
In *EuroSys*, 2010.

[66] Niel Provos.
libevent — An Event Notification Library, 2008.
http ://monkey.org/ provos/libevent/.

[67] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis.
Evaluating mapreduce for multi-core and multiprocessor systems.
In *HPCA '07 : Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[68] Rick Jones.
Netperf hompage.
`http://www.netperf.org/netperf/`.

[69] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang.
Enabling Scalability and Performance in a Large Scale CMP Environment.
In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys'07)*, Lisbon, Portugal, June 2007. ACM Press.

[70] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter.
Open Versus Closed : a Cautionary Tale.
In *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI'06)*, San Jose, CA, May 2006. USENIX Association.

[71] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan.
Larrabee : a many-core x86 architecture for visual computing.
ACM Trans. Graph., 27(3) :1–15, 2008.

[72] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar.
Hass : a scheduler for heterogeneous multicore systems.
SIGOPS Operating Systems Review, 43(2) :66–75, 2009.

[73] SPEC (Standard Performance Evaluation Corporation).
Specweb 2005.
`http://www.spec.org/web2005/`.

[74] SPEC (Standard Performance Evaluation Corporation).
Specweb 2009.
`http://www.spec.org/web2009/`.

[75] SPEC (Standard Performance Evaluation Corporation).
Specweb99.
`http://www.spec.org/web99/`.

[76] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan.
Chord : a Scalable Peer-to-peer Lookup Protocol for Internet Applications.
IEEE/ACM Transactions on Networking, 11(1) :17–32, 2003.

[77] Jeremy Stribling, Jinyang Li, Isaac G. Councill, M. Frans Kaashoek, and Robert Morris.
OverCite : A Distributed, Cooperative CiteSeer.
In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, USA, May 2006. USENIX Association.

[78] Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen.
Fast switching of threads between cores.
SIGOPS Oper. Syst. Rev., 43(2) :35–45, 2009.

[79] Herb Sutter and James Larus.
Software and the concurrency revolution.
ACM Queue, 3(7) :54–62, 2005.

[80] David Tam, Reza Azimi, and Michael Stumm.
Thread clustering : sharing-aware scheduling on smp-cmp-smt multiprocessors.
In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, volume 41, pages 47–58, New York, NY, USA, 2007. ACM.

[81] J. Torrellas, H. S. Lam, and J. L. Hennessy.
False Sharing and Spatial Locality in Multiprocessor Caches.
IEEE Transactions on Computers, 43(6) :651–663, 1994.

[82] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas.
Light-weight communications on intel's single-chip cloud computer processor.
SIGOPS Oper. Syst. Rev., 45(1) :73–83, February 2011.

[83] Bryan Veal and Annie Foong.
Performance Scalability of a Multi-Core Web Server.
In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems (ANCS '07)*, Orlando, FL, USA, December 2007. ACM Press.

[84] Robert von Behren, Jeremy Condit, and Eric A. Brewer.
Why events are a bad idea (for high-concurrency servers).
In *HOTOS'03 : Proceedings of the 9th USENIX workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[85] Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer.
Capriccio : Scalable threads for internet services.
In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*, Bolton Landing, New York, USA, October 2003. ACM Press.

[86] Ivan Voras and Mario Zagar.
Characteristics of multithreading models for high-performance io driven network applications.
AFRICON 2009, 2009.

[87] Michael Walfish, Mythili Vutukuru, Hari Balakrishnan, David Karger, and Scott Shenker.
DDoS Defense By Offense.
In *Proceedings of the 2006 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Pisa, Italy, September 2006. ACM Press.

[88] Matt Welsh and David Culler.
Adaptive overload control for busy internet servers.
In *USITS'03 : Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[89] Matt Welsh, David Culler, and Eric Brewer.
SEDA : An architecture for well-conditioned scalable internet services.
In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP 2001)*, Banff Alberta Canada, October 2001. ACM Press.

[90] Robert W. Wisniewski and Bryan Rosenburg.
Efficient, unified, and scalable performance monitoring for multiprocessor operating systems.
In *Supercomputing*, 2003.

[91] Wolfram Gloger.
ptmalloc.
`http://www.malloc.de/en/`.

[92] Wm. A. Wulf and Sally A. McKee.
Hitting the memory wall : implications of the obvious.
SIGARCH Comput. Archit. News, 23(1) :20–24, March 1995.

[93] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and M. Frans Kaashoek.
Multiprocessor Support for Event-Driven Programs.
In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, TX, USA, June 2003. USENIX Association.

[94] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova.
Addressing shared resource contention in multicore processors via scheduling.
In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

Résumé

Cette thèse traite des performances des serveurs de données en multi-cœur. Plus précisément nous nous intéressons au passage à l'échelle avec le nombre de cœurs. Dans un premier temps, nous étudions le fonctionnement interne d'un support d'exécution événementiel multi-cœur. Nous montrons tout d'abord que le faux-partage ainsi que les mécanismes de communications inter-cœurs dégradent fortement les performances et empêchent le passage à l'échelle des applications. Nous proposons alors plusieurs optimisations pour pallier ces comportements. Dans un second temps, nous comparons les performances en multi-cœur de trois serveurs Web, chacun représentatif d'un modèle de programmation. Nous remarquons que les différences de performances observées entre les serveurs varient lorsque le nombre de cœurs augmente. Après une analyse approfondie des performances observées, nous identifions la cause de la limitation du passage à l'échelle des serveurs étudiés. Nous présentons une proposition ainsi qu'un ensemble de pistes pour lever cette limitation.

Mots-clés. Architectures multi-cœur, Performance des serveurs de données, Programmation événementielle, Gestion de la mémoire, Serveurs Web, Analyse de performance, Modèles de programmation

Abstract

This thesis studies the performances of data servers on multicores. More precisely, we focus on the scalability with the number of cores. First, we study the internals of an event-driven multicore runtime. We demonstrate that false sharing and inter-core communications hurt performances badly, and prevent applications from scaling. We then propose several optimisations to fix these issues. In a second part, we compare the multicore performances of three Webservers, each reprensentative of a programming model. We observe that the differences between each server's performances vary as the number of cores increases. We are able to pinpoint the cause of the scalability limitation observed. We present one approach and some perspectives to overcome this limit.

Keywords. Multicore architectures, Data servers performances, Event-driven programming, Memory management, Webservers, Performance analysis, Programming models