Communication inter-cœurs optimisée pour le parallélisme de flux.

Thomas Preud'Homme

THÈSE

présentée devant

L'UNIVERSITÉ DE PARIS 6

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE PARIS 6

Mention INFORMATIQUE

Équipe d'accueil : REGAL, Paris

École doctorale ED130 EDITE

soutenue devant la commission d'examen par

Thomas PREUD'HOMME

le 10 juin 2013

Communication inter-cœurs optimisée pour le parallélisme de flux.

Directeur : Bertil Folliot (Professeur)
Encadrants : Julien Sopena (Maître de Conférences)
Encadrants : Gaël Thomas (Maître de Conférences)

Composition du jury

Rapporteurs	Christophe CÉRIN	Professeur au LIPN
	Jean Frédéric MYOUPO	Professeur au MIS
Examinateurs	Bertrand LE CUN	Maître de Conférences au PRiSM
	Albert COHEN	Directeur de recherche à l'INRIA
	Jean-Luc LAMOTTE	Professeur au LIP6
	Bertil FOLLIOT	Professeur au LIP6

LIP6 – Laboratoire d'Informatique de Paris VI

Remerciements

Cette thèse n'aurait pas pu être possible sans le concours de nombreuses personnes que je tiens à remercier.

En premier lieu, je souhaite remercier ma fiancée, Yan Zhang, pour tout le soutien moral qu'elle m'a apporté au jour le jour. J'admire également sa patience à m'écouter décrire les difficultés que je rencontrais alors qu'elle en traversait déjà elle même. Enfin, je lui suis très reconnaissant d'avoir cru en moi pendant tout ce temps et de m'encourager dans toutes mes initiatives.

Je dois également un grand merci à Julien Sopena pour ses nombreux conseils, en particulier en terme de méthodes de travail, et pour avoir adapté son encadrement à ma personnalité. Il a su encourager ma curiosité tout en maintenant mon attention canalisée vers l'accomplissement de cette thèse, m'évitant ainsi de me disperser. Ce fut un grand plaisir de travailler avec lui.

Cette thèse n'aurait pas été commencée sans l'énergie et l'enthousiasme de Gaël Thomas. Sa passion pour la recherche en système est contagieuse : sa joie d'enseigner le système m'a convaincu et les nombreuses discussions que nous avons eu m'ont convaincu de faire une thèse.

Mes remerciements vont aussi à Bertil Folliot pour son encadrement éclairé. Il a été à l'origine de toutes les étapes importantes de cette thèse, et en particulier dans l'acceptation d'un article à SBAC-PAD qui fut un tournant pour la thèse.

Je remercie sincèrement Christophe Cerin, professeur au LIPN, et Jean Frédéric Myoupo, professeur à l'université de Picardie Jules-Verne, d'avoir accepté de prendre de leur temps pour rapporter ma thèse. Leurs excellentes remarques ont permis d'améliorer substantiellement le manuscrit.

Je remercie également Bertrand Le Cun, maître de conférence au PRiSM, Albert Cohen, directeur de recherche à l'INRIA, et Jean-Luc Lamotte, professeur au LIP6, d'avoir accepté d'être dans mon jury et en particulier Albert Cohen dont les travaux ont inspiré les miens.

Toutes ces années de thèse n'auraient pas été aussi agréables sans la présence des nombreux doctorants, ingénieurs et stagiaires avec qui j'ai eu l'occasion de travailler. Nos nombreuses discussions m'ont permis d'apprendre beaucoup d'informations que je n'aurais pas appris autrement et ont souvent été très amusantes. Je remercie en particulier Lamia Benmouffok et Mathieu Valero pour leur soutien moral tout au long de cette thèse, Clément Desmoulins, Sergey Legtchenko, Nicolas Pierron, Lokesh Gidra, Florian David, Harris Bakiras, Maxime Lorrillere, Maxime Veron et Brice Berna pour les discussions techniques et geeks que nous avons eu, Nicolas Hidlago, Erika Rosas, Pierpaolo Cincilla, Suman Saha, Marek Zawirski et Masoud Saeida Ardekani pour le fun que vous m'avez apporté, Pierre Sutra pour avoir souvent été mon camarade de nuit blanche et Ruijing Hu pour m'avoir appris à être rigoureux dans mes réponses, Mathieu Sassolas et Maximilien Colange pour nos passionantes discussions sur l'histoire et les sciences en particulier.

Je souhaite remercier les ingénieurs système et le personnel administratif du LIP6 qui ont permis à ma thèse de se dérouler sans accrocs techniques ou administratifs. Je suis particulièrement reconnaissant à Nicolas Gibelin, Jean-Luc Mounier, Thierry Lanfroy, Véronique Varennes, Marguerite Sos et Catherine Mercier pour les nombreux services rendus.

Ma gratitude va également à toutes les personnes m'ayant aidé dans la relecture de mon manuscrit ou dans son amélioration : Matteo Cypriani, Mathilde Sopena, Pierre Sens.

Enfin, je remercie chaleureusement ma famille et mes amis pour l'intérêt qu'ils ont porté dans ma thèse et en particulier dans son bon déroulement. Leurs nombreux messages d'encouragements ont été très stimulants et réconfortants.

Résumé

Parmi les différents paradigmes de programmation parallèle, le parallélisme de flux présente l'avantage de conserver la séquentialité des algorithmes et d'être ainsi applicable en présence de dépendances de données. De plus, l'extension de calcul par flux pour OpenMP proposée par Pop et Cohen permet de mettre en œuvre cette forme de parallélisme sans requérir de réécriture complète du code, en y ajoutant simplement des annotations. Cependant, en raison de l'importance de la communication nécessaire entre les cœurs de calcul, les performances obtenues en suivant ce paradigme sont très dépendantes de l'algorithme de communication utilisé. Or l'algorithme de communication utilisé dans cette extension repose sur des files gérant plusieurs producteurs et consommateurs alors que les applications mettant en œuvre le parallélisme de flux fonctionnent principalement avec des chaînes de communication linéaires.

Afin d'améliorer les performances du parallélisme de flux mis en œuvre par l'extension de calcul par flux pour OpenMP, cette thèse propose d'utiliser, lorsque cela est possible, un algorithme de communication plus spécialisé nommé BatchQueue. En ne gérant que le cas particulier d'une communication avec un seul producteur et un seul consommateur, BatchQueue atteint des débits jusqu'à deux fois supérieurs à ceux des algorithmes existants. De plus, une fois intégré à l'extension de calcul par flux pour OpenMP, l'évaluation montre que BatchQueue permet d'améliorer l'accélération des applications jusqu'à un facteur 2 également. L'étude montre ainsi qu'utiliser des algorithmes de communication spécialisés plus efficaces peut avoir un impact significatif sur les performances générales des applications mettant en œuvre le parallélisme de flux.

Abstract

Among the various paradigms of parallelization, pipeline parallelism has the advantage of maintaining sequentiality of algorithms, thus being applicable in case of data dependencies. More over, the stream-computing extension for OpenMP proposed by Pop and Cohen allows to apply this form of parallelization without needing a complete rewrite of the code, by simply adding annotations to it. However, due to the *importance* of the communication needed between the cores, the performances obtained by following this paradigm depends very much on the communication algorithm used. Yet, the communication algorithm used in this extension relies on queues that can handle several producers and consumers while applications using pipeline parallelism mainly works with linear communication chains.

To improve the performances of pipeline parallelism implemented by the stream-computing extension for OpenMP, this thesis propose to use, whenever possible, a more specialized communication algorithme called BatchQueue. By only handling the special case of a communication with one producer and one consumer, BatchQueue can reach throughput up to two time faster than existing algorithms. Furthermore, once integrated to the stream-computing extension for OpenMP, the evaluation shows that BatchQueue can improve speedup of application up to a factor 2 as well. The study thus shows that using a more efficient specialized communication algorithm can have a significant impact on overall performances of application implementing pipeline parallelism.

Table des matières

Table des figures

Chapitre 1

Introduction

Sommaire

1.1 Contexte

En 1965, Gordon Earle Moore[1] remarqua [M+98] que sur les 6 dernières années le nombre de transistors par circuit intégré avait doublé tous les deux ans et extrapola qu'une telle tendance se poursuivrait pendant encore 10 ans. Loin de se tromper, sa conjecture se vérifie encore de nos jours et est utilisée par l'industrie des semi-conducteurs pour la planification à long terme ainsi que pour fixer des objectifs de recherche et développement.

Pendant près de 40 ans [Sut05], cette augmentation du nombre de transistors par circuit intégré allait de pair avec une augmentation de la fréquence d'horloge des processeurs. En conséquences, les programmes pouvaient faire des calculs plus complexes sans que le temps d'exécution n'augmente trop, le matériel étant renouvelé régulièrement. Ainsi, les programmes ont pu gagner en complexité pour gérer des ensembles de données toujours plus grand. Les programmeurs s'occupaient alors plus des fonctionnalités que de l'efficacité.

Il y a environ 10 ans, la dissipation thermique des processeurs étant devenue trop importante, la fréquence d'horloge a alors cessé d'augmenter au profit d'une multiplication du nombre de cœurs de calcul disponibles dans un processeur. Pour continuer à améliorer les performances, les programmeurs doivent donc parvenir à tirer parti de ces cœurs de calcul additionnels.

La conception de programmes parallèles, c'est à dire capables de s'exécuter sur plusieurs unités de calcul, devient ainsi un enjeu majeur du monde numérique. Écrire de tels pro-

1. Gordon E. Moore deviendra plus tard l'un des trois co-fondateurs d'Intel Corporation

grammes consiste à découper une tâche en plusieurs calculs de plus petite taille pouvant alors s'exécuter sur les différents cœurs du système : c'est le processus de parallélisation.

La parallélisation d'un programme étant propre au calcul effectué, chaque parallélisation est unique. Cependant, les mécanismes de cette parallélisation reposent sur un nombre restreint de paradigmes. On distingue ainsi deux grands types de parallélisme : le parallélisme de donnée et le parallélisme de tâche. Le premier consiste à exécuter en parallèle différents calculs correspondant à des tâches distinctes tandis que le second consiste à exécuter un même calcul sur des données indépendantes. Étant donné leur fonctionnement, le parallélisme de tâche s'applique souvent aux programmes composés de plusieurs modules distincts, le parallélisme de donnée s'appliquant quant à lui aux nombreuses boucles de calcul itérant sur un ensemble de données.

Un des problèmes rencontrés dans la mise en œuvre de ces paradigmes est la présence de liens causals entre les calculs. Certains calculs utilisent en effet le résultat d'autres calculs, créant un lien de dépendance entre eux : ce type de dépendance se nomme une dépendance de données. Par exemple, si a est défini par $a = 2+3$ et b par $b = a*2$, il existe une dépendance de donnée entre b et a car b utilise le résultat du calcul de a. Ce type de dépendance se produit notamment dans le cas d'interactions entre modules, ou encore lorsque des boucles de calcul utilisent une donnée résultant de l'itération précédente.

La présence de dépendances de données a un impact fort sur les performances du parallélisme. En effet, elles imposent une exécution séquentielle des calculs concernés. Or, comme l'a énoncé Gene Amdahl[Amd67], les gains théoriques pouvant être obtenu par la parallélisation sont limités par la présence de portions de code séquentiel. Aussi, un programme présentant un grand nombre de dépendances ne peut être amélioré par le parallélisme de donnée ou le parallélisme de tâche.

Cependant, les programmes destinés au traitement de flux de données peuvent recourir dans ce cas à un autre type de parallélisme : le parallélisme de flux. Ce paradigme consiste à diviser le traitement effectué sur chaque donnée du flux en une succession de sous-traitements appelés « étapes », les étapes s'exécutant en parallèle. Les données du flux transitent alors d'une étape à une autre dans l'ordre où elles apparaissent afin d'être complètement traitées. Lorsqu'une étape finit de traiter une donnée, elle envoie le résultat à l'étape suivante et récupère le résultat de l'étape précédente. De cette façon, le traitement des données reste séquentiel et les dépendances de données sont respectées. Le parallélisme provient alors de l'exécution des étapes en parallèle sur des données successives du flux. À un instant donné, la dernière étape traite la donnée i pendant que l'étape précédente traite la donnée $i + 1$ et ainsi de suite (voir figure 1.1). La latence reste ainsi inchangée avec ce paradigme mais le débit est augmenté. Ce paradigme est donc intéressant dès lors que le nombre de données à traiter est grand au regard de la latence de traitement d'une donnée.

Un des points cruciaux pour obtenir de bonnes performances avec le parallélisme de flux est l'efficacité de l'algorithme de communication utilisé. En effet, après chaque donnée à traiter, une étape doit transmettre le résultat de son calcul à l'étape suivante. En considérant la dernière étape, cela signifie qu'une donnée finit son traitement global chaque fois qu'une étape a fini de s'exécuter et qu'une communication s'est produite. Étant donné *(i)* un temps de traitement global T_{seq} *(ii)* un temps de communication T_{comm} *(iii)* n étapes de même durée , le débit est donné par $\frac{T_{seq}}{n} + T_{comm}$ unités de temps. Aussi, le débit maximal possible en augmentant le nombre n d'étapes est limité à une donnée toutes les T_{comm} unités de temps. La figure 1.2 montre comment l'accélération varie pour plusieurs valeurs de T_{comm}.

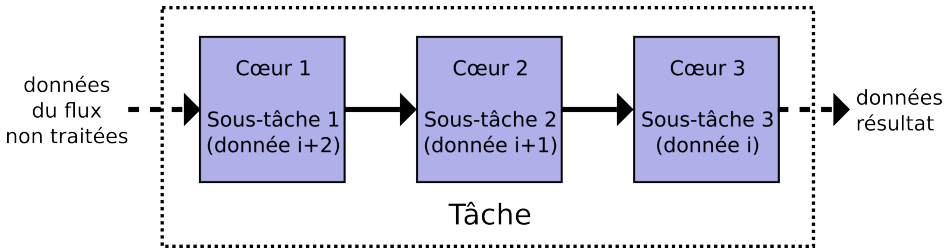


FIGURE 1.1 – Flux de données avec parallélisme de flux

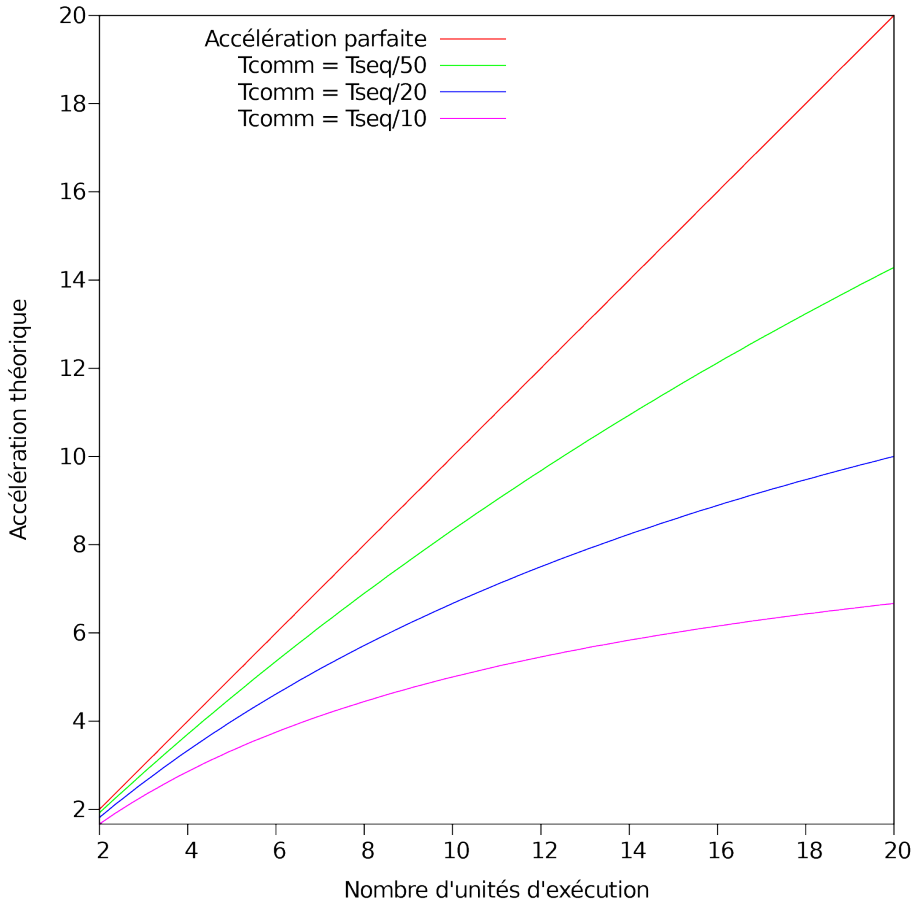


FIGURE 1.2 – Influence du débit de la communication sur l'accélération fournie par le parallélisme de flux

Dans le contexte des systèmes multi-cœurs actuels, les unités de calcul que sont les cœurs ne disposent pas de primitives de communication en tant que tel. À la place, les cœurs partagent la mémoire et peuvent s'échanger des données en y effectuant des lectures et écritures. Une donnée peut ainsi être transmise d'un cœur à un autre en l'écrivant à une adresse donnée depuis le cœur émetteur puis en la lisant à cette même adresse depuis le cœur récepteur. La transmission d'un nombre arbitraire de données nécessite alors l'utilisation d'un algo-

rithme de communication producteur / consommateur pour gérer la synchronisation entre le cœur émetteur et le cœur récepteur. En effet, avant d'écrire des données en mémoire, le cœur émetteur doit s'assurer que les données déjà contenues à ces adresses ont bien été lues par le cœur récepteur. Réciproquement, avant de lire des données, le cœur récepteur doit s'assurer que les adresses dans lesquelles il va les lire contiennent bien de nouvelles données.

La littérature dans le domaine des systèmes répartis comporte de nombreux algorithmes de communication de type producteur / consommateur, tant dans le domaine des systèmes répartis que dans celui des systèmes à mémoire partagée. Un des axes majeurs sur lesquels ont été développés ces algorithmes est l'amélioration de l'efficacité de la communication, mesurée tantôt en débit et tantôt en latence. Pourtant, ces algorithmes ne sont pas adaptés aux besoins exigeants du parallélisme de flux. Pour passer à l'échelle du nombre de cœurs utilisés, celui-ci nécessite en effet d'employer un algorithme de communication dont le débit est suffisamment élevé pour que le temps entre deux envois de données soit négligeable au regard du temps de calcul d'une étape. De plus, il est nécessaire que cet algorithme soit asynchrone pour s'accomoder des variations de rythme des différentes étapes. Or, si des algorithmes de communication asynchrone existent, le débit dont ils sont capables n'est pas aussi important qu'il pourrait l'être. En particulier, ces algorithmes ne prennent pas suffisamment en compte le coût du partage de données.

Bien que la communication des systèmes multi-cœurs soit basée sur le partage de mémoire, la mémoire centrale est en pratique peu utilisée pour partager des variables. Celle-ci étant lente d'accès, les cœurs du système ont donc recours à un ensemble de mémoires cache matérielles pour éviter d'y accéder. Typiquement, le système est composé de mémoires caches partagées par plusieurs cœurs et chaque cœur possède également sa propre mémoire cache dans laquelle il y effectue ses opérations d'accès mémoire. Ces différentes mémoires cache forment ainsi avec la mémoire centrale une mémoire partagée répartie dont la cohérence est maintenue par un protocole appelé MOESI. C'est donc dans ces mémoires caches que les données sont partagées sous l'action du protocole MOESI.

Pour que ce protocole fonctionne, des messages sont échangés entre les caches afin de se notifier des différents accès mémoire. Or l'envoi d'un message nécessite plusieurs cycles processeurs, temps pendant lequel plusieurs instructions pourraient être exécutées. Le mécanisme de synchronisation a donc un coût élevé et joue en conséquence un rôle essentiel dans l'efficacité d'une communication entre deux cœurs et a fortiori dans l'efficacité du parallélisme de flux.

1.2 Contributions

Parmi les algorithmes de communication de la littérature, de nombreux efforts ont été effectués pour ne pas utiliser de verrou dans la synchronisation entre producteur et consommateur. Cependant, en dépit de ces efforts, le coût induit par les mécanismes de maintien de la cohérence mémoire rend cette synchronisation coûteuse malgré tout. Il est donc indispensable pour obtenir un grand débit de communication d'éviter de solliciter cette cohérence matérielle.

Or, si la communication dans les systèmes à mémoire partagée repose sur l'existence de mécanisme de cohérence, il reste possible de limiter son impact en relâchant certaines contraintes habituellement requises pour une communication. En effet, la communication

effectuée dans le parallélisme de flux possède des caractéristiques particulières :

— elle ne concerne qu'un seul producteur et consommateur ;

— le débit a plus d'impact sur les performances que la latence.

Cette thèse propose l'élaboration d'un nouvel algorithme de communication tirant parti de ces particularités pour effectuer des optimisations supplémentaires. Il est ainsi possible d'envoyer les données par lot pour réduire le coût de synchronisation par donnée envoyée et de se limiter à une communication simple entre un producteur et un consommateur dans le but de simplifier la synchronisation.

Les contributions de cette thèse sont :

1. L'élaboration d'un algorithme de communication inter-cœurs performant appelé BatchQueue. Les bonnes performances de cet algorithme sont obtenues en prenant en compte l'influence des mécanismes de fonctionnement des caches matériels, en particulier celui assurant la cohérence entre les caches et le mécanisme de préchargement.
2. Une étude comparative étendue des différents algorithmes de communication existants. Cette étude compare les algorithmes dans deux configurations : la communication entre deux cœurs d'une part, et la communication entre un ensemble de cœurs chaînés entre eux d'autre part. Deux aspects des algorithmes sont évalués : leur débit maximal et l'impact qu'ils ont sur les performances des programmes au sein desquels ils sont mis en œuvre.
3. L'incorporation de l'algorithme de communication BatchQueue au sein de l'extension de calcul par flux pour OpenMP, une solution de parallélisation novatrice pour le parallélisme de flux. L'implémentation inclus notamment un mécanisme de sélection automatique entre BatchQueue et l'algorithme de communication déjà inclus dans cette solution de parallélisation, celui-ci étant moins rapide mais gérant plus de cas.
4. Une évaluation de l'extension de calcul par flux pour OpenMP avec et sans la présence de l'algorithme BatchQueue. L'évaluation consiste en une série de trois programmes spécialement annotés pour mettre en œuvre le parallélisme de flux afin de tirer parti des différentes unités de calcul disponibles.

1.3 Plan de la thèse

Le présent manuscrit s'organise de la façon suivante :

Chapitre 2 : Architectures multi-cœurs En tant qu'intermédiaires pour accéder à la mémoire, les caches matériels jouent un rôle critique dans les performances des programmes. Ce chapitre est donc dédié à la présentation des caches matériels et leur influence sur les performances. La première partie explique le fonctionnement des caches matériels et les différents paramètres qui affectent leur efficacité. La seconde partie détaille ensuite les différentes organisations possibles des caches matériels au sein des systèmes ayant plusieurs unités de calcul et leur impact sur la latence et le débit d'accès à la mémoire. Enfin, la troisième partie décrit le mécanisme de cohérence des caches et les coûts qui lui sont associés.

Chapitre 3 : Algorithme de communication inter-cœurs BatchQueue Les systèmes multi-cœurs tendent à communiquer plus que les systèmes répartis traditionnels ce qui les rend sensibles à l'efficacité des algorithmes de communication utilisés. Or, la mémoire partagée dont disposent ces systèmes permet la mise au point d'algorithmes de communication plus efficaces. Ce chapitre décrit l'élaboration d'un tel algorithme. Ce chapitre contient tout d'abord un état de l'art des algorithme de communication existant classés par familles. Puis, le fonctionnement du nouvel algorithme de communication BatchQueue est présenté suivi d'une comparaison de ses performances avec les algorithmes existants.

Chapitre 4 : Parallélisme de flux optimisé avec BatchQueue Un usage intéressant d'un algorithme de communication inter-cœurs rapide est l'amélioration du parallélisme de flux. Celui-ci repose sur la circulation d'un flux de données à travers les différents cœurs de calcul d'un système et ses performances sont donc très liées à celles de l'algorithme de communication utilisé. Ce chapitre traite de l'utilisation du l'algorithme BatchQueue afin d'améliorer les performances de l'extension de calcul par flux pour OpenMP. Ce chapitre commence par présenter et comparer les différentes formes de parallélisme ainsi que les outils haut niveau permettant de les mettre en œuvre. Ce chapitre décrit ensuite en détail le processus d'intégration de BatchQueue dans l'extension de calcul par flux pour OpenMP. Enfin, ce chapitre présente une évaluation de l'amélioration des performances ainsi obtenue au travers des tests en environnement contrôlé et des applications converties pour utiliser l'extension de calcul par flux pour OpenMP.

Chapitre 5 : Conclusion Ce chapitre conclut ce manuscrit par un résumé des résultats obtenus durant cette thèse et par une discussion sur les perspectives de ce travail.

Chapitre 2

Architectures multi-cœurs

Sommaire

De nos jours, la majeure partie des systèmes informatiques disposent de plusieurs unités de calcul de mêmes fréquences, qu'il s'agisse de superordinateurs, d'ordinateurs de bureau, d'ordinateurs portables ou de systèmes embarqués. Ces systèmes sont alors dits « à multitraitement symétrique », ou simplement *SMP* (pour « Symmetric Multi-Processing »). Contrairement à l'augmentation en fréquence des cœurs de calcul, cette évolution des systèmes ne peut se faire de façon transparente pour les applications. En effet, l'utilisation efficace de ces architectures passe par une parallélisation des applications.

Pour que cette parallélisation soit efficace, il est nécessaire de bien prendre en compte les spécificités matérielles de ces systèmes. Connaître ces spécificités est également nécessaire pour comprendre certaines répercussions sur les performances des programmes, tel le coût important d'une section critique lorsqu'elle est exécutée sur plusieurs unités de calcul [LDT+].

Il existe cependant plusieurs familles de systèmes *SMP*, chacune ayant ses propres particularités. Il faut donc connaître les particularités de tous ces systèmes pour obtenir de bonnes performances sur chacun d'eux. Pour simplifier la parallélisation d'applications, cette thèse

propose donc deux contributions permettant de paralléliser une application de façon efficace sans nécessiter de connaissances étendues sur les systèmes *SMP*. Comprendre comment ces contributions parviennent à être efficaces requiert en revanche une bonne compréhension des caractéristiques des systèmes *SMP* que ce chapitre a pour but de détailler.

La première section de ce chapitre est dédiée à la présentation des caches matériels et de leurs paramètres en l'absence de concurrence, c'est à dire dans le cadre d'une unité de calcul isolée. La section suivante étudie alors les différentes architectures de ces systèmes, en particulier la connexion entre les unités de calcul. Enfin, la dernière section introduit le système de cohérence mémoire et les coûts qui y sont associés.

2.1 Caches matériels

Les accès à la mémoire font partie des instructions les plus fréquemment utilisées dans tous les programmes. Or, ces instruction d'accès à la mémoire ont un coût élevé en nombre de cycles processeur. Sans précaution particulière, de nombreux cycles seraient perdus à chaque accès à la mémoire, réduisant d'autant l'efficacité globale du système. L'utilisation de caches matériels permet de réduire de manière drastique l'impact des accès à la mémoire. Cependant, l'accès à la mémoire doit suivre certaines contraintes pour obtenir les meilleures performances possibles des caches matériel.

Cette section présente l'origine et le fonctionnement des caches matériels, ainsi que les contraintes qui y sont associées.

2.1.1 Idée générale

Dans un ordinateur, la mémoire centrale et le processeur opèrent à des fréquences différentes : le processeur opère à une fréquence significativement plus élevée que celle de la mémoire. De plus, les mémoires centrales de type DRAM qui se trouvent dans les ordinateurs actuels possèdent de nombreuses limitations physiques qui mènent à des cycles mémoire non utilisés pour transmettre des données [1]. En conséquence, une instruction d'accès mémoire nécessite de nombreux cycles processeur pour s'exécuter.

Pour atténuer ce problème, des mémoires plus petites que la mémoire centrale mais d'accès plus rapide sont disposées entre elle et le processeur. Ces mémoires sont généralement désignées sous le nom de **caches matériels**. En dehors des zones mémoire qui servent à communiquer avec les périphériques, toute donnée lu ou écrite en mémoire passe par le cache matériel. Ainsi, lorsqu'une donnée doit être lue en mémoire, celle-ci est d'abord cherchée dans le cache matériel. Si celle-ci n'y est pas présente, le cache matériel lit la donnée depuis la mémoire centrale puis la renvoie au processeur : ce scénario se nomme un **défaut de cache**. De la même manière, une donnée à écrire est d'abord stockée dans le cache matériel puis celui-ci propage l'écriture en mémoire centrale. L'idée derrière ce dispositif est d'éviter un accès à la mémoire en cas d'utilisation ultérieure d'une donnée.

Le concept de cache matériel repose sur l'hypothèse que l'accès aux données vérifie le principe dit de localité. Deux types de localité existent :

— localité temporelle ;

— localité spatiale.

1. Pour plus d'informations sur ces limitations physiques, le lecteur se référera à la section 2 de l'article [Dre07].

2.1.1.1 Localité temporelle

La localité temporelle est le principe selon lequel les données sont souvent accédées plusieurs fois, et cela dans un court laps de temps. En effet, leur utilité dépend entièrement de la probabilité de présence des données dans le cache. Cette probabilité est d'autant plus grande que le temps séparant deux accès à une même donnée est court.

Une caractéristique essentielle d'un cache est qu'il a une capacité bien plus petite que la mémoire principale : si la capacité des deux mémoires était du même ordre de grandeur, c'est dans la mémoire la plus rapide d'accès que les données seraient stockées. Une conséquence de la différence de taille est qu'il est possible pour la mémoire principale de contenir plus de données que le cache ne le peut. Une donnée d peut donc être accédée mais le cache être déjà rempli d'autres données. Lorsque cela se produit, une donnée doit être retirée du cache afin de libérer de la place pour stocker la donnée d – on parle alors d'éviction de donnée. Un nouvel accès à la donnée ainsi évincée occasionne alors un défaut de cache : la donnée doit être copiée depuis la mémoire à nouveau.

C'est pour cette raison qu'un cache repose sur la brièveté du laps de temps qui sépare les différents accès aux données. Plus le temps entre deux accès à une donnée d est grand, plus le nombre de données différentes qui peuvent être accédé pendant cet intervalle est important. Or si ce nombre est trop grand, la donnée d sera alors évincée et l'accès suivant occasionnera un nouveau défaut de cache. Pour que le cache permette effectivement de réduire le nombre de défaut de cache, il est donc nécessaire que le temps entre les accès successifs aux données soit court.

2.1.1.2 Localité spatiale

La localité spatiale est le principe selon lequel des données proches l'une de l'autre en mémoire sont accédées dans un intervalle de temps court. Lorsque ce principe est vérifié, il devient avantageux de copier simultanément dans le cache les données contiguës en mémoire. En effet, le coût d'une copie n'est souvent pas linéaire vis à vis du nombre de données copiées, c'est à dire qu'il est souvent plus rapide de copier plusieurs données en une fois que de les copier une par une. C'est le cas notamment de la mémoire de type DRAM utilisée dans les mémoires centrales aujourd'hui. Il est donc intéressant de copier simultanément dans le cache – et donc à moindre coût – les données contiguës en mémoire, sachant qu'il existe une forte probabilité que celles-ci soient accédées par la suite.

Ce principe se vérifie également pour d'autres types de cache, tel le cache des pages où un bloc entier est lu depuis le disque lorsqu'une donnée est accédée. Les caches matériels requiert par contre une localité spatiale plus forte pour être efficace, la quantité de données pouvant être transférées ensembles étant plus petite.

La localité temporelle et la localité spatiale ont toutes deux leur origine dans les modèles habituels de programmation. La localité temporelle provient ainsi de l'utilisation de boucles dans les langages de programmation. L'exécution d'une boucle amène en effet à accéder en lecture ou écriture des données de façon répétée dans un court laps de temps, rendant les caches matériels utiles. De même, le principe de localité spatiale s'explique par l'existence de structures, d'objets et de tableaux. Ces structures de données ont pour point commun de grouper de façon contiguë en mémoire des données ayant un lien logique entre elles. Ces données sont alors souvent accédées ensemble, notamment au sein de boucles dans le cas des tableaux, créant ainsi la localité spatiale.

2.1.2 Paramètres de performance des caches matériels

Si les principes de localité temporelle et spatiale sont essentiels à l'efficacité des caches matériels, il existe en pratique, de nombreux autres paramètres influençant leur efficacité. De fait, il n'est pas possible d'obtenir de bonnes performances sans en tenir compte. Cette sous-section a donc pour but de décrire chacun de ces paramètres.

2.1.2.1 Lignes de cache

La propriété de localité spatiale est un des éléments majeur de la performance des caches matériels. Elle permet d'améliorer leurs performances en copiant simultanément des données contiguës en mémoire. Cependant, l'efficacité de ce mécanisme dépend pour beaucoup du placement des données dans la mémoire. En effet, pour des raisons de simplicité d'implémentation, la mémoire centrale ainsi que le cache matériel ne permettent pas de copier n'importe quelle séquence de données mais seulement les séquences dont la première donnée a une adresse multiple d'une puissance de deux donnée. Il n'est donc pas possible de récupérer systématiquement dans le cache une donnée et les n suivantes.

La mémoire centrale est virtuellement découpée en segments, appelés lignes de caches, dont la taille correspond à la quantité de données copiées simultanément depuis la mémoire vers un cache matériel. L'expression « **ligne de cache** » désigne ainsi un ensemble de données et par extension la taille de cet ensemble. Lorsqu'une donnée est accédée, c'est la ligne de cache entière à laquelle elle appartient qui est copiée dans le cache. Pour une donnée d'adresse $addr$ et une ligne de cache de taille N, les données copiées sont donc celles qui résident dans l'intervalle d'adresses $[\![a * N, (a+1) * N[\![$ avec $a = \lfloor \frac{addr}{N} \rfloor$. Autrement dit, les bits d'une adresse ayant un poids supérieur ou égal à N déterminent la ligne de cache dans laquelle réside la variable située à cette adresse.

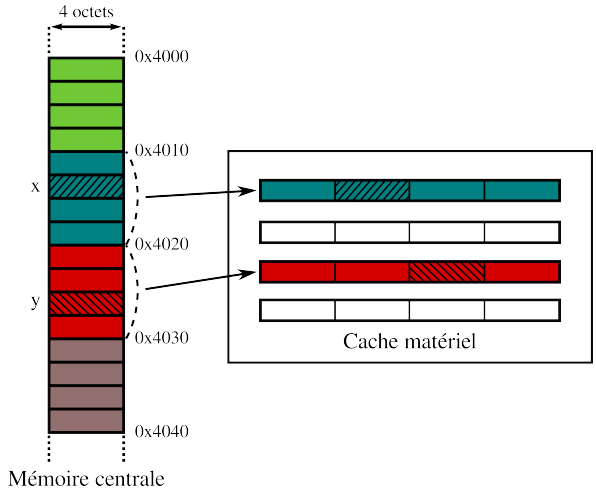


FIGURE 2.1 – Effets sur le cache de la lecture de x et y

La figure 2.1 présente ce phénomène au travers la lecture de deux données x et y. Les données x et y résidant aux adresses 0x4014 et 0x4028, leur lecture mènent à la copie des deux lignes de cache contenant les données situées entre les adresses 0x4010 et 0x401F d'une part et 0x4020 et 0x402F d'autre part.

Le découpage de la mémoire en lignes de cache est une information importante à prendre en compte pour obtenir de bonnes performances. Une variable dont la position en mémoire n'est pas considérée peut en effet générer plus de défauts de cache que nécessaire si celle-ci se retrouve en fin de ligne de cache. Par exemple une variable occupant 8 octets peut nécessiter de copier depuis la mémoire deux lignes de cache au lieu d'une si celle-ci occupe les 4 derniers octets d'une ligne de cache et les 4 premiers octets de la ligne de cache suivante. Pour cette raison, de nombreuses variables sont placées en mémoire de façon à être en début de ligne de cache. On dit qu'elles sont « alignées à la taille d'une ligne de cache », ou simplement « alignées à une ligne de cache ». Aligner une variable n'est cependant pas sans conséquences puisque l'alignement nécessite souvent de laisser une zone de mémoire inutilisée avant une donnée pour celle-ci soit alignée. Un choix des variables à aligner doit donc être opéré.

2.1.2.2 Préchargement

La taille des lignes de cache de la mémoire résulte d'un compromis. Augmenter sa taille permet de mieux tirer parti du principe de localité mais augmente dans le même temps sa latence d'accès. La taille d'une ligne de cache est donc un équilibre entre ces deux tendances et est choisie afin d'obtenir les meilleures performances. Ce faisant, il est assez courant d'avoir une localité spatiale plus étendue que la taille des lignes de cache choisie : il faut alors copier plusieurs lignes de caches depuis la mémoire.

Pour y remédier, les caches matériels sont équipés d'une **unité de préchargement**. Cette unité a pour but de détecter des modèles d'accès à la mémoire et de charger en avance les lignes de cache en fonction de ce modèle. Ainsi, si des données s'étalant sur deux lignes de cache sont accédées séquentiellement, cette unité va copier en avance la deuxième ligne de cache avant qu'elle ne soit réellement utilisée. Ainsi, lorsqu'elles sont accédées, ces données sont déjà présentes dans le cache et aucun cycle processeur n'est perdu.

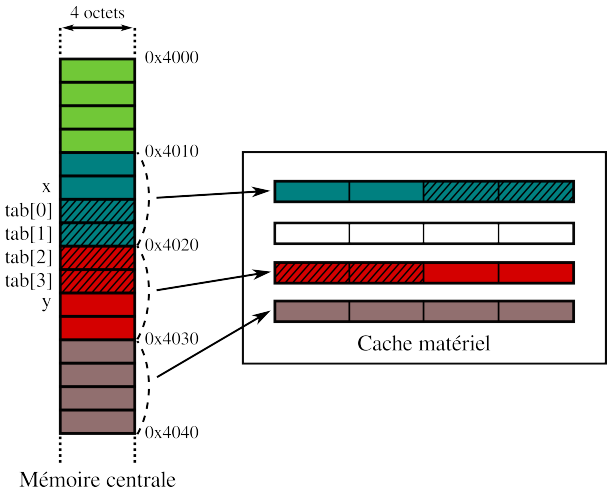


FIGURE 2.2 – Effets sur le cache de la lecture de tab en présence d'une unité de préchargement

La figure 2.2 présente le préchargement qui se produit après la lecture des entrées du tableau *tab* occupant deux lignes de cache consécutives. On observe ainsi que la ligne de

cache qui suit les deux qu'occupe *tab* est également copiée dans le cache. Dans ce cas précis le préchargement est inutile mais si le tableau était plus grand, aucun défaut de cache ne se serait produit lors de l'accès à la troisième ligne de cache.

Initialement seuls les accès séquentiels étaient détectés mais les unités de préchargement des processeurs modernes peuvent détecter n'importe quelle séquence d'accès linéaire. Par exemple, ces unités peuvent détecter une séquence où une ligne sur trois est accédée.

La présence d'unité de préchargement n'a pas que des effets positifs sur les performances d'un cache. Comme le montre l'exemple de la figure 2.2, les prévisions qu'effectue cette unité quant aux lignes de cache nécessaires dans le futur permettent de récupérer à l'avance ces lignes de cache, masquant ainsi le temps de copie. Quand le cache matériel est plein, des lignes de cache doivent être évincées pour permettre aux lignes de cache copiées par l'unité de préchargement d'être mises dans le cache. Or, comme il s'agit de prévisions, il s'avère parfois que les lignes de cache récupérées ne sont pas utilisées. Dans ce cas, des lignes de cache sont évincées à tort, alors que celles-ci auraient pu s'avérer utiles. La présence d'une unité de préchargement incite donc le programmeur à organiser les données de telle sorte qu'elles soient accédées séquentiellement afin d'améliorer la localité spatiale et réduire le nombre de mauvaises prévisions.

2.1.2.3 Taille des caches

Parmi les différents paramètres des caches matériels, la taille est l'un des facteurs ayant le plus d'impact sur les performances. Comme mentionné précédemment, les principes de localité temporelle et spatiale, sur lesquels reposent l'intérêt des caches matériels, sont fortement liés à l'utilisation de boucles dans les langages de programmation. Les boucles se distinguent en effet par des accès répétés à un ensemble de variables dont certaines, tels les tableaux, occupent une zone contiguë en mémoire. Les boucles tirent ainsi parti de la présence de caches matériels car les variables accédées durant un tour de boucle sont déjà présentes dans le cache, ayant déjà été copiées dans celui-ci par un tour de boucle précédent. Le premier tour de boucle bénéficie également de la présence d'un cache car l'accès aux cases d'un tableau ne nécessite pas autant d'accès à la mémoire qu'il y a de cases, en raison de l'existence des lignes de cache.

Ceci ne fonctionne cependant que si l'ensemble des données manipulées pendant un tour de boucle, on parle alors d'ensemble de travail d'une boucle, est plus petit que la taille du cache. En effet, si l'ensemble de travail d'une boucle est plus grand que la taille du cache, un tour de boucle accède alors à plus de données que le cache ne peut en stocker et des évictions doivent se produire. Dans les cas extrêmes, chacune des données est évincée avant d'être accédée à nouveau. Il est donc important pour qu'une boucle bénéficie de la présence d'un cache matériel que celui-ci soit plus grand que son ensemble de travail. Or les boucles n'ont pas toutes un ensemble de travail de même taille, certaines sont en effet plus complexes que d'autres et leur ensemble de travail est donc plus grand. La taille d'un cache influence donc le nombre de boucles tirant parti de la présence de celui-ci car leur ensemble de travail est plus petit que celui-ci.

La taille d'un cache influence aussi tout autre segment de code d'un programme manipulant de façon répétée un ensemble restreint de données telles les fonctions ou de manière générale tout segment correspondant à une sous-tâche du programme. La notion d'ensemble de travail n'est en effet pas propre aux boucles : elle désigne l'ensemble des données utilisées par un programme pendant un laps de temps donné. Les fonctions et sous-tâches d'un

programme ont également un ensemble de travail dont la taille par rapport à celle du cache est importante pour les performances. La taille d'un cache est donc un facteur important de performances et, pour une taille de cache donnée, il convient pour un programmeur de faire en sorte que les ensembles de travail des segments de code qu'il écrit soient plus petits que la taille du cache.

2.1.2.4 Associativité

Le lien entre taille d'un cache et taille des ensembles de travail qui peuvent y résider n'est cependant pas linéaire. Pour des raisons de complexité d'implémentation, la plupart des caches matériels ne permettent pas à une ligne de cache d'être stockée n'importe où. Pour une ligne de cache donnée, seul un sous-ensemble des emplacements de stockage peut être utilisés pour stocker cette ligne de cache. On parle dans ce cas de **cache associatif avec ensembles à N voies** (« N-way set associative cache »). Cela signifie que chaque ligne de cache est associée à un unique ensemble et que celle-ci peut être stockée dans n'importe lequel des N emplacements de cet ensemble.

FIGURE 2.3 – Découpage d'une adresse mémoire

Tout comme les bits de poids fort d'une adresse déterminent dans quelle ligne de cache réside la donnée à cette adresse, certains de ces bits déterminent à quel ensemble est associée la ligne de cache. Une adresse est divisée en trois parties dont l'organisation est présentée dans la figure 2.3. Ces trois parties sont :

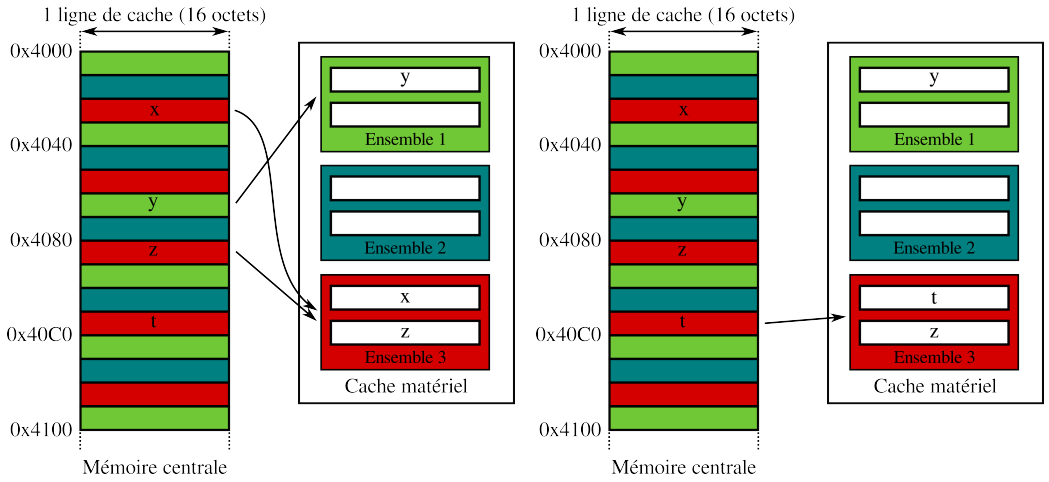
— l'étiquette ;

— l'index ;

— le décalage.

Parmi ces 3 ensembles de bits, ceux de l'étiquette et de l'index sont les bits de poids fort indiquant la ligne de cache à laquelle appartient la donnée à cette adresse tandis que ceux du décalage donnent la position de la donnée dans la ligne de cache. Les bits de poids fort ont donc également un autre sens : les **bits de l'index** déterminent un ensemble d'emplacements dans lesquels elle peut être stockée, les **bits de l'étiquette** servent à différencier les lignes de cache qui partagent un même ensemble. En effet, toutes les lignes de cache dans un ensemble ont, par définition, les mêmes bits d'index. À chaque ligne de cache présente dans un ensemble sont alors associés les bits de l'étiquette pour identifier de quelle ligne de cache il s'agit.

Lorsqu'une donnée est accédée, que ce soit en lecture ou en écriture, l'unité de gestion du cache analyse dans son adresse les bits de l'index pour déterminer dans quel ensemble celle-ci doit être stockée. Ensuite, elle compare l'étiquette de l'adresse avec les étiquettes de chaque lignes de cache dans cet ensemble. Si une correspondance se produit, la donnée peut être lue ou écrite en utilisant les bits du décalage pour y accéder. Sinon, la ligne de cache à laquelle la donnée appartient est copiée depuis la mémoire dans l'une des entrées de l'ensemble. Un tel exemple d'accès à des données est représenté dans la figure 2.4. La figure montre le comportement du cache en réaction à la lecture de quatre données, dont



(a) Après lecture de x, y et z (b) Après lecture additionelle de t

FIGURE 2.4 – Fonctionnement d'un cache associatif à 3 voies

trois résident à des adresses ayant les même bits d'index. Bien que le cache comporte assez d'emplacements pour stocker toutes les données, le fait que celui-ci ait une associativité à 2 voies empêche les données d'être stockée n'importe où.

Ce fonctionnement concerne le cas général des caches associatifs. Il existe cependant deux autres formes de caches matériels : les caches *pleinement associatifs* et les caches *à association directe*.

Caches pleinement associatifs Ceci désigne les caches n'ayant qu'un seul ensemble et où les lignes de cache peuvent être stockées à n'importe quel endroit. Ces caches ne peuvent être utilisés que pour des caches de très petite taille car leur complexité grandit exponentiellement avec leur taille. En effet, pour que le cache puisse fonctionner dans des délais très court il faut pouvoir comparer simultanément l'étiquette avec toutes les entrées, ce qui nécessite de nombreux circuits électroniques.

Caches à association directe À l'inverse, ces caches ne contiennent qu'une entrée. Une ligne de cache ne peut donc être stockée qu'à un seul endroit dans le cache. Il en résulte qu'un accès à deux lignes de cache ayant le même index rend le cache complètement inutile puisque celles-ci se remplacent alors l'une l'autre, occasionnant des défauts de cache à chaque accès : on parle alors de conflit. À cet égard l'utilisation de caches associatifs à N voies par ensemble résulte d'un compromis entre complexité – et donc prix – du cache et nombre de conflits.

Une conséquence de l'utilisation de caches associatifs est l'apparition de conflits dans des cas où l'ensemble de travail est inférieur à la taille du cache. Les performances ne sont donc maximales que si les données sont correctement réparties en mémoire vis à vis des bits de poids forts. Ainsi, l'alignement de nombreuses données sur la taille d'une page mémoire est susceptible de créer de nombreux conflits et de dégrader les performances [MS09].

2.1.2.5 Politique d'éviction

Dans un cache matériel, aucune ligne de cache n'est retirée tant que le besoin ne s'en fait pas sentir. Pour un ensemble donné, chaque nouvelle ligne de cache copiée dans l'ensemble remplit un peu plus celui-ci jusqu'à ce qu'il soit plein. À partir de cet instant, chaque ligne copiée dans l'ensemble doit évincer une autre ligne de cache, idéalement une ligne qui ne contient aucune donnée qui sera accédée plus tard. Le choix de la ligne à évincer est important : si celui-ci est mal fait, le cache risque d'évincer des lignes de cache contenant des données qui seront utilisées peu de temps après, causant ainsi un défaut de cache qui aurait pu être évité. On appelle ce choix des données à évincer la *politique d'éviction*.

La politique la plus répandue prend pour hypothèse que les données respectent une variante plus stricte du principe de localité temporelle. L'hypothèse est que la probabilité qu'une donnée soit réutilisée au bout d'un temps t est inversement proportionnelle à ce temps. C'est à dire qu'une donnée a plus de probabilité d'être réutilisée après un court temps qu'après un temps important. La conséquence est que plus une donnée reste longtemps sans être utilisée, moins elle a de chances d'être à nouveau utilisée. La politique d'éviction consiste alors à remplacer la ligne de cache qui n'a pas été utilisée pendant le temps le plus long. On parle alors de politique LRU pour « Least Recently Used » (« Utilisé le Moins Récemment »). Cette politique étant coûteuse à implémenter, ce sont souvent des approximations de cette politique qui sont utilisées : la ligne de cache évincée n'est alors qu'une des lignes utilisées il y a le plus longtemps.

2.1.2.6 Interaction avec la mémoire

Lorsqu'une ligne de cache est accédée en écriture, la ligne de cache qui la contient est d'abord copiée dans le cache. C'est cette copie qui est modifiée lors de l'écriture. La valeur de la donnée n'est alors présente que dans le cache. Cette valeur doit donc être finalement propagée dans la mémoire.

La propagation des écritures vers la mémoire peut s'effectuer de deux façons différentes. La première consiste à écrire en mémoire tout ce qui est écrit dans le cache. Chaque écriture se produit donc simultanément dans le cache et en mémoire. Cette propagation est dite **en écriture directe** (« write-through »). La deuxième méthode consiste à n'effectuer les écritures que dans le cache en retardant leur propagation en mémoire. Celle-ci n'est alors faite qu'au moment de l'éviction de la ligne de cache. On parle dans ce cas de cache **en écriture différée** (« write-back »).

Bien que les deux modes de propagation présentent des avantages, c'est la politique d'écriture différée qui est de loin la plus utilisée. Comparée à la politique d'écriture directe, la politique d'écriture différée permet de réduire de façon drastique le nombre d'écritures en mémoire, réduisant d'autant la bande passante mémoire utilisée. Ainsi, lorsque deux écritures sont effectuées dans une même ligne de cache sur un court laps de temps, cette politique n'exécute qu'une écriture en mémoire, lorsque la ligne de cache est évincée. Dans les mêmes conditions, la politique d'écriture directe occasionne deux écritures. De plus, la politique d'écriture différée permet de réduire la latence d'écriture dans le cas de deux écritures successives, la seconde écriture n'ayant pas à attendre la fin de la première.

En revanche, la politique d'écriture différée peut provoquer une écriture en mémoire à n'importe quel moment, y compris lors d'un accès en lecture. En effet, si un défaut de cache se produit suite à un accès en lecture, la donnée accédée doit être copiée en mémoire ce qui

conduit à une éviction d'une ligne de cache. Si cette éviction est instantanée lorsque la ligne n'a pas été modifiée, elle peut engendrer une forte latence lorsque la propagation en mémoire d'une ligne de cache modifiée est nécessaire. Cependant, avec la localité spatiale, l'écriture différée se montre particulièrement performante et réduit énormément le nombre d'écritures en mémoire. Au regard du gain en bande passante, la latence induite par l'éviction des lignes de cache modifiées devient alors un coût tolérable. Ceci explique que c'est la politique à écriture différée qui est utilisée le plus souvent.

2.1.2.7 Hiérarchie de caches

Plusieurs procédés techniques existent pour fabriquer un cache, chacune ayant une latence d'accès et un coût qui lui est propre. De façon générale, plus la latence proposée est faible, plus le coût de construction est important. Il faut alors trouver le juste compromis entre :

— augmenter l'intérêt d'un accès réussi au cache, en utilisant des mémoires cache à accès toujours plus rapide ;

— minimiser le nombre de défauts de cache, c'est à dire augmenter la taille des caches.

La solution utilisée dans tous les processeurs modernes est de mettre en place un système hiérarchique de caches comprenant généralement deux ou trois niveaux. Chaque cache dispose notamment, en plus d'une taille et d'une latence d'accès qui lui sont propres, d'une taille de ligne de cache spécifique. De cette façon, les données qui font partie d'un ensemble de travail de petite taille restent dans le cache de premier niveau, tandis que les données d'un ensemble de travail de grande taille ont la possibilité de rester dans un cache de deuxième ou troisième niveau. Le cache de premier niveau est en contact direct avec le processeur et propose la meilleure latence possible pour les données qui peuvent y résider. Les autres caches nécessitent un plus grand nombre de cycles pour être accédés mais restent néanmoins plus rapides d'accès que la mémoire, offrant ainsi une amélioration des performances pour les ensembles de travail de grande taille. La figure 2.5 résume représente un exemple d'une telle hiérarchie de caches.

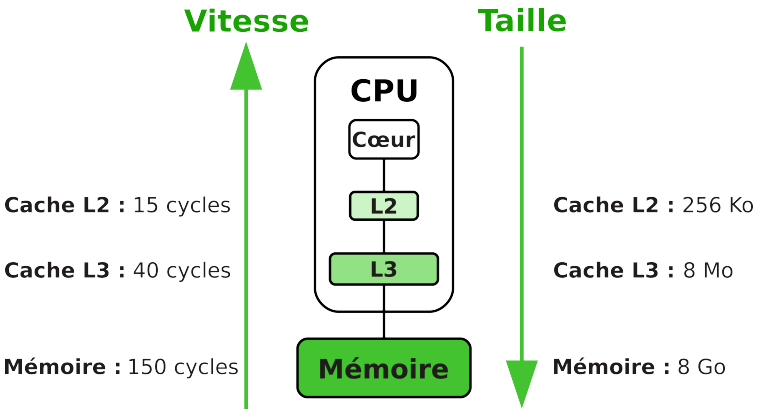


FIGURE 2.5 – Hiérarchie des caches

L'utilisation d'une **hiérarchie de caches** pose la question de la politique de stockage à employer lorsqu'une donnée dans un cache de niveau n est copiée dans un cache de niveau

inférieur m (par exemple une donnée copiée du cache de niveau 2 dans le cache de niveau 1). Deux politiques existent : soit la donnée ainsi copiée est disponible uniquement dans le cache de niveau m, soit elle est répliquée à tous les niveaux. La première approche est dite exclusive et la seconde approche inclusive. L'**approche exclusive** permet de minimiser l'espace occupé par une donnée. L'**approche inclusive** en revanche permet de supprimer une donnée d'un cache sans avoir à l'écrire dans le cache de niveau inférieur. C'est cette dernière approche qui offre les meilleurs résultats, l'espace ainsi perdu dans les caches de niveau inférieurs étant faible au regard de leur taille. En effet, la taille de chaque cache est un ordre de grandeur plus grand que celle des caches immédiatement supérieurs.

2.1.2.8 Traduction d'adresse

Dans les systèmes modernes, chaque processus voit la mémoire comme s'il était le seul à l'utiliser : on parle alors d'**espace d'adressage**. Cet espace d'adressage est permis par l'utilisation d'adresses virtuelles, c'est à dire que les adresses manipulées par un programme n'identifient pas un emplacement dans la mémoire physique. Pour trouver un emplacement, il faut effectuer une traduction de l'adresse virtuelle vers l'adresse physique.

Pour ne pas occasionner un surcoût mémoire trop important, la correspondance entre adresses virtuelles et adresses physiques n'est pas maintenue pour chaque adresse mais pour des groupes d'adresses. Pour ce faire, la mémoire est divisée en pages mémoire et chaque page virtuelle est associée à une page physique. Une page est alors définie par un certain nombre de bits de poids fort dans les adresses : toutes les adresses partageant les même bits de poids fort appartiennent ainsi à la même page, comme le montre l'exemple dans le tableau 2.1.

Numéro de page	Adresse	Représentation binaire de l'adresse	Décalage dans la page
41	0x41FFE	01000001111111111110	0xFFE
41	0x41FFF	01000001111111111111	0xFFF
42	0x42000	01000010000000000000	0x0
42	0x42001	01000010000000000001	0x1
42	0x42002	01000010000000000010	0x2
...	...	...	...
42	0x42FFD	01000010111111111101	0xFFD
42	0x42FFE	01000010111111111110	0xFFE
42	0x42FFF	01000010111111111111	0xFFF
43	0x43000	01000011000000000000	0x0
43	0x43001	01000011000000000001	0x1

TABLE 2.1 – Structuration de la mémoire en pages de 4 Kio

Le calcul de l'adresse physique consiste alors à traduire l'adresse de base de sa page, c'est à dire l'adresse la plus petite dans cette page, puis d'ajouter le décalage de l'adresse par rapport au début de la page. Par exemple, étant donnée une taille de page de 4 Kio et l'adresse 0x42160 à traduire, la traduction consiste à appliquer la traduction existante pour l'adresse 0x42000 en 0x64000 puis ajouter le décalage de 0x42160 à 0x42000, soit 0x160, pour

obtenir 0x64160. Un exemple de traduction d'adresse est donné dans la figure 2.6 [2] pour le cas de pages de 4 Kio avec des adresses sur 32 bits et l'*extension d'adresse physique* (« Physical Address Extension » ou PAE) activé.

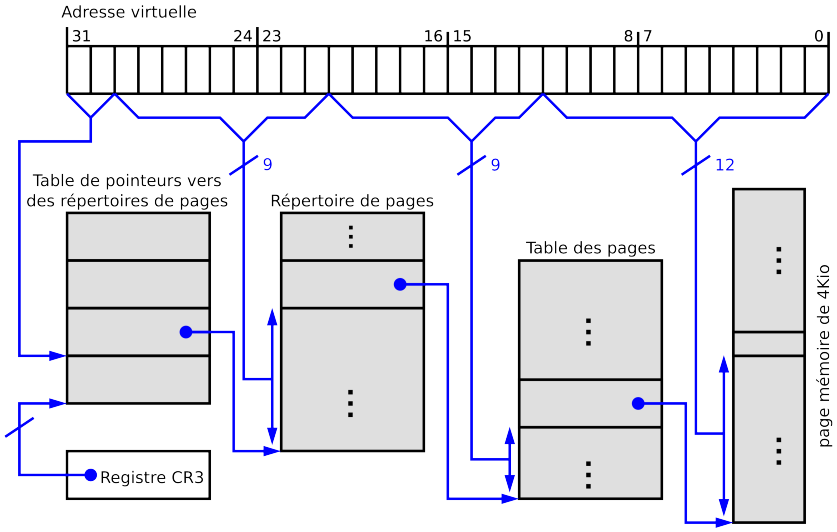


FIGURE 2.6 – Traduction d'adresses

Pour fonctionner, le système de traduction d'adresse a donc besoin des correspondances entre adresse virtuelle et adresse physique pour les adresses de base de toutes les pages mémoire virtuelle. Cette information est stockée dans des tables associées à chaque processus, chaque processus ayant son propre espace d'adressage. La plupart des processus n'utilisant pas toute la mémoire virtuelle disponible, la correspondance des adresses de base est stockée au sein de plusieurs tables, organisées hiérarchiquement. Les bits de poids fort d'une adresse sont alors divisés en plusieurs groupes, chaque groupe constituant un index pour l'une des tables de la hiérarchie. Le groupe avec les bits de poids le plus fort indexe la table racine dont l'entrée correspondante pointe vers une table de deuxième niveau. Le groupe de bits suivant indexe alors cette table et ainsi de suite sur 2 à 4 niveaux. Si un ensemble important d'adresses n'est pas utilisé, cette organisation permet de ne pas allouer la hiérarchie de tables correspondantes en stockant simplement la valeur nulle dans l'entrée pointant sur cette hiérarchie.

Ce mécanisme de mémoire virtuelle signifie qu'une simple lecture nécessite plusieurs accès mémoire, pour effectuer la résolution de l'adresse puis pour la lecture. Pour éviter tout ralentissement, les processeurs disposent d'un cache matériel dédié à la mémorisation des traductions d'adresses. Ce cache s'appelle un cache de traduction d'adresses (« Translation Lookaside Buffer »). Ainsi, la résolution d'adresse ne nécessite la plupart du temps aucun accès à la mémoire car l'adresse traduite est déjà dans le cache.

Généralement, le cache de traduction d'adresses est très petit et n'est composé que d'un seul niveau de cache. Ce n'est en pratique pas un problème car une entrée dans ce cache est suffisante pour traduire toutes les adresses appartenant à la même page mémoire. Or, la

2. Cette figure est un travail dérivé dont l'original a été contribué par RokerHRO. En tant que tel, ce travail est sous licence CC-By 3.0 [ccb12]. Les sources sont disponibles sur simple demande à thomas.preudhomme@celest.fr

taille d'une page mémoire est grande : la plupart du temps la taille d'une page mémoire est de 4 Kio et peut être de 4 Mio voire même 1 Gio dans certaines configurations. Un faible nombre d'entrées permet donc de traduire un grand nombre d'adresses virtuelles. De plus, les traductions d'adresses sont propres à un processus. Lorsqu'un processus est remplacé par un autre, toutes les traductions d'adresses doivent donc être invalidées. Cela se produisant assez régulièrement, un cache de grande taille ne présente qu'un intérêt réduit. Certains mécanisme ad hoc non présentés ici permettent cependant d'éviter une partie de ce surcoût.

2.1.2.9 Recherche d'une donnée

Comme présenté à la section 2.1.2.4, le mécanisme de recherche dans un cache associatif repose sur l'utilisation de l'index d'une adresse pour déterminer dans quelle voie peut être stockée la donnée et de l'étiquette pour vérifier sa présence. Cependant, au regard de l'existence d'espaces d'adressage, il devient nécessaire de préciser sur quelle adresse – réelle ou virtuelle – sont obtenus l'index et l'étiquette. Les deux alternatives sont possibles et le choix se fait séparément pour l'index et l'étiquette. Ainsi l'index peut être obtenu depuis l'adresse virtuelle mais l'étiquette depuis l'adresse physique. Dans ce cas, l'étiquette inclus également les bits d'index de l'adresse physique. En effet, deux adresses virtuelles ayant les mêmes bits d'index peuvent avoir des adresses physiques avec des bits d'étiquette identiques.

Le choix d'utiliser des bits de l'adresse virtuelle ou de l'adresse physique est une question de performance. En dépit d'une latence d'accès très faible pour le cache de traduction d'adresses, l'accès à celui-ci, puis la recherche de la donnée accédée, prend plus de temps qu'un cycle processeur. L'utilisation des bits de l'adresse physique réduit donc les performances du cache. En revanche, l'utilisation de bits de l'adresse virtuelle pose des problèmes de cohérence mémoire de deux types :

— **problèmes de synonymes** (aussi appelés problèmes d'alias) : des adresses virtuelles distinctes correspondent à la même adresse physique ;

— **problèmes d'homonymes** : la même adresse virtuelle correspond à des adresses physiques différentes.

Les problèmes de synonymes surviennent lorsque l'adresse virtuelle est utilisée pour obtenir les bits d'index et qu'il existe plusieurs adresses virtuelles correspondant à la même adresse physique. Dans une telle configuration, il est possible pour une donnée d'occuper deux emplacements d'un cache si celle-ci est accédée par deux adresses physiques différentes. Une incohérence se produit alors dès qu'une des entrées est modifiée. Les problèmes de synonymes peuvent être évités en utilisant l'adresse physique pour les bits d'index ou en empêchant au niveau du système la correspondance de plusieurs adresses virtuelles avec une même adresse physique. Cette dernière solution n'est cependant plus utilisée, son coût d'implémentation étant trop coûteux par rapport au nombre d'occurences de ces problèmes.

Les problèmes d'homonymes sont plus courants car ils découlent de l'utilisation de l'adresse virtuelle pour les bits d'étiquette en présence d'espaces d'adressage. Deux adresses virtuelles peuvent dans ce cas être identiques, partageant ainsi les mêmes bits d'étiquette, et correspondre à deux adresses physiques différentes. Ces problèmes sont généralement évités en utilisant l'adresse physique pour les bits d'étiquette ou en vidant le cache dès que l'espace d'adressage change. Ils peuvent également être évités en attachant des bits identifiant les espaces d'adressage à chaque emplacement du cache.

Abbréviation	Index	Étiquette	Avantages / Inconvénients	Utilisation typique
VIVT	adresse virtuelle	adresse virtuelle	+ rapide − problème de synonymes	cache L1
VIPT	adresse virtuelle	adresse physique	+ plus rapide que PIPT − moins rapide que VIVT	cache L1
PIVT	adresse physique	adresse virtuelle	− lent − problèmes de synonymes	non utilisé
PIPT	adresse physique	adresse physique	− lent	cache L2 ou L3

TABLE 2.2 – Intérêt des différents mécanismes de recherche d'une donnée dans un cache

Quatre combinaisons sont possibles concernant le type d'adresse utilisé pour l'index et l'étiquette. Celles-ci sont présentées dans la tableau2.2. Sur ces quatre combinaisons, l'utilisation de l'adresse physique pour obtenir l'index et de l'adresse virtuelle pour obtenir l'étiquette n'est jamais utilisée car elle ne présente aucun d'intérêt. Cette solution est lente et présente des problèmes d'alias dû à l'utilisation de l'adresse virtuelle pour obtenir l'étiquette. Parmi les trois combinaisons restantes, celles où l'index est obtenu à partir de l'adresse virtuelle sont utilisées pour le cache de premier niveau pour des questions de performances. La dernière combinaison, où l'adresse physique est utilisée pour l'index et l'étiquette, est utilisée dans les caches de second et troisième niveau.

En dépit du coût de traduction d'adresse, l'utilisation de l'adresse physique pour obtenir l'étiquette reste envisageable pour le cache de premier niveau car la traduction peut s'effectuer en même temps que l'index est analysé pour trouver l'ensemble à laquelle la donnée accédée appartient. La recherche d'une ligne de cache n'est alors ralentie que par le temps nécessaire pour finir la traduction d'adresse une fois l'ensemble déterminé.

Malgré l'impact important des décisions concernant la méthode pour rechercher une donnée dans un cache, le seul élément de performance sur lequel un programmeur a le contrôle est la présence de problèmes d'alias. Il est en effet possible pour un programmeur d'empêcher ces problèmes de survenir en évitant qu'un programme n'accède à une même zone mémoire physique depuis plusieurs zones mémoire virtuelles.

2.1.2.10 Caches spécialisés

Dans un processeur, plusieurs composants effectuent des accès mémoire. Outre les accès aux données, le chargement des instructions nécessite aussi des accès à la mémoire. Tous ces accès génèrent des pertes de cycles processeur. Aussi un système de cache est employé pour chacun de ces mécanismes.

Cependant, ces différents accès à la mémoire ont des comportements assez distincts. La lecture des instructions en mémoire se produit par exemple de façon beaucoup plus séquentielle que la lecture des données en mémoire. À cet égard, elle bénéficie d'ailleurs beaucoup plus de la présence d'une unité de préchargement. Les instructions ont également la particularité d'être très rarement modifiées, ce qui contraste avec les données qui sont modifiées régulièrement. De plus, les éléments accédés par chacun des composants se situent à des emplacements bien distincts en mémoire.

Pour prendre en compte les différences qui existent entre les différentes sources d'accès

mémoire, les processeurs modernes ont recours à des **caches spécialisés**. Il existe ainsi un cache pour les données et un cache pour les instructions. Cette séparation des caches n'est réalisée que pour les caches de premier niveau. Les caches de niveau inférieur sont suffisamment grands pour contenir tous les éléments différents sans impact important pour les performances : on parle alors de **cache unifié**. Le cache de traduction d'adresses de deuxième niveau, lorsqu'il y en a un, reste néanmoins distinct du cache unifié.

La séparation des caches de premier niveau impose peu de contraintes sur l'écriture des programmes. Ce dispositif est relativement transparent pour les programmes et a principalement un impact positif vis à vis sur les performances. Le dispositif peut cependant être pénalisant pour les programmes qui n'ont pas un comportement d'accès habituel. Ainsi, un programme dont le code se modifie fréquemment se verra pénalisé au niveau des performances. En effet, les instructions étant supposées changer rarement, les caches d'instructions ne possèdent pas de mécanisme de mise à jour. Si des instructions sont modifiées, la seule façon de mettre à jour le cache est de le purger puis le laisser se remplir à nouveau au fur et à mesure de l'exécution des instructions.

Impact des mécanismes de cache sur les performances Comme en témoigne cette section, de nombreux paramètres influencent l'efficacité des caches matériels ce qui rend difficile l'écriture d'un programme performant. Tous les éléments présentés dans cette section suggèrent de suivre certaines règles dans la disposition des données en mémoire et dans la façon d'y accéder. La prise en compte de ces règles explique en partie les bons résultats des travaux présentés dans la suite de cette thèse.

Parmi les règles à suivre, la plus simple à appliquer est de grouper les données ensemble le plus possible pour minimiser la taille des ensembles de travail. Il est également important que les données fréquemment accédées soient alignées sur la taille d'une ligne de cache pour réduire le nombre de lignes de cache nécessaires à leur stockage. De plus, les données souvent accédées ensemble doivent être proches en mémoire pour augmenter la localité spatiale et il faut privilégier des accès séquentiels pour tirer au mieux parti de l'unité de préchargement. Enfin, les données doivent être réparties en mémoire le plus uniformément possible pour éviter les problèmes de conflits et ainsi réduire le nombre d'évictions inutiles.

2.2 Architectures mémoire des systèmes SMP

De nos jours, l'augmentation de la puissance de calcul des systèmes est permise par la présence de plusieurs unités de calcul : on parle alors de systèmes à multitraitement. Ceux-ci peuvent aussi bien être réalisés avec des unités de calcul toutes identiques qu'avec des unités de calcul différentes. Lorsque les unités de calcul sont identiques, les systèmes sont dits **à multitraitement symétrique** ou simplement **systèmes SMP**. Dans l'autre cas, les systèmes sont dits **à multitraitement asymétrique**. Les systèmes *SMP* étant de loin les plus répandus, c'est ceux-ci que cette thèse étudie.

Comparés aux systèmes n'ayant qu'une unité de calcul, les systèmes *SMP* sont plus difficiles à utiliser de manière efficace. Comme eux, les systèmes *SMP* utilisent des caches matériels pour permettre un accès rapide à la mémoire et sont donc sujets aux mêmes problématiques que celles décrites dans la section précédente. La mémoire étant partagée entre toutes les unités de calcul, ces systèmes sont également concernés par le problème du partage de la mémoire et de ses différents niveaux de cache.

La bonne utilisation des ressources de calcul dont disposent les systèmes *SMP* nécessite d'ajouter aux contraintes liées à la gestion de la mémoire la prise en compte de l'organisation des différentes unités de calcul. Or il existe une multitude d'architectures mémoire différentes au sein des systèmes *SMP*. Si les détails diffèrent d'une architecture à une autre, trois grande familles d'architectures se détachent. Dans cette perspective, cette section détaille les trois types d'architectures des systèmes *SMP*. L'architecture multi-processeurs est l'objet de la première partie. La deuxième partie est dédiée à l'architecture multi-cœurs. Enfin, la troisième partie traite de l'architecture NUMA.

2.2.1 Architecture multi-processeurs

L'architecture multi-processeurs constitue la première approche pour construire des systèmes offrant du multitraitement symétrique. Bien que leur démocratisation se soit produite assez tard, les premiers systèmes à architecture multi-processeurs sont apparus dans les années 1960 avec la sortie du multi-processeur symétrique D825 en 1962 [Wil94]. Comme le nom l'indique, l'architecture multi-processeurs consiste en l'utilisation de plusieurs processeurs au sein d'un même système. Les différents processeurs sont alors tous reliés au bus système (« Front Side Bus ») et partagent l'accès à la mémoire centrale. Cette architecture est présentée dans la figure 2.7.

Bien qu'offrant du parallélisme, l'amélioration des performances offert par une telle architecture est limitée. En effet, le faible couplage des processeurs dans une architecture multi-processeurs occasionne une dégradation des performances dès lors que les processeurs ont besoin de communiquer. La raison de cette dégradation est que, s'agissant de multiples processeurs, aucun cache n'est partagé et la communication entre processeurs nécessite d'utiliser le bus système, et cela quelque soit les processeurs. Or le bus système a une latence d'accès importante, plus grande que l'accès au cache de dernier niveau. En conséquence, toute communication entre deux processeurs réduit les performances du système. Les systèmes ayant une telle architecture sont donc plutôt destinés à exécuter des tâches indépendantes l'une de l'autre.

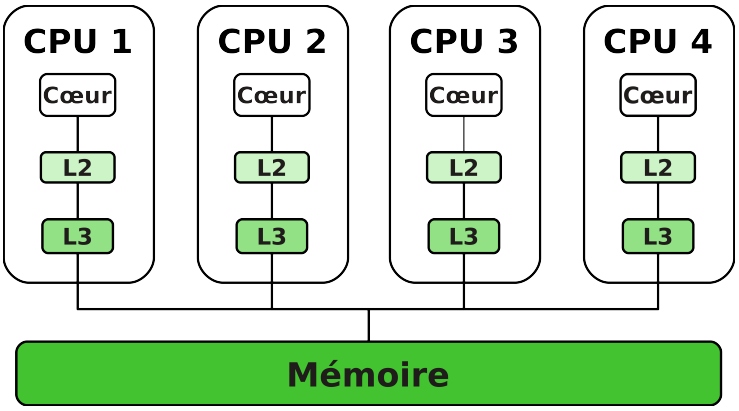


FIGURE 2.7 – Architecture multi-processeurs

De plus, l'architecture multi-processeurs n'améliore le parallélisme que de façon partielle puisque seules les ressources de calcul sont multipliés. Ainsi, toutes les ressources qui

concernent l'accès à la mémoire sont partagées. C'est le cas notamment de la mémoire mais également du bus système qui permet d'accéder à la dite mémoire. Or ces deux composants offrent une bande passante limitée, insuffisante pour permettre un accès à la mémoire à tous les processeurs en même temps. En conséquence, lorsqu'un nombre important de processeurs accèdent à la mémoire, le débit qu'ils obtiennent ne correspond qu'à une fraction du débit maximal qu'ils peuvent atteindre. Le débit obtenu est alors égal à la bande passante offerte par la mémoire et le bus système divisée par le nombre de processeurs effectuant un accès mémoire. Cette limite a également un effet aggravant sur l'importance de la latence de communication entre processeurs. En effet, le bus système ne peut être accédé que par un seul processeur à la fois. Aussi, lorsque plusieurs processeurs essaient d'accéder au bus système les accès sont sérialisés, augmentant d'autant plus les latences de communication inter-processeurs.

Le problème de la bande passante limitée du bus système et de la mémoire centrale a un impact sévère sur l'efficacité de l'architecture multi-processeurs. En effet, le problème est d'autant plus sévère qu'il y a de processeurs qui accèdent simultanément à la mémoire. Or plus le nombre de processeurs dans un système est important, plus le risque d'accès concurrent est élevé. Dès lors, lorsque le nombre de processeurs dépasse les quelques unités, ceux-ci passent la majeure partie de leur temps en conflit d'accès à la mémoire. Une telle architecture limite donc fortement l'accélération qui peut être obtenue en ajoutant des processeurs dans un système.

2.2.2 Architecture multi-cœurs

L'architecture multi-cœurs constitue la deuxième approche pour construire des systèmes offrant du multitraitement symétrique. Cette architecture est beaucoup plus récente comparée à l'architecture multi-processeurs : le premier produit – l'IBM POWER4 – est sorti en 2001 et la démocratisation s'est enclenchée en 2005 avec les premiers processeurs multi-cœurs grand public proposés par Intel et AMD. Cette architecture consiste en un unique processeur possédant plusieurs cœurs de calcul. Cette approche peut ainsi être combinée à une approche multi-processeurs en intégrant dans un même système plusieurs processeurs multi-cœurs.

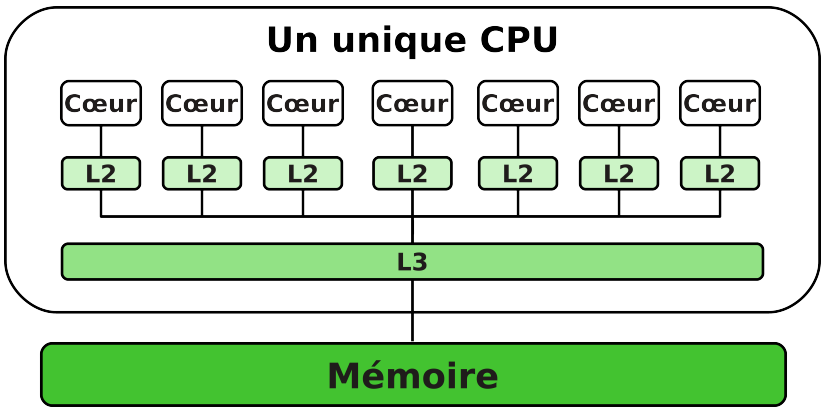


FIGURE 2.8 – Architecture multi-cœurs

Contrairement à l'architecture multi-processeurs, l'architecture multi-cœurs contient des

caches partagés entre plusieurs unités d'exécution. Il peut s'agir d'un cache partagé de second niveau ou de troisième niveau et le partage peut s'opérer entre tous les cœurs du système ou seulement un sous-ensemble. Ainsi, il existe des systèmes avec plusieurs caches de deuxième niveau, chacun partagé entre un sous-ensemble distinct des cœurs disponibles. Dans d'autres systèmes (voir figure 2.8), le partage se produit au niveau du cache de troisième niveau qui est commun à tous les cœurs. L'interconnexion des différents cœurs se produit par le biais d'un canal de communication interne au processeur, et non via le bus système. Le résultat est une communication bien plus rapide que dans le cas d'une architecture multi-processeurs, ce canal de communication offrant une meilleure bande passante et une latence plus faible que le bus système.

Cette architecture offre également une légère amélioration au problème de bande passante vers la mémoire. En effet, l'existence de caches matériels partagés permet d'offrir une donnée à plusieurs cœurs de calcul en n'ayant besoin que d'un seul accès à la mémoire : celui pour copier la donnée dans le cache. Ainsi si tous les cœurs accèdent à une même zone mémoire, le nombre d'accès à celle-ci est réduit au nombre de caches partagés dans le système.

Cependant, dans le cas général où les cœurs accèdent simultanément à des zones mémoire distinctes, le problème de bande passante rencontré par l'architecture multi-processeurs reste présent. En effet, au lieu de se produire sur le bus système, la contention se produit alors sur le canal de communication interne au processeur. Bien que celui-ci soit plus rapide, sa bande passante reste toujours insuffisante pour permettre à tous les cœurs d'accéder à la mémoire en même temps. Par ailleurs, ce canal est déjà utilisé pour communiquer entre les différents niveau de cache, événement qui est plus fréquent qu'un accès à la mémoire. Ce canal est donc naturellement plus sollicité que le bus système.

L'architecture multi-cœurs parvient ainsi à résoudre le problème de latence de communication entre les processeurs auxquel est confrontée l'architecture multi-processeurs. En revanche, le problème de bande passante reste présent, quoique légèrement atténué.

2.2.3 Architecture NUMA

L'architecture **NUMA** constitue la dernière approche visant à augmenter les ressources de calcul des systèmes informatiques par le biais du parallélisme. La dénomination de cette architecture est un acronyme qui signifie « accès non uniforme à la mémoire » (« **Non Uniform Memory Access** »). Cet acronyme se réfère à l'hétérogénéité des latences d'accès à la mémoire. En effet, dans cette architecture la mémoire centrale est partitionnée en plusieurs modules, chacun étant connecté à un ensemble différent d'unités de calcul. Cette architecture comporte alors plusieurs nœuds connectés entre eux, un **nœud** étant la combinaison d'un module mémoire et d'un ensemble d'unités de calcul. Cette organisation des unités de calcul et de la mémoire est présentée dans la figure 2.9.

Pour une unité de calcul donnée, la latence d'accès à la mémoire est donc fonction du module mémoire accédé. Si la zone mémoire accédée par une unité de calcul provient de la mémoire du nœud dont elle fait partie, l'accès est dit local et a une faible latence. Si la zone mémoire est gérée par un autre module mémoire, l'accès doit transiter par l'interconnexion entre les nœuds avant d'atteindre la mémoire : l'accès est dit distant et a une latence importante. De plus, les nœuds ne sont pas tous reliés les uns aux autres : accéder à un module mémoire distant peut donc requérir de traverser plusieurs nœuds.

Une architecture est dite NUMA dès lors que la mémoire centrale est composée de plu-

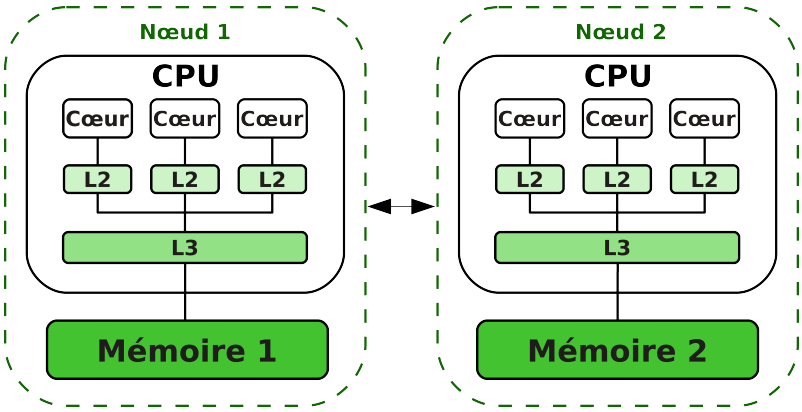


FIGURE 2.9 – Architecture NUMA

sieurs modules mémoire connectés à différents ensembles d'unités de calcul. Il est ainsi possible de combiner cette approche avec les deux approches précédentes. Il existe par exemple des systèmes possédant plusieurs nœuds mémoire, chaque nœud étant composé de plusieurs processeurs comprenant chacun plusieurs cœurs de calcul.

L'approche NUMA a pour but de résoudre le problème de bande passante inhérents aux architectures multi-processeurs et multi-cœurs. La solution proposée consiste donc à associer à chaque processeur un module mémoire distinct qui lui est directement connecté. Ainsi, tous les processeurs peuvent accéder à leur module mémoire simultanément sans se gêner l'un l'autre puisque aucun bus n'est partagé. Cependant, le problème n'est résolu qu'au niveau des nœuds. En effet, il y a toujours contention d'accès à la mémoire ce qui limite le nombre d'unités de calcul par nœud. De plus, le problème n'est résolu au niveau des nœuds que si les unités de calcul dans ceux-ci accèdent uniquement à leur module mémoire local. Dans le cas contraire, les accès mémoire doivent emprunter les canaux de communication qui relient les nœuds entre eux et une contention est susceptible d'apparaître.

Malgré les limitations restantes concernant un accès concurrent à la mémoire par plusieurs processeurs, l'architecture NUMA présente une réelle amélioration par rapport aux autres approches. Elle permet ainsi d'obtenir une meilleure accélération pour un nombre de processeur donné et permet donc d'obtenir une accélération plus importante en intégrant un plus grand nombre de processeurs dans un système. Cependant, cette approche présente de nouvelles difficultés d'utilisation liées à la différence de latence d'accès à la mémoire en fonction de la zone mémoire accédée. Ainsi, un programme qui nécessite plus de mémoire qu'un module ne peut en fournir doit fragmenter ses données en deux ensembles : les données souvent accédées sont placées sur le module mémoire local et le reste des données est placé sur les modules distants. En pratique, cette situation est susceptible de se produire assez fréquemment dans les systèmes à architecture NUMA étant donné les contraintes de placement des programmes. De nombreux programmes nécessitant une quantité moyenne de mémoire peuvent en effet partager le même module mémoire car ils s'exécutent sur le même processeur. Ceci se produit par exemple lorsqu'une ou plusieurs tâches monopolisent de nombreuses ressources de calcul, forçant toutes les autres tâches à s'exécuter sur un petit nombre de processeurs.

Impact des architectures mémoire sur les performances Les systèmes possédant plusieurs unités de calcul apportent aujourd'hui une réelle amélioration des performances dès lors qu'un programme peut être parallélisé. Cependant, ces systèmes sont plus difficiles à exploiter car il faut prendre en compte la latence d'accès à la mémoire et sa bande passante, et placer les données en mémoire en conséquence. De plus, ces systèmes diffèrent au niveau des détails au sein de trois grandes familles de systèmes SMP, dont les caractéristiques sont résumées dans le tableau 2.3.

Familles de SMP	Latence de communication entre les unités de calcul	Concurrence d'accès à la mémoire
Systèmes multi-processeurs	forte	forte
Systèmes multi-cœurs	faible	forte
Systèmes NUMA	faible (intra-nœuds) forte (inter-nœuds)	faible

TABLE 2.3 – Caractéristiques des différentes familles de systèmes SMP

Dans tous les cas, il convient donc d'organiser les accès aux données de manière à tirer le meilleur parti possible des hiérarchies de latence et de débit formées par les caches et la mémoire. En particulier le nombre d'accès à la mémoire doit être réduit le plus possible afin d'éviter les contentions. Les programmes s'exécutant sur des systèmes multi-processeurs et NUMA doivent également veiller à réduire respectivement la communication entre processeurs et entre les nœuds étant donné la latence de telles communications.

2.3 Système de cohérence de la mémoire

L'existence de plusieurs unités de calcul dans un système a des implications importantes sur les performances de celui-ci. D'un côté, les applications parallélisables peuvent ainsi bénéficier d'une accélération dont l'importance croît avec le nombre d'unité de calcul. D'un autre côté, obtenir une accélération linéaire vis à vis du nombre d'unités de calcul requiert des précautions supplémentaires qui rendent l'optimisation des programmes encore plus compliquée. En particulier, il est nécessaire de réduire le nombre d'accès à la mémoire et d'éviter les communications entre unités de calcul à cause des limitations des différentes architectures comprenant plusieurs unités de calcul.

L'impact du multitraitement sur les performances ne se limite cependant pas à l'architecture mémoire des systèmes. Il provient également du système de cohérence des caches qui permet au multitraitement d'être transparent pour les applications. Le multitraitement diffère en effet du traitement séquentiel en ce qu'une donnée peut être présente simultanément dans plusieurs caches matériels, chaque unité de calcul disposant de ses propres caches matériels. Il est alors nécessaire de garantir une cohérence de la mémoire au niveau des caches. Or cette cohérence nécessite de nombreux échanges de messages entre les différentes unités de calcul, ajoutant contention et latence.

Cette section étudie le fonctionnement du système de cohérence des caches ainsi que les conséquences de ce système sur les performances des applications dans les systèmes offrant du multitraitement. La première partie de cette section présente les garanties que le système de cohérence des caches à pour but d'offrir. La seconde partie décrit le fonctionnement du

système de cohérence des caches MOESI qui est utilisé dans les systèmes actuels. Enfin, la troisième et dernière partie examine les conséquences de ce fonctionnement sur les performances des applications au travers de scénarios typiques de partage de données.

2.3.1 Origine des incohérences mémoire

Outre l'organisation des caches et de la mémoire, la mise en œuvre du multitraitement doit également gérer les conséquences de la présence de plusieurs caches et modules mémoire sur la validité des programmes. Dans un système n'ayant qu'une seule unité de calcul, la lecture d'une donnée est garantie par construction de retourner la dernière valeur y ayant été écrite en mémoire. À l'inverse, dans un système possédant plusieurs unités de calcul, aucune garantie n'est naturellement assurée et la lecture peut retourner une valeur provenant d'une écriture plus ancienne de cette donnée.

2.3.1.1 Distribution des caches

Dans un système ayant plusieurs unités de calcul, chaque unité possède son propre cache matériel de premier niveau. Dans une telle configuration, deux processus P_1 et P_2 s'exécutant sur deux unités de calcul distinctes peuvent travailler simultanément sur une même donnée à partir de leurs caches respectifs C_1 et C_2. Une modification de cette donnée par le processus P_1 ne la modifie ainsi que dans le cache C_1. Si cette donnée est ensuite accédée par P_2, la lecture retourne retourne alors une valeur différente de celle qui a été écrite par le processus P_1, la lecture étant faite à partir du cache C_2.

Bien qu'un système avec une seule unité de calcul possède également plusieurs caches matériels, la situation est différente. Dans ce cas de figure, un seul cache est accédé par les applications : le cache de plus haut niveau. Une incohérence entre les caches est possible mais elle n'est pas visible des applications. Un exemple de scénario est le cas où une donnée est présente dans les caches de premier et second niveau et qu'un processus la modifie : celle-ci n'est alors modifiée que dans le cache de premier niveau. L'incohérence reste inaperçue des applications car tout accès ultérieur à cette donnée sera effectuée dans le cache de premier niveau. Finalement, cette incohérence se résout lorsque la ligne de cache qui contient la donnée dans le cache de premier niveau est évincée. Le contenu de celle-ci est copié dans le cache de second niveau et une seule copie cohérente avec ce que les processus ont pu lire existe alors.

2.3.1.2 Exécution dans le désordre

Un autre problème de cohérence résulte du mécanisme d'**exécution dans le désordre** des instructions (« out-of-order execution »), que mettent en œuvre les unités de calcul modernes. Ce changement de l'ordre des instructions opéré par les unités de calcul permet d'améliorer les performances en évitant qu'une instruction en attente de données bloque les instructions suivantes. Les instructions sont alors réordonnées pour exécuter celles qui sont prêtes, retardant l'exécution de l'instruction bloquée au moment où les données dont elle dépend sont disponibles. Par exemple deux instructions de lecture peuvent être réordonnées si la donnée lue par la deuxième instruction est déjà disponible. Pendant ce temps, la donnée lue par la première instruction est accédée en mémoire, masquant ainsi la latence d'accès à la mémoire. La réorganisation d'instructions n'est possible que lorsque les instructions n'ont pas

de dépendances entre elles, c'est à dire quand l'exécution des instructions réordonnées ne dépends pas du résultat des instructions qui les précèdent ou les succèdent.

Plusieurs types de dépendance entre les données doivent être conservées lors des réorganisations :

— **lecture après écriture** : la lecture doit retourner la valeur précédemment écrite ;

— **écriture après lecture** : la lecture **ne doit jamais** retourner la valeur d'une écriture qui succède à la lecture ;

— **écriture après écriture** : les écritures doivent s'exécuter dans l'ordre.

Pour que ces dépendances restent satisfaites malgré la réorganisation des instructions, les unités de calcul possèdent des tampons de lectures et d'écritures. Les lectures et écritures sont alors faites dans ces tampons et si une violation de dépendance est détectée, les instructions fautives sont réexécutées. L'existence de ces tampons permet également une optimisation supplémentaire en exécutant les instructions qui suivent un branchement avant de connaître si le branchement sera effectué ou non : c'est le principe d'exécution spéculative. Si le branchement est effectué, il suffit alors de vider les tampons, simulant ainsi le fait qu'aucune instruction n'ait été exécutée.

Le mécanisme d'exécution dans le désordre n'est pas un élément nouveau des systèmes *SMP*. Il est en effet apparu pour la première fois dans un produit commercialisé en 1990 au sein du microprocesseur POWER1, celui-ci ne disposant que d'une unité de calcul. Il s'est largement répandu dans la suite des années 90. Aucun problème de cohérence ne résulte de la présence de ce mécanisme dans les systèmes n'ayant qu'une unité de calcul car des dispositifs de sûreté y sont incorporés garantissant que le résultat est identique à une exécution dans l'ordre.

En revanche, des incohérences peuvent se produire dans les systèmes *SMP* car les unités de calcul ne détectent que les violations de dépendances locales. Elles ne peuvent donc pas détecter des violations qui impliquent plusieurs unités de calcul. Ainsi, le code d'exclusion mutuelle présenté dans la figure 2.10 ne fonctionne pas correctement lorsque les unités de calcul mettent en œuvre l'exécution dans le désordre, même si tous les caches matériels sont partagés. En effet, les écritures effectuées au début du code par chacun des processus sont susceptibles d'être exécutées après les lectures effectuées dans le test de branchement car ces deux accès mémoire portent sur des variables différentes. Or ce code repose sur l'exécution en séquence des instructions qui le compose.

Fonction process1

```
a ⟵ 1 ;
si b = 0 alors
    section critique ;
    a ⟵ 0 ;
sinon
    a ⟵ 0 ;
    Recommencer ;
fin
```

Fonction process2

```
b ⟵ 1 ;
si a = 0 alors
    section critique ;
    b ⟵ 0 ;
sinon
    b ⟵ 0 ;
    Recommencer ;
fin
```

FIGURE 2.10 – Exclusion mutuelle sans prise de verrou

2.3.2 Solutions pour le maintien de la cohérence

Les deux problèmes de cohérence de la mémoire présentés ci-avant sont orthogonaux l'un à l'autre : il faut donc deux solutions distinctes afin d'offrir un comportement cohérent de la mémoire aux applications. L'un des problèmes concerne la gestion de la mémoire à l'intérieur des unités de calcul et nécessite donc une **cohérence interne** aux unités de calcul. L'autre concerne la gestion de la mémoire à l'extérieur des unités de calcul, au sein des caches matériel. Elle nécessite en conséquence une **cohérence externe** aux unités de calcul.

Exécution dans le désordre Le problème d'exécution dans le désordre ne peut être résolu car il demanderait aux unités de calcul d'avoir accès aux tampons de lectures et d'écritures de chacune des autres unités afin de détecter les dépendances dans les accès mémoire des différentes unités. Cependant, cette forme d'incohérence se produit assez rarement dans les programmes, aussi il est acceptable de fournir un moyen de désactiver l'exécution dans le désordre ponctuellement. En pratique, la désactivation prends la forme d'une instruction qui force tous les accès mémoire effectués par les instructions avant celle-ci à être terminés avant qu'un nouvel accès mémoire puisse être commencé. Une telle instruction s'appelle une **barrière mémoire**.

Distribution des caches En revanche, la cohérence des caches est possible grâce à l'utilisation d'un protocole de cohérence des caches. Un protocole de cohérence atteint son but en réalisant un système de verrou lecteurs / écrivains : pour une ligne de cache et un instant donné, il ne peut exister qu'un seul écrivain ou plusieurs lecteurs. Lorsqu'un lecteur accède à une donnée qui vient d'être écrite, il la copie alors dans son cache depuis le cache de l'écrivain précédent, garantissant ainsi la cohérence des valeurs lues pour cette donnée. Le fonctionnement détaillé d'un tel protocole est décrit en détail dans la section suivante.

La cohérence offerte aux applications par le système est caractérisée par le modèle de cohérence suivi. Les premiers systèmes ayant plusieurs unités de calcul avaient une architecture multi-processeurs qui ne permettaient pas l'exécution dans le désordre. Le modèle de cohérence ainsi fourni était la **cohérence séquentielle** [Lam79]. Ce modèle de cohérence garantit que le résultat du point de vue de la mémoire de n'importe quelle exécution est le même que si les accès mémoire de tous les processeurs étaient exécutés dans un ordre séquentiel, et que pour chaque processeur ses accès mémoire apparaissent dans cette séquence dans l'ordre où ils apparaissent dans le programme. L'ordre séquentiel dont il est question est unique pour une exécution donnée. Cela implique notamment que tous les processeurs voient les accès mémoire dans le même ordre.

Les unités de calcul actuelles permettent maintenant pour la plupart d'exécuter des instructions dans le désordre aussi les systèmes *SMP* actuels offrent une cohérence moins forte telle la **cohérence faible** [DSB86, AH90] ou la **cohérence au relâchement** [KSB95, DSB88, GLL+90, Mos93, Kel95]. Ces deux modèles de cohérence définissent certains accès comme des points de synchronisation pour la mémoire. Tous les accès à la mémoire entre ces points de synchronisation peuvent alors être vus dans n'importe quel ordre par les différentes unités de calcul. La différence entre ces deux modèles tient dans la granularité des points de synchronisation. Dans la cohérence faible les points de synchronisation sont globaux à toute la mémoire. Lorsqu'un point de synchronisation est rencontré, tous les accès antérieurs sont répercutés dans toutes les unités de calcul. En revanche, les points de synchronisation dans la cohérence au relâchement sont limité à une zone mémoire limitée.

Des définitions plus précises et formelles des modèles de cohérences fournis par les systèmes *SMP* peuvent également être trouvées pour les architectures x86-CC [SSN+09] ainsi que pour les architectures ARM et POWER [AFI+09]. Ces publications détaillent en particulier comment ces architectures dévient en quelques points de la documentation fournie par les constructeurs eux-mêmes.

2.3.3 Fonctionnement du système de cohérence des caches

Le maintien d'une vue cohérente de la mémoire pour les caches nécessite que ceux-ci communiquent entre eux afin qu'une modification soit propagée dans l'ensemble des caches qui possèdent la donnée. Cette communication est décrite par un algorithme appelé protocole de cohérence des caches. Suite à des améliorations visant à réduire la quantité de communication nécessaire au maintien de la cohérence, plusieurs protocoles ont émergé. Parmi ceux-ci, le protocole MOESI [AMD12] est celui qui semble s'imposer. Il est par exemple utilisé au sein des systèmes x86 modernes.

Le principe de ces protocoles est d'interdire à deux caches de contenir deux valeurs différentes pour une même donnée, ou pour être plus exact pour la même ligne de cache. Lorsqu'un cache souhaite modifier une donnée, il commence par prendre possession de celle-ci, c'est à dire que sa copie devient la valeur de référence pour tous les caches. Il peut alors modifier la donnée en toute tranquillité, étant assuré que les autres caches devront récupérer cette nouvelle valeur en cas de nouvel accès à cette ligne de cache.

Les protocoles de cohérence peuvent se représenter sous forme de graphe à états, celui-ci indiquant l'évolution de la cohérence d'une ligne de cache d'un cache donnée. Chaque état exprime non seulement la validité de la ligne mais aussi son degré de partage avec les autres caches. Les transitions correspondent, elles, aux différents accès possibles à cette ligne – lecture ou écriture – faits localement ou par une autre unité de calcul. Le même graphe s'applique à toutes les lignes de cache et définit donc les combinaisons d'états valides pour une ligne de cache sur les différents caches. C'est ainsi que la cohérence de la mémoire est maintenue.

Le protocole MOESI possède cinq états : *modifiée* (« modified »), *possédée* (« owned »), *exclusive* (« exclusive »), *partagée* (« shared ») et *invalide* (« invalid »). Ce sont les initiales des noms anglais de ces états qui donnent le nom du protocole. Les accès à la mémoire responsables des transitions d'un état à un autre sont caractérisés par leur type, lecture ou écriture, et leur provenance – cache local ou distant. Le graphe à états pour le protocole MOESI est présenté dans la figure 2.11.

Les états se comprennent de la façon suivante :

— état **invalide** : la ligne n'est pas présente dans le cache ou son contenu n'est pas à jour et ne peut donc être utilisé ;

— état **exclusive** : la ligne est à jour uniquement dans ce cache ;

— état **partagée** : la ligne est à jour dans ce cache ainsi que dans au moins un cache distant ;

— état **modifiée** : la ligne est modifiée par rapport au contenu dans la mémoire centrale et n'est à jour dans aucun des autres caches ;

— état **possédée** : la ligne est modifiée par rapport au contenu dans la mémoire centrale et est à jour dans au moins un autre cache.

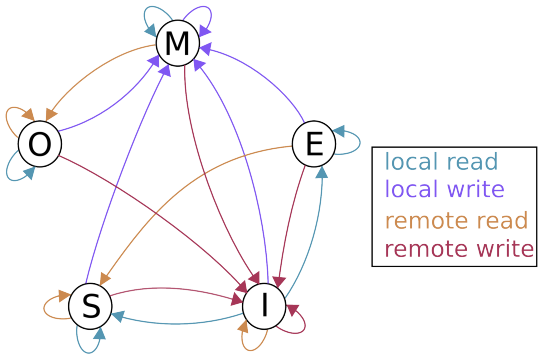


FIGURE 2.11 – Protocole de cohérence des caches MOESI

L'ensemble des combinaisons d'états possibles pour une ligne de cache donnée dans deux caches est alors donné par le tableau 2.4, le protocole MOESI assurant d'éviter toute combinaison incohérente (marquées en rouge dans ce tableau).

	Modifiée	Exclusive	Possédée	Partagée	Invalide
Modifiée	non	non	non	non	oui
Exclusive	non	non	non	non	oui
Possédée	non	non	non	oui	oui
Partagée	non	non	oui	oui	oui
Invalide	oui	oui	oui	oui	oui

TABLE 2.4 – Combinaisons d'états valides entre deux caches pour une ligne de cache donnée

Le fonctionnement du protocole se comprend plus aisément en considérant le cas d'une ligne de cache et de deux caches *C1* et *C2*, présenté dans le tableau 2.5. Initialement, la ligne de cache n'est disponible dans aucun des deux caches, aussi son état est *invalide* dans chacun d'eux (étape 1). Suite à une lecture dans le cache *C1*, la ligne entre alors dans l'état *exclusive* pour ce cache (étape 2). À ce stade, la ligne est toujours dans l'état *invalide* dans le cache *C2*. Un accès en lecture au niveau du cache *C2* mène à changer l'état de la ligne dans les caches *C1* et *C2* à l'état *partagée* (étape 3). Si la ligne est modifiée dans le cache *C2*, elle entre alors dans l'état *modifiée* dans celui-ci tandis qu'elle repasse à l'état *invalide* dans le cache *C1* (étape 4) : la cohérence est ainsi assurée. Enfin, lorsque le cache *C1* cherche à obtenir la nouvelle valeur, la ligne entre dans l'état *partagée* sur le cache *C1* tandis qu'elle entre dans l'état *possédée* sur le cache C2 (étape 5) : la ligne a été mise à jour dans celui-ci mais la modification n'a pas été répercutée en mémoire centrale.

Le protocole de cohérence des caches n'est pas toujours suffisant pour écrire des algorithmes parallèles. Tout comme des instructions de barrière mémoire sont nécessaires pour garantir l'exactitude de certains algorithmes, il est parfois requis de pouvoir modifier de façon atomique une donnée en fonction de sa valeur actuelle. Le protocole MOESI ne permet pas cela car la ligne de cache correspondante est susceptible d'être modifiée par un autre cache entre une lecture et une écriture, la ligne étant alors invalidée. Pour permettre ce comportement, les unités de calcul possèdent des instructions permettant d'enchaîner une instruction de lecture et une instruction d'écriture de façon atomique et de signaler une er-

Étape	1	2	3	4	5
Action	Situation initiale	Lecture dans C1	Lecture dans C2	Écriture dans C2	Lecture dans C1
Cache C1	Invalide	Exclusive	Partagée	Invalide	Partagée
Cache C2	Invalide	Invalide	Partagée	Modifiée	Possédée

TABLE 2.5 – Exemple de mise en cohérence d'une ligne de cache entre deux caches

reur si cela n'était pas possible. Une implémentation possible de telles instructions consiste à s'appuyer sur le protocole MOESI et d'empêcher l'invalidation de la ligne de cache le temps de l'instruction. Ceci est réalisé en plaçant la ligne de cache dans l'état *exclusive*, invalidant alors la ligne de cache dans les autres caches. Pour accéder à la ligne, ceux-ci devront donc la récupérer auprès du cache effectuant une mise à jour atomique, celui-ci pouvant retarder sa réponse aussi longtemps que nécessaire pour terminer la mise à jour.

2.3.4 Conséquences pour les performances

Coût du protocole MOESI Dans le graphe à états représentant le protocole MOESI, les transitions sont la conséquence d'événements liés à des accès à la mémoire. Ces accès peuvent être locaux s'ils sont le fait du processeur associé au cache considéré, ou bien distants si l'accès est effectué par l'un des autres caches. Pour qu'un cache réagisse à un accès distant, un message doit être envoyé. Ainsi, lorsqu'une ligne de cache est modifiée, un message d'invalidation est envoyé de façon synchrone à tous les autres caches pour qu'ils placent la ligne dans l'état *invalide*. Dans le cas où la ligne modifiée était initialement dans l'état *invalide*, ce message s'accompagne d'une réponse envoyée par les autres caches contenant le contenu à jour de cette ligne de cache. Cette réponse est nécessaire car la ligne de cache est susceptible de n'être que partiellement modifiée, auquel cas il est important que la partie non modifiée soit à jour. Ce message d'invalidation s'appelle **RFO**, ce qui signifie requête pour possession (« **Request For Ownership** ») voire lecture pour possession (« **Read For Ownership** »). Le tableau 2.6 récapitule les transitions responsables du coût du protocole de cohérence MOESI de par les messages qu'elles nécessitent.

Type d'accès \ État de la ligne	Modifiée	Exclusive	Possédée	Partagée	Invalide
Lecture	local	local	local	local	requête
Écriture	local	local	requête	requête	requête

TABLE 2.6 – Utilisation de requêtes en fonction de l'état de la ligne de cache locale

De même, dans le cas du protocole MESI qui ne dispose pas de l'état *possédée*, un changement d'état de *modifiée* à *partagée* nécessite d'écrire la ligne de cache modifiée en mémoire ce qui est très coûteux. Cette écriture est nécessaire car la sémantique de l'état *partagée* est que la ligne de cache est à jour dans plusieurs caches et dans la mémoire. Enfin, des messages synchrones sont également nécessaires dans le cas d'un défaut de cache pour qu'une ligne de cache dans l'état *invalide* demande la version à jour à la mémoire ou aux autres processeurs

pour récupérer ladite version.

Réduction du nombre de messages Les messages générés pour maintenir la cohérence entre les caches ont un coût important pour les performances, tant du point de vue de la saturation de la bande passante que de la latence d'accès au données. Aussi, des efforts ont été effectués pour réduire leur nombre. Le protocole MOESI est le résultat de ces efforts.

Du point de vue de la cohérence, seuls les états *modifiée*, *partagée* et *invalide* du protocole MOESI sont nécessaires. Il existe d'ailleurs un protocole de cohérence MSI plus ancien que MOESI qui fonctionne exactement de cette façon. Les deux états supplémentaires que sont *exclusive* et *possédée* n'existent donc que dans le but de réduire le nombre de messages échangés entre les processeurs. Ainsi, l'état *exclusive* permet d'éviter un message RFO lorsque le cache en question sait qu'aucun autre cache ne possède cette ligne. De même, l'existence d'un état *possédée* permet de ne pas mettre à jour la mémoire centrale lorsqu'une ligne de cache dans l'état modifiée est demandée par un autre cache. Dans ce cas, la ligne est dans l'état *partagée* dans les caches qui ont reçu une copie et dans l'état *possédée* dans le cache qui a effectué une modification. Ainsi, si ce dernier souhaite évincer cette ligne de cache, il sait qu'il doit d'abord propager cette donnée en mémoire. Le tableau 2.7 résume la signification des états du protocole MOESI vis à vis de la propagation de modifications en mémoire et de la présence de lignes de cache dans plusieurs caches.

	Modifiée	Exclusive	Possédée	Partagée	Invalide
Mémoire cohérente	non	non	oui	oui	oui
Ligne partagée	non	oui	non	oui	indéfini

TABLE 2.7 – Cohérence des caches et de la mémoire en fonction des états

Identification du destinataire des messages Pour un nombre important des messages générés par le protocole MOESI, le destinataire final du message n'est pas connu à l'avance. En effet, les caches ne disposent pas d'informations sur la localisation des lignes de cache. C'est le cas lors de la lecture d'une ligne invalide, mais aussi lorsqu'un cache souhaite invalider les copies d'une ligne après une écriture. Deux approches existent pour gérer ces messages : l'approche par **espionnage** (« **snooping** ») et l'approche par **répertoire** (« **directory** »).

L'approche par espionnage consiste à envoyer ces messages à tous les caches. Ces derniers ont alors besoin de surveiller le médium de communication, qu'il s'agisse du bus système ou du canal de communication entre caches, pour détecter ces messages et y réagir. La réponse au cache distant peut être d'envoyer une ligne de cache s'il souhaite y accéder mais ne la possède pas ou bien de marquer une ligne comme *invalide*.

L'approche par répertoire consiste à disposer pour chaque nœud mémoire d'un système appelé répertoire qui mémorise l'état des lignes de cache du nœud mémoire. Le répertoire mémorise aussi pour chaque ligne l'identité d'un cache possédant cette ligne. Le répertoire sert alors d'intermédiaire à tout message pour mettre en contact les caches devant échanger un message et met à jour ses informations au passage. Ce faisant, des messages point à point sont utilisés à la place de diffusion pour toutes les transitions sauf dans le cas où une ligne de cache est modifiée et qu'elle est présente dans plusieurs caches.

Des deux solutions, l'approche par espionnage est plus coûteuse en bande passante. Par

contre, l'approche par répertoire augmente la latence puisque les caches doivent d'abord contacter le répertoire avant de contacter le destinataire final. Les architectures NUMA récentes constituées de plusieurs processeurs multi-cœurs tendent à combiner ces deux approches : l'approche par espionnage est utilisée au sein d'un processeur entre les différents cœurs de calcul tandis que l'approche par répertoire est utilisée pour la cohérence entre les nœuds mémoire.

2.3.5 Impact des protocoles de cohérence mémoire sur les performances

Le coût de la mise en œuvre de la cohérence ne peut être évité dès lors que des données sont partagées. Cependant, au niveau des applications, il existe des solutions pour réduire le besoin de cohérence d'un algorithme. La première de ces solutions consiste à éviter les instructions de barrière mémoire et de lecture / écriture atomique. Bien que de nombreuses structures classiques ne peuvent être manipulées de manière concurrente sans avoir recours à ces instructions [AGH+11], il est possible d'utiliser d'autres structures ne présentant pas de telles contraintes. En particulier, une attention spéciale doit être portée sur les instructions de lecture / écriture atomiques car celles-ci imposent un coût pour tous les caches voulant accéder à la ligne de cache modifiée. Les modifications atomiques suivent en effet un modèle de cohérence séquentielle : elles bloquent tout accès à la ligne concernée jusqu'à ce que la modification soit complète. À l'opposé, les instructions de barrière mémoire n'impactent que le processeur qui l'effectue.

Une deuxième solution consiste à éviter le problème du **faux partage**. Ce problème désigne le maintien excessif de la cohérence lorsque plusieurs variables faisant partie de la même ligne de cache sont accédées de façon concurrente. Lorsque cela se produit, les modifications de la variable nécessitent d'invalider toutes les autres copies de la ligne, invalidant du même coup les autres données présentes dans la ligne. Deux unités de calcul peuvent alors se ralentir mutuellement alors qu'elles n'utilisent pas réellement les mêmes données. Ce problème provient de la granularité du protocole de cohérence qui agit au niveau de la ligne de cache. Ainsi, si une variable est souvent modifiée il est préférable de la positionner en mémoire telle qu'elle se retrouve dans une ligne de cache différente de toute autre donnée.

Enfin, il faut éviter le partage de données en lecture / écriture. En effet, cette configuration génère de nombreux échanges de messages sur certaines mises en œuvre, tant pour la propagation des nouvelles valeurs que pour l'invalidation des messages. C'est pourtant une configuration très fréquente puisque c'est celui d'une variable de synchronisation. La solution consiste donc à augmenter le nombre d'actions effectuées entre deux synchronisations afin de réduire le nombre de synchronisations par actions effectuées et donc le nombre de messages pour un problème donné à traiter.

2.4 Conclusion

En raison de la différence de fréquence entre les unités de calcul et les mémoires de type DRAM, l'accès à la mémoire a un coût important pour les programmes. L'écriture de programmes efficaces nécessite donc de prendre un soin particulier à réduire le nombre d'accès mémoire. L'existence de caches matériels permet de réduire de façon significative la latence d'accès à la mémoire mais rend plus complexe sa gestion en forçant les programmeurs à

suivre un grand nombre de règles.

La démocratisation des architectures offrant du multitraitement permet une augmentation théorique de la puissance de calcul considérable mais au prix d'une complexité accrue pour la programmation. En effet, l'architecture mémoire de ces systèmes tend à réduire la bande passante disponible pour une unité de calcul donnée, augmentant d'autant les contraintes d'accès à la mémoire. De plus, le protocole de cohérence des caches permettant de masquer aux programmes l'aspect réparti des caches matériels a lui aussi un impact sur les accès mémoire en raison des nombreux messages qu'il génère. Tous ces changements nécessitent donc de repenser les algorithmes existants afin de mieux tirer parti de ces architectures par le biais d'une réduction drastique des accès mémoire.

Chapitre 3

Algorithme de communication inter-cœurs BatchQueue

Sommaire

AVEC le développement des architectures permettant le multitraitement, en particulier des architectures multi-cœurs, les ordinateurs deviennent de plus en plus des systèmes répartis. De systèmes n'exécutant qu'une seule tâche à la fois, les ordinateurs se sont dotés de la capacité d'exécuter plusieurs tâches simultanément. Certains systèmes actuels – les systèmes NUMA – sont ainsi composés de plusieurs nœuds, chaque nœud disposant d'une ou plusieurs unités de calcul, d'un module mémoire et d'un point d'accès aux périphériques qui lui sont propres. Dans un tel cas de figure, seuls les périphériques sont alors partagés par les différents nœuds. Les nœuds sont donc très autonomes et se comportent comme des systèmes répartis.

Les systèmes NUMA se démarquent cependant des systèmes répartis formés de plusieurs ordinateurs par le fait que le matériel fournit aux applications un accès transparent

à l'ensemble de la mémoire. Cet accès, désigné sous le terme de mémoire partagée répartie, permet à des applications prévues pour des systèmes non répartis de s'exécuter sur ces systèmes sans avoir à effectuer de changement. Cependant, l'accès aux différents modules mémoire n'est transparent que du point de vue de l'usage, il ne l'est pas du point de vue des performances. Un accès pour un cœur à un module mémoire appartenant au même nœud est nettement plus rapide qu'un accès à un module mémoire appartenant à un autre nœud. La proximité physique des modules mémoires rend la différence plus faible que dans le cas d'un système réparti où les composants sont reliés par un réseau local mais celle-ci reste tout de même important, en particulier pour les mémoires caches. Ainsi, le rapport entre un accès à un cache local et un accès à un cache dans un autre nœud peut atteindre un facteur supérieur à 100 dans le cas d'un cache de premier niveau.

Pour certaines applications, le coût associé à une vue unifiée de la mémoire est trop important pour être acceptable. Celles-ci doivent alors tenir compte de l'aspect réparti du système afin que chaque composant qui les compose soit aussi autonome que possible et ne fasse ainsi uniquement usage du module mémoire du nœud sur lequel il s'exécute. Cette approche revient à concevoir ces applications sous forme d'applications réparties, où chaque composant communique avec les autres par passage de messages. Une modification si profonde d'un programme nécessite un investissement non négligeable mais le gain en performances qui en découle en justifie l'intérêt pour certaines classes de programmes. C'est le cas notamment des noyaux de systèmes d'exploitations ; les performances de ceux-ci impactent tous les programmes qui s'exécutent sur le système. C'est ainsi que plusieurs travaux [SVS94, WA09, BBD+09] s'intéressent aux noyaux avec pour but de concevoir ceux-ci sous la forme de noyaux répartis.

Modifier une application pour qu'elle devienne une application répartie permet de réduire de façon efficace le nombre d'accès à des mémoires faisant partie de nœud distant. Cependant, une application répartie peut nécessiter une grande communication afin de fonctionner, celle-ci pouvant alors devenir un facteur clé dans les performances obtenues. Il est donc essentiel que la communication inter-cœur soit la plus rapide possible, les performances de nombreuses applications étant directement liées aux performances de cette communication.

La communication inter-cœurs est un exemple du cas plus général de transmission de données entre un producteur et un consommateur. Ce cas général a déjà été largement traité par la communauté scientifique. Cependant, le contexte du multi-cœurs a ses propres contraintes – notamment le système de maintien de la cohérence des caches – mais aussi ses propres assouplissements. Ce chapitre a pour but d'étudier en profondeur cette famille d'algorithme. La première section présente un état de l'art des différents algorithmes pouvant être utilisé pour communiquer entre deux cœurs. La deuxième section présente un nouvel algorithme de communication inter-cœur nommé BatchQueue offrant de meilleures performance que les algorithmes existants en prenant mieux en compte les spécificités des architectures multi-cœurs. Enfin, la troisième section présente une évaluation de l'algorithme de communication BatchQueue par rapport aux solutions existantes.

3.1 État de l'art des algorithmes de communication inter-cœurs

La première partie de cette section est dédiée aux primitives des systèmes d'exploitation, telles celles permettant la communication par le réseau et celles pour la communication

inter-processus. S'ensuivent une présentation de l'algorithme classique du domaine, à savoir la file sans verrou à accès simultané de Lamport ainsi que les algorithmes dérivés de celui-ci. Enfin, la section se termine par une présentation des algorithmes écrits et optimisés spécifiquement pour le contexte multi-cœurs.

3.1.1 Primitives de communication des systèmes d'exploitation

Les systèmes d'exploitation modernes offrent de multiples mécanismes de communications. Sur des systèmes multi-cœurs, ceux-ci peuvent être utilisés pour effectuer une communication entre deux cœurs. Parmi les mécanismes de communication offerts, on peut citer les primitives de communication réseau, les primitives de communication inter-processus et les primitives offrant la possibilité de partager de la mémoire entre plusieurs processus. Bien que faisant partie des primitives de communication inter-processus en pratique, ces dernières primitives sont traitées de façon séparée en raison du fait qu'elles ne proposent pas de mécanisme de communication au sens strict.

3.1.1.1 Communication par le réseau

Les mécanismes de communication réseau offerts par les systèmes d'exploitation ont pour objet de permettre la communication avec des systèmes distants. De tels mécanismes peuvent cependant être utilisés dans le but de fournir une communication inter-cœurs. Par exemple, la communication par le protocole IP peut-être utilisée pour communiquer entre deux processus locaux s'exécutant sur deux cœurs distincts en utilisant l'hôte spécial *localhost* ou les adresses IP spéciales correspondantes 127.0.0.1/8 et : :1/128.

Utiliser un mécanisme de communication réseau offre la possibilité de répartir les processus concernés sur des machines distinctes de façon transparente, sans avoir à changer le code. Cet avantage a un prix puisqu'une communication réseau nécessite une encapsulation des données à envoyer afin d'y adjoindre des informations de routage telles que l'adresse de l'émetteur et du destinataire.

Par ailleurs, les primitives permettant d'utiliser les mécanismes de communication réseau sont fournies par le noyau des systèmes d'exploitation. Dès lors, leur utilisation requiert d'effectuer des passages de mode utilisateur à mode noyau et réciproquement. Un tel changement de mode est coûteux : il faut sauvegarder le contexte d'exécution courant et charger celui à exécuter pour un changement de mode. Le nombre de cycles nécessaires pour exécuter deux changements de contexte est de l'ordre de mille instructions [HAF+07].

Des solutions de plus haut niveau reposent sur ces mécanismes réseaux et en héritent donc le même surcoût en performance. On peut citer notamment les appels de procédure à distance ou le mécanisme « mailslot » sous Microsoft Windows.

3.1.1.2 Communication Inter-Processus

Les systèmes d'exploitation multi-tâches fournissent également un ensemble de primitives permettant aux différentes tâches s'exécutant sur le système de collaborer. Parmi ces primitives se trouvent des primitives de synchronisation mais également des primitives de communication, généralement référées en tant qu'IPC, pour Inter-Process Communication – Communication Inter-Processus. Les systèmes de la famille Unix proposent ainsi [Pro12] les tubes nommés et anonymes ainsi que les files de messages et sockets unix. Les autres

systèmes d'exploitation proposent des primitives équivalentes. Par exemple, les systèmes Microsoft Windows proposent également les tubes nommés et anonymes [Net12].

Tout comme les mécanismes de communication réseau, les primitives de communication inter-processus sont fournies par les noyaux des systèmes d'exploitation. Ils souffrent donc du même surcoût en communication dû aux changements de mode. Ils sont, en revanche, exempts du problème d'encapsulation des données envoyées.

De nombreuses autres solutions proposent des mécanismes de communication de plus haut niveau en se basant sur les primitives précédemment citées. Parmi ceux-ci on peut citer D-bus et le système d'atome dans le serveur X pour Unix et COM, DDE et le système de copie de données pour Microsoft Windows. Puisque ces mécanismes reposent sur les primitives de communication du noyau, ils en héritent les coûts dûs au changement de modes et ajoutent un surcoût lié à la structuration des données. Ces mécanismes ne sont d'ailleurs généralement pas utilisés pour la multi-programmation mais plutôt pour permettre une collaboration entre applications.

Mémoire partagée Le mécanisme de mémoire partagée, bien que faisant partie des mécanismes de communication inter-processus fournis par les systèmes d'exploitation, ne permet pas directement de communiquer entre deux processus. Il permet, en revanche, de fournir à plusieurs processus un espace commun où ceux-ci peuvent partager des données grâce au système de cohérence des caches présenté dans le chapitre 2. Le système de cohérence des caches n'offrant qu'une cohérence faible, ce mécanisme est généralement combiné à des primitives de synchronisation permettant de créer des sections critiques tels les sémaphores ou les verrous d'exclusion mutuelle. Ces primitives ont l'avantage supplémentaire d'éviter d'effectuer des attentes actives puisqu'elles mettent les tâches en attente d'une ressource en pause jusqu'à ce que la ressource demandée soit disponible.

La mémoire partagée n'induit pas de changement de mode et permet donc une communication efficace. C'est sur la base de ce mécanisme, exception faite des processus léger qui partagent déjà leur espace d'adressage, que sont construits tous les algorithmes présentés dans les sections suivantes.

3.1.2 File sans verrou à accès simultané de Lamport

L'utilisation de primitive de synchronisation pour protéger l'accès à un segment de mémoire partagée offre des performances accrues par rapport à la communication par le réseau ou encore la communication inter-processus fournie par les systèmes d'exploitation. Pour autant, le coût des primitives de synchronisation reste important. C'est pourquoi des solutions sans verrou ont été mises au point.

Lamport est le premier à avoir proposé un algorithme de communication sans verrou [Lam83]. Il s'est placé pour cela dans le cadre restreint d'une file avec producteur et consommateur uniques. L'algorithme nécessite trois variables partagées :

— un tampon tab de taille N dans lequel les données sont transmises ;

— un compteur cpt_{prod} d'éléments produits ;

— un compteur cpt_{cons} d'éléments consommés.

Aucun indice n'est nécessaire pour représenter la progression du producteur et du consommateur car celle-ci est représentée par les mêmes compteurs de production et consommation,

en considérant leur valeur modulo la taille du tampon. Afin d'éviter la concurrence d'accès sur les variables, chaque variable n'est accédée en écriture que par un seul un écrivain. Le producteur a accès en écriture sur le tampon et le compteur d'éléments produits tandis que le consommateur a accès en écriture sur le compteur d'éléments consommés.

L'algorithme repose alors sur le maintien d'un invariant :

La différence entre le compteur d'éléments produits cpt_{prod} et le compteur d'éléments consommés cpt_{cons} doit toujours être bornée entre 0 et N.

Cet invariant signifie qu'avant de produire, le producteur doit s'assurer que la différence entre les deux compteurs est strictement inférieure à N, assurant ainsi qu'il y a une entrée vide dans le tableau. De même, le consommateur doit s'assurer que cette différence est strictement supérieure à 0, c'est à dire qu'il reste une valeur à lire. Maintenir cet invariant impose l'utilisation d'une attente active tant que celui-ci n'est pas respecté. En effet, mettre en pause et reprendre un fil d'exécution nécessite un support du système d'exploitation et a donc un coût prohibitif. L'utilisation d'une attente active est donc adéquat car elle repose sur l'hypothèse que l'attente est de courte durée. L'algorithme proposé par Lamport est présenté dans les fonctions produit_lamport() et consomme_lamport() dans les figure 3 et 4 ci-dessous. La primitive *Attendre* dans l'algorithme se réfère donc à une attente active.

Attendre cpt_{prod} - cpt_{cons} != N ;
tab[cpt_{prod} mod N] $\longleftarrow$ $data$;
cpt_{prod} $\longleftarrow$ cpt_{prod}+1 ;

FIGURE 3.1 – produit_lamport( )

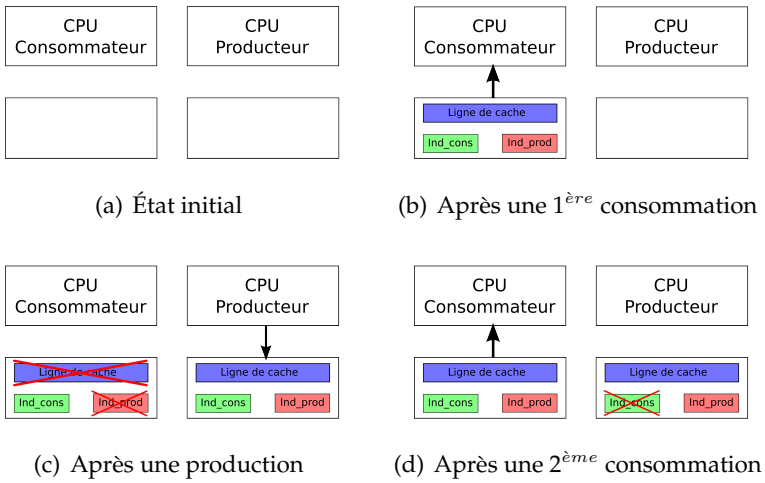
Attendre cpt_{prod} - cpt_{cons} != 0 ;
$data$ $\longleftarrow$ tab[cpt_{cons} mod N] ;
cpt_{cons} $\longleftarrow$ cpt_{cons}+1 ;
Result : data

FIGURE 3.2 – consomme_lamport( )

Les figures 3.3(a) à 3.3(d) présentent l'exécution de cet algorithme dans une architecture multi-cœurs. Seuls les caches de premier niveau sont présentés car seul ce niveau de cache est impliqué en cas de communication importante et c'est précisément dans ce contexte que la communication doit être efficace.

La figure 3.3(a) présente la situation à l'origine où le tampon partagé entre le producteur et le consommateur et leurs indices sont absents des caches de premier niveau. Pour lire une donnée dans le tampon, le consommateur doit d'abord comparer son indice à celui du producteur afin de savoir si une donnée est disponible. Si tel est le cas, il effectue la lecture correspondante (figure 3.3(b)). Les valeurs courantes des deux indices ainsi que du tampon sont alors chargées dans le cache. La récupération de ces valeurs depuis le cache de niveau supérieur est coûteuse car celui-ci est plus lent. Une fois la donnée consommée, le consommateur incrémente son indice de consommation.

Le producteur doit alors récupérer la valeur courante des deux indices dans son cache (figure 3.3(c)) pour vérifier qu'il a la place d'écrire une nouvelle donnée. Lorsque, le pro-



(a) État initial (b) Après une $1^{ère}$ consommation

(c) Après une production (d) Après une $2^{ème}$ consommation

FIGURE 3.3 – Fonctionnement de l'algorithme producteur / consommateur classique

ducteur écrit la donnée dans le tampon et incrémente son indice de production, ces données sont invalidées dans le cache du consommateur. Or, comme il a été décrit dans le chapitre 2, cette invalidation coûte également cher car elle doit être envoyée à tous les cœurs, qu'ils possèdent ou non la donnée.

Toutes les valeurs invalidées devront être rechargées depuis le cœur du producteur lorsque le consommateur effectuera à nouveau une lecture (figure 3.3(d)). Contrairement à la figure 3.3(a) où les caches étaient vides, cette consommation s'accompagnera d'une invalidation de l'indice du consommateur lorsque celui-ci est incrémenté.

Il ressort de cet exemple d'exécution que la file sans verrou à accès simultané de Lamport est susceptible de générer de nombreuses invalidations et lecture de données depuis un cache distant lorsque le producteur et le consommateur fonctionnent en parallèle. L'algorithme peut en effet générer jusqu'à trois invalidations et trois lectures distantes par donnée envoyée dans le pire cas.

La file sans verrou de Lamport présente néanmoins une efficacité supérieure par rapport aux solutions utilisant les primitives de synchronisation fournies par les systèmes d'exploitation. Le surcoût conséquent dû au maintien de la cohérence des trois variables partagées indique cependant qu'une amélioration de l'efficacité de la communication reste possible.

Files avec producteurs et consommateurs multiples À partir des travaux de Lamport sur les files sans verrous [Lam83], de nombreuses solutions ont été proposées pour étendre l'absence d'utilisation de verrous aux files avec plusieurs producteurs et consommateurs, que ce soit en configuration statique [TZ01, Val94], ou dynamique [Val94, GLR83, LMS04, HSS07, MC87, PLJ91, PLJ94, MS98, MNSS05]. Ces travaux améliorent la communication inter-cœurs par rapport à la file de Lamport en permettant à des schémas de communication plus compliqués de bénéficier du gain en performance des algorithmes de communication sans verrou sur des architectures multi-cœurs. Les configurations statiques apportent une meilleure localité des données et un surcoût mémoire moindre tandis que les configurations dynamiques, basées sur une liste chaînée, fournissent un environnement plus flexible.

La contrepartie de ces solutions est que les algorithmes sont plus complexes et nécessitent l'utilisation d'opérations atomiques telles des compare-and-swap (comparaison et échange). Ceci a pour conséquence un surcoût accru par rapport à la file sans verrou de Lamport. De plus, tout comme celle-ci, ces algorithmes ne prennent pas de précautions particulières vis à vis du système de cohérence des caches.

3.1.3 Solutions optimisées pour les architectures multi-cœurs

Dans la file sans verrou de Lamport, une partie importante du temps passé pour envoyer une donnée d'un cœur à un autre concerne le maintien de la cohérence entre les caches. En conséquence, de nombreuses solutions [WKWY07, LBC10, GMV08, ZOYB09] se proposent d'améliorer l'efficacité de la communication en réduisant le partage de variables. Cette réduction du partage peut être atteinte par deux approches complémentaires :

— réduction du nombre de variables partagées ;

— réduction de la fréquence de modification des variables partagées.

3.1.3.1 Barrelfish

Une de ces solutions est proposée au sein du système Barrelfish [BBD+09]. Barrelfish est un noyau de système d'exploitation suivant le modèle *multikernel*, dont l'approche est de considérer un système multi-cœurs comme un système réparti. Le noyau est alors constitué d'une instance par cœur, chacune possédant une partie de l'état du système. Les cœurs de calcul ne possédant pas l'état complet du système, ils sont amenés à communiquer très fréquemment entre eux afin d'offrir les services que propose le noyau de système d'exploitation. Les contraintes de performance qu'un noyau se doit de respecter imposent donc au noyau Barrelfish d'avoir une communication très efficace.

À cette fin, les auteurs de Barrelfish proposent un algorithme de communication qui consiste à fixer la taille des messages à la taille qu'occupe une ligne de cache : une ligne de cache ne contient ainsi qu'un seul message. De cette façon, si le consommateur commence à consommer un message et qu'en nouveau message est envoyé, cela n'invalide pas la ligne de cache que le consommateur est en train de lire.

Ce système permet donc d'améliorer la latence liée à la transmission d'un message. Le producteur n'a pas à invalider la ligne de cache plusieurs fois parce que le consommateur est en train de consommer un autre message et le consommateur ne doit pas récupérer plusieurs fois une ligne de cache parce que le producteur est en train d'envoyer un nouveau message. En revanche, le débit n'est pas amélioré significativement. En effet, chaque message réside dans une ligne de cache différente, il y a donc une lecture depuis le cache du producteur pour chaque message consommé. De même, pour chaque message produit, le producteur doit invalider la ligne de cache dans laquelle il réside. En effet, la ligne de cache a été récupérée par le consommateur lors de la consommation du message qui se trouvait précédemment dans cette même ligne de cache. Il y a donc au total une invalidation et une lecture depuis un cache distant pour chaque message transmis. L'algorithme de communication utilisé dans Barrelfish réduit la concurrence sur les lignes de cache mais nécessite un plus grand nombre d'échanges de lignes de cache, d'où l'absence d'amélioration du débit.

3.1.3.2 FastForward

À l'opposé de l'algorithme de communication proposé dans Barrelfish, de nombreux algorithmes ont pour objectif d'améliorer le débit de la communication au prix d'un compromis sur la latence si nécessaire. FastForward [GMV08] est l'un de ces algorithmes.

La solution adoptée dans FastForward pour réduire l'impact du maintien de la cohérence des caches consiste à réserver une valeur du même type que les données transmises que celles-ci ne prendront jamais afin d'indiquer qu'une entrée du tampon est vide. Par exemple si les données envoyées sont des pointeurs vers des structures ou objets, la valeur NULL est réservée pour indique qu'une entrée ne contient aucune donnée. Le producteur teste alors si le tampon est plein en vérifiant que la prochaine entrée du tampon contient cette valeur réservée. De façon symétrique, le consommateur teste si le tampon est vide en vérifiant que la prochaine entrée ne contient pas cette valeur réservée. Ainsi, il n'est plus nécessaire au producteur et au consommateur de partager des compteurs d'éléments produits et consommés, seuls des indices propre à chacun sont nécessaires.

Cette élégante approche est malgré tout toujours sujette à des invalidations de lignes de cache. Cela s'explique par le faux partage qui survient lorsque le producteur et le consommateurs travaillent sur des entrées du tampon se trouvant dans la même ligne de cache. La ligne de cache correspondante est alors manipulée par deux cœurs bien que le producteur et le consommateur ne travaillent pas sur la même donnée.

Pour résoudre ce problème, les auteurs de ce travail proposent de temporiser la consommation par rapport à la production, quitte à augmenter la latence entre la production et la consommation d'une donnée. En d'autres termes, il est suggéré par les auteurs d'empêcher le producteur et le consommateur de travailler sur la même ligne de cache. Une façon d'y parvenir est de maintenir à tout instant la distance entre l'entrée du tampon dans laquelle le producteur va produire et l'entrée du tampon dans laquelle le consommateur va consommer supérieure ou égale à la taille d'une ligne de cache.

Dans leur article, les auteurs suggèrent pour la distance une borne inférieure de 2 fois la taille d'une ligne de cache. De cette façon, le producteur et le consommateur agissent toujours sur des lignes de cache distinctes. Cette temporisation peut être maintenue en contrôlant la distance à intervalle régulier : une attente active est alors effectuée si celle-ci est inférieure à la valeur minimale admise, jusqu'à ce que la distance dépasse un seuil donné. Le seuil proposé dans l'article est de 6 fois la taille d'une ligne de cache.

FastForward ne fonctionne donc efficacement qu'avec un producteur et un consommateur travaillant à des vitesses proches et relativement stables. En effet, l'attente active proposée pour maintenir la distance entre le producteur et le consommateur nécessite de surveiller les indices, en permanente évolution, du producteur et du consommateur. Cette surveillance cause ainsi des invalidations de lignes de cache que cherche précisément à éviter l'algorithme FastForward.

3.1.3.3 Algorithme utilisé dans l'extension de calcul par flux d'OpenMP

L'approche utilisé par FastForward pour résoudre le problème de synchronisation entre le producteur et le consommateur est assez atypique. Les autres algorithmes essayant de résoudre le même problème conservent une structure plus proche de l'algorithme de Lamport, avec des améliorations pour réduire l'effet du maintien de la cohérence entre les caches.

L'algorithme de communication utilisé dans l'extension de calcul par flux d'OpenMP [PC11],

ci-après appelé algorithme GOMP [1], est de cette catégorie. L'algorithme GOMP fait face à la difficulté supplémentaire d'être une file à multiples producteurs et consommateurs, ce qui rend la gestion du maintien de la cohérence d'autant plus nécessaire et difficile. En effet, la concurrence d'accès aux variables partagées se produit également entre les producteurs et entre les consommateurs.

Le principe général employé dans l'algorithme GOMP pour éviter la concurrence est de conserver une copie locale des variables partagées et d'utiliser ces copies locales autant que possible. Pour des raisons de performances, les producteurs et consommateurs ne sont pas reliés directement au tampon de communication. À la place, les producteurs et consommateurs sont connectés à des structures représentant l'ensemble des producteurs et l'ensemble des consommateurs. Cet arrangement des structures permet d'avoir différents niveaux de copies locales, de la même manière que les processeurs ont différents niveaux de mémoire cache. En pratique, cela signifie que le contenu des variables pour lesquelles beaucoup de contention existe est copié localement à la fois dans la structure qui représente l'ensemble des producteurs (respectivement consommateurs) et à la fois dans les structures représentant les producteurs (respectivement consommateurs) eux-mêmes.

Néanmoins, la modification simultanée de ces variables partagées est susceptible de créer une situation de faux partage si celles-ci résident dans la même ligne de cache. Pour éviter ce faux partage, les structures utilisées par l'algorithme GOMP sont disposées telles que les variables partagées ont leur propre ligne de cache. Cependant, comme vu précédemment pour FastForward, le faux partage peut se produire sur le tampon de communication lui-même si des participants, producteurs ou consommateurs, travaillent sur des données différentes reposant dans la même ligne de cache. Ce scénario ne se produit pas pour l'algorithme GOMP en pratique car les données sont produites et consommées par lot dont la taille est 32 fois la taille d'une donnée. Comme les éléments envoyés sont généralement plus grand qu'un octet, les producteurs et consommateurs agissent donc en permanence sur des lignes de cache différentes.

Il ressort de l'analyse ci-dessus que l'algorithme GOMP est conçu pour gérer la communication inter-cœur efficacement. L'algorithme parvient à réduire la contention sur les variables partagées ainsi que le faux partage, à la fois sur les variables et sur le tampon de communication. Cependant la nature même de l'algorithme GOMP en tant que file à multiples producteurs et consommateurs en limite son efficacité. La restriction à un unique producteur et consommateur permet en effet plus de simplicité et des optimisations supplémentaire quant au système de maintien de la cohérence des caches.

3.1.3.4 DBLS and MCRingBuffer

Le problème du faux partage lorsque le producteur et le consommateur opère sur la même ligne de cache est inhérent aux files de communication sur des architectures multi-cœurs. Forcer la taille des données à la taille d'une ligne de cache comme cela est fait dans Barrelfish ne résout pas le problème : il y a nécessairement pour chaque ligne de cache, et donc pour chaque donnée envoyée, une lecture depuis un cache distant et une invalidation, comme vu précédemment. Il n'y a donc pas d'autres alternatives que de maintenir dans le tampon de communication une distance suffisante entre le producteur et le consommateur

1. Il n'y a qu'un seul algorithme de communication dans l'extension de calcul par flux pour OpenMP. Aussi, son nom provient de l'implémentation OpenMP dans laquelle il a été intégré, à savoir l'implémentation GNU OpenMP, communément appelée GOMP.

ainsi que proposé par les auteurs de FastForward. Il est alors essentiel pour un algorithme de communication inter-cœurs d'être capable de maintenir cette distance suffisante sans générer de nombreuses invalidations et lecture depuis un cache distant. L'algorithme GOMP y parvient lorsque les données envoyées sont plus grandes qu'un octet mais son efficacité est limité par sa nature de file à multiples producteurs et consommateurs.

À cet effet, DBLS [WKWY07] et MCRingBuffer [LBC10] proposent une communication à un seul producteur et consommateur dont l'envoi des données est retardé jusqu'à ce que N lignes de cache soient remplies, N étant plus grand que 1. Cet envoi retardé des données permet de remplir une ligne de cache en ne l'invalidant qu'une seule fois, ce qui est le cas optimal pour une ligne de cache partagée.

Reprenant le principe utilisé dans l'algorithme GOMP, les invalidations dues au changement de valeur des indices de consommation et production sont réduites en utilisant, pour chaque indice, une variable partagée et deux variables locales. Le producteur et le consommateur possèdent ainsi deux variables locales : une « copie locale » de leur propre indice – l'indice mis à jour après chaque production ou consommation – et une « copie miroir » de l'indice partagé indiquant la position de l'autre participant.

Les mises à jour s'effectuent :

— chaque fois qu'une donnée est produite (respectivement consommée) pour la copie locale ;

— toutes les N lignes remplies (resp. vidées) pour la copie miroir et la variable partagée.

Dans le cas de MCRingBuffer, les copies miroir sont mises à jour depuis les variables partagées uniquement lorsqu'aucune progression ne peut être faite à partir des copies miroir. De plus, les auteurs de MCRingBuffer explicitent le fait que les variables du producteur et du consommateur doivent être dans des lignes de cache séparées.

Par l'utilisation des copies locales et miroir, DBLS et MCRingBuffer parviennent à réduire drastiquement l'impact du système de cohérence des caches et ainsi augmenter le débit de communication. Ce faisant, de nombreuses variables sont introduites, augmentant d'autant l'empreinte mémoire d'un canal de communication et la complexité de l'algorithme. De plus, bien que la fréquence de modification des variables partagée soit diminuée, leur nombre reste quant à lui inchangé.

3.1.3.5 File logicielle par lot

La « file logicielle par lot [ZOYB09] » ou CSQ (pour Clustered Software Queue) propose une alternative intéressante à DBLS et MCRingBuffer en permettant la synchronisation entre le producteur et le consommateur sans nécessiter d'indices partagés de production et consommation. Comme ces deux solutions, CSQ retarde l'envoi des données pour minimiser les invalidations de lignes de cache. Cependant, CSQ remplace les variables partagées du producteur et du consommateur par un ensemble de bits, un par lot de données transmises en même temps.

L'idée utilisée par CSQ est que le producteur et le consommateur n'ont pas besoin de connaître leur position respective. Ce dont ils ont besoin en revanche est de savoir si l'entrée du tampon dans laquelle ils s'apprêtent à produire ou consommer est vide ou non. Il est donc suffisant d'avoir un bit de statut par entrée du tampon indiquant si l'entrée correspondante est vide ou pleine pour permettre la synchronisation entre le producteur et le consommateur.

Pour autant, remplacer les variables partagées de production et consommation par un bit par entrée du tampon serait contre-productif. En effet, bien que DBLS et MCRingBuffer utilisent des variables de production et consommation partagées, celles-ci ne sont mises à jour qu'occasionnellement. CSQ opère également un traitement des données par lot. Pour cela il utilise un ensemble de tampons au lieu d'un unique tampon, et les données sont produites et consommées par tampon entier. Il est donc possible de réduire le nombre de bit à un par tampon. La sémantique des bits de statut s'applique alors au tampon entier : ils indiquent si le prochain tampon à traiter est rempli ou vide.

Ces caractéristiques permettent à l'algorithme CSQ de réduire de façon importante le surcoût lié au maintien de la cohérence des caches. Néanmoins, CSQ sollicite le maintien de la cohérence en plusieurs occasions, dont certaines pourraient être évitées. En effet, pour chaque tampon de données consommé, le consommateur doit récupérer depuis le cache du cœur exécutant le producteur la ligne de cache qui contient le bit de statut. De façon symétrique, pour chaque tampon de données produites, le producteur doit invalider la ligne de cache qui contient le bit de statut associé. Conscients de ce problème, les auteurs ont étudié l'influence de la taille du tampon sur le débit et proposent de grouper les données à envoyer dans des lots de taille égale à 64 lignes de cache. Ce faisant, l'ensemble des tampons de communication occupe une place conséquente : 2048 lignes de cache avec les paramètres que suggère l'algorithme. Cette quantité représente plus que la taille du cache de premier niveau, cela signifie donc que l'ensemble des tampons ne tient pas entièrement dans celui-ci.

Un autre problème que présente CSQ est la consommation mémoire lié à l'utilisation des bits de statut. Pour maintenir de bonnes performances, les bits de statut doivent être chacun dans des lignes de cache différentes. Sans cette précaution, l'algorithme CSQ rencontrerait un nombre important de faux partage. Or, l'article présentant CSQ recommande [2] l'utilisation de 64 tampons, l'idée étant d'offrir plus de souplesse au niveau de la synchronisation entre le producteur et le consommateur. Cela signifie donc l'utilisation de 64 lignes de caches en plus de celles nécessaires pour les tampons eux-mêmes. En considérant la taille des tampons recommandée dans l'article – 64 lignes de cache – le surcoût est faible. Mais si la taille des tampons est réduite pour réduire la taille de l'ensemble des tampons à une petite fraction du cache de premier niveau, alors le surcoût devient conséquent.

CSQ offre donc une communication efficace mais certains points peuvent encore être améliorés. Tout comme DBLS et MCRingBuffer, CSQ parvient à réduire de façon importante le nombre d'invalidations et de lecture depuis un cache distant. CSQ présente également une synchronisation entre le producteur et le consommateur différente de ces autres solutions, qui ne nécessite qu'un bit par tampon. Par contre, CSQ impose une synchronisation par tampon produit même si plusieurs lignes de cache pourraient être produites et consommées en même temps. De plus, CSQ a une empreinte mémoire non négligeable à cause des bits de synchronisation.

Il ressort de cet état de l'art que les solutions pour communiquer d'un cœur à un autre sont nombreuses. Les solutions existantes couvrent de nombreux cas d'utilisation, de la coopération entre processus via une communication occasionnelle à la répartition d'un calcul sur plusieurs cœurs via la technique de parallélisme de flux permise par une communica-

2. L'article étudie également l'influence d'un nombre variable de tampons mais jamais avec moins de 32 tampons.

tion intense. Or, bien que des algorithmes s'illustrent dans ce dernier domaine, tel CSQ ou MCRingBuffer, des progrès restent possibles qui peuvent améliorer de façon notable l'efficacité de toutes les applications ayant recours au parallélisme.

3.2 Communication inter-cœurs rapide avec BatchQueue

Comme le détaille la section précédente, de nombreux algorithmes de communication inter-cœurs existent déjà, dont un nombre important cherche à offrir de bonnes performances. Comme il a été noté, des améliorations sont cependant toujours possibles avec à la clé un nombre plus grand de programmes parallélisables et une accélération plus grande pour ceux déjà parallélisables. Par exemple, les algorithmes tendent à augmenter le nombre de variables à mettre à jour et à être sensiblement plus compliqués que la file sans verrou de Lamport, ce qui augmente le nombre moyen d'instructions à exécuter par donnée envoyée. Le nombre d'alternatives effectuées tends également à augmenter, ce qui réduit l'efficacité du flot d'instructions exécuté.

Cette section présente un nouvel algorithme de communication appelé BatchQueue qui parvient à réduire le nombre de variables utilisés, en particulier le nombre de variables partagées, et réduire le coût de la synchronisation. La première partie de cette section présente le principe de fonctionnement de BatchQueue et fait le parallèle avec les solutions existantes. La partie suivante présente l'algorithme BatchQueue en détail et montre comment BatchQueue règle chacun des points listés dans la première partie. Enfin, la dernière partie décrit une optimisation de BatchQueue impliquant un contournement du système de cohérence des caches afin de mieux parer aux effets secondaires indésirables du préchargement des lignes de cache par le processeur.

3.2.1 Principe

Parmi les nombreux algorithmes de communication qui ont été proposés ces dernières années afin d'améliorer l'efficacité de la communication inter-cœurs, CSQ est la solution la plus aboutie. Celui-ci représente un progrès substantiel par rapport à la file sans verrou de Lamport mais des améliorations restent cependant possibles. BatchQueue est un effort pour remédier aux limites existantes et ainsi augmenter encore l'efficacité de la communication inter-cœurs. À cette fin, BatchQueue reprend certains des principes qui font de CSQ une solution efficace – file à producteur et consommateur unique, envoi des données par lot et algorithme minimal – et en ajoute de nouveaux.

File à producteur et consommateur unique Comme tous les algorithmes de communication récents, BatchQueue est une file avec un unique producteur et consommateur. En effet, la gestion de multiples producteurs et/ou de multiples consommateurs rend la synchronisation plus complexe et plus coûteuse. Or, de nombreuses applications ne nécessitent pas de tels mécanismes et peuvent se contenter d'une communication à un producteur et un consommateur. De plus, la limitation à deux entités, producteur et consommateur, permet plusieurs optimisations : les verrous et instructions compare-et-échange (« compare-and-swap ») deviennent inutiles et la synchronisation avec une variable booléenne est alors possible. Le choix de ne supporter qu'un seul producteur et un seul consommateur permet donc d'améliorer considérablement l'efficacité de la communication.

Envoi des données par lot L'envoi des données par lot est la technique qui consiste à retarder l'envoi des données jusqu'à ce que suffisamment de données soient prêtes à être envoyées. Cette technique participe à réduire l'effet du maintien de la cohérence des caches de deux façon différentes :

— réduction du nombre de synchronisation producteur / consommateur ;

— suppression du faux partage sur les données.

Ces effets ont tous les deux un impact très important sur l'efficacité de la communication. En ce qui concerne la synchronisation producteur / consommateur, le coût provient de l'échange qui est nécessaire entre les deux participants. Pour effectuer cet échange, chaque participant effectue deux actions :

— Consultation du statut de l'autre participant en consultant une variable partagée ;

— Notification de l'autre participant en modifiant une variable partagée.

Autrement dit, lors de chaque synchronisation producteur / consommateur deux lectures sont effectuées depuis un cache distant ainsi que deux invalidations. S'agissant d'un échange, il y a une dépendance entre ces opérations qui sont donc effectuées l'une après l'autre. Réduire le nombre de synchronisations réduit donc à la fois le nombre de lectures distantes et le nombre d'invalidations.

La suppression du faux partage a également un impact important puisque celui-ci peut se produire pour chaque donnée transmise. En effet, si le producteur et le consommateur travaillent en parallèle, chaque production génère des invalidations et chaque consommation qui s'ensuit doit récupérer le nouveau contenu de la ligne de cache depuis le cache du producteur. Retarder l'envoi des données permet de cantonner le producteur et le consommateur à des parties du tampon de communication qui résident dans des lignes de cache différentes.

Algorithme minimal Un autre élément important que partagent CSQ et BatchQueue est le fait d'avoir un algorithme minimal, à comparer aux nombreux tests de MCRingBuffer par exemple. Cette minimalité du code est particulièrement importante dans la section des algorithmes, appelée chemin rapide (« fastpath »), qui est exécutée pour chaque donnée envoyée. Dans le cas de CSQ et BatchQueue le chemin rapide n'est constitué que de deux actions : écrire la donnée dans le tampon et tester si la fin du tampon est atteinte. Bien que moins important, le nombre d'actions effectuées lors de l'envoi réel des données peut également avoir une influence non négligeable lorsque l'envoi des données n'est pas beaucoup retardé. Dans ce cas également BatchQueue et CSQ ne comportent que très peu d'actions. La moyenne du nombre d'instructions exécutées par donnée envoyée est donc plus faible dans BatchQueue et CSQ que dans les autres algorithmes.

Nombre minimal de tampons BatchQueue veille également à remédier aux faiblesses que possèdent les algorithmes existants, en particulier CSQ. À cette fin, deux améliorations sont proposées dans BatchQueue. La première est que, contrairement à CSQ, BatchQueue fait le choix de n'utiliser que deux tampons. Ceci permet, à taille globale constante, d'avoir des tampons de plus grande capacité et ainsi limiter le nombre de synchronisations producteur / consommateur. A contrario, pour une taille des tampons fixée, cela permet de limiter la taille

globale de l'ensemble des tampons. De cette manière, l'ensemble de la structure de communication peut tenir dans le cache de premier niveau, ce qui n'est pas le cas pour CSQ. Or le fait que l'ensemble des données accédées par un algorithme tient dans le cache de premier niveau a une influence considérable [Fur03, Dre07] sur les performances de l'algorithme en question en raison de la différence de latence d'accès. Il est ainsi essentiel qu'un algorithme de communication nécessite aussi peu de mémoire que possible.

Unique variable de synchronisation La seconde amélioration que propose BatchQueue est l'utilisation d'une unique variable de synchronisation là où CSQ en a besoin d'une par tampon de communication. Ceci est rendu possible par le respect d'un invariant – décrit dans la section suivante – concernant la façon de mettre à jour cette variable. L'utilisation d'une unique variable de synchronisation pour deux tampons de communication améliore l'efficacité de l'algorithme de deux façons.

Tout d'abord, les variables de synchronisation sont systématiquement mises dans des lignes de cache distinctes afin d'éviter le faux partage. En conséquence, l'utilisation d'une unique variable de synchronisation ne nécessite qu'une ligne de cache supplémentaire au lieu de deux lignes de cache comme c'est le cas dans CSQ. Cette différence est particulièrement importante lorsque les tampons sont petits, de la taille d'une ligne de cache par exemple.

Ensuite, l'utilisation d'une unique variable a le potentiel de réduire le nombre de défauts de cache. En effet, dans le cas où deux variables sont utilisées, leur modification implique nécessairement un défaut de cache. En effet, si l'on considère la variable d'un tampon donné, celle-ci sera modifiée tour à tour par le producteur et le consommateur. Dans le cas d'une unique variable, le dernier à avoir modifié la variable peut être le premier à la modifier à nouveau.

BatchQueue propose donc une évolution des algorithmes de communication inter-cœurs récents en reprenant les principes qui rendent ces algorithmes efficaces et en en ajoutant de nouveaux. Ainsi, BatchQueue conserve un envoi retardé des données et un algorithme minimal. BatchQueue propose en outre de n'utiliser que deux tampons de communication et d'effectuer la synchronisation entre le producteur et le consommateur à l'aide d'une unique variable. Ces apports réduisent de façon importante l'empreinte mémoire de l'algorithme de communication et réduisent le coût de la synchronisation entre le producteur et le consommateur.

3.2.2 Algorithme

L'algorithme BatchQueue est présenté dans les fonctions produit_batchqueue() et consomme_batchqueu
Le principe est de diviser le tampon de communication tab en deux parties, appelées demi-tampons, dont la taille N est un multiple de celle d'une ligne de cache. Ceci permet à la production et la consommation de se produire en parallèle depuis des lignes de cache différentes, évitant ainsi le faux partage. Les demi-tampons sont lus et écrits d'une traite. Quand les demi-tampons sont complètement traités, une synchronisation a lieu pour les échanger. Après celle-ci, le consommateur lit depuis le demi-tampon que le producteur vient juste de remplir et le producteur écrit dans le tampon que le consommateur vient juste de consommer. Afin de rendre le code plus simple et efficace, la taille N d'un demi-tampon doit être une puissance de 2 de la taille d'une ligne de cache et les demi-tampons doivent être alignés. Les indices de production et de consommation sont notés respectivement ind_{prod} et ind_{cons}.

$tab[ind_{prod}] \longleftarrow donnee$;
$ind_{prod} \longleftarrow (ind_{prod}+1) \bmod 2N$;
si ind_{prod} *mod* $N = 0$ **alors**
  Attendre $statut$ = **faux** ;
  $statut \longleftarrow$ **vrai** ;
fin

FIGURE 3.4 – produit_batchqueue()

Attendre $statut$ = **vrai** ;
pour $i \longleftarrow ind_{cons}$ **a** $ind_{cons}+N-1$ **faire** $tampon\_resultat[i] \longleftarrow tab[i]$;
;
$ind_{cons} \longleftarrow (ind_{cons}+N) \bmod 2N$;
$statut \longleftarrow$ **faux** ;

FIGURE 3.5 – consomme_batchQueue()

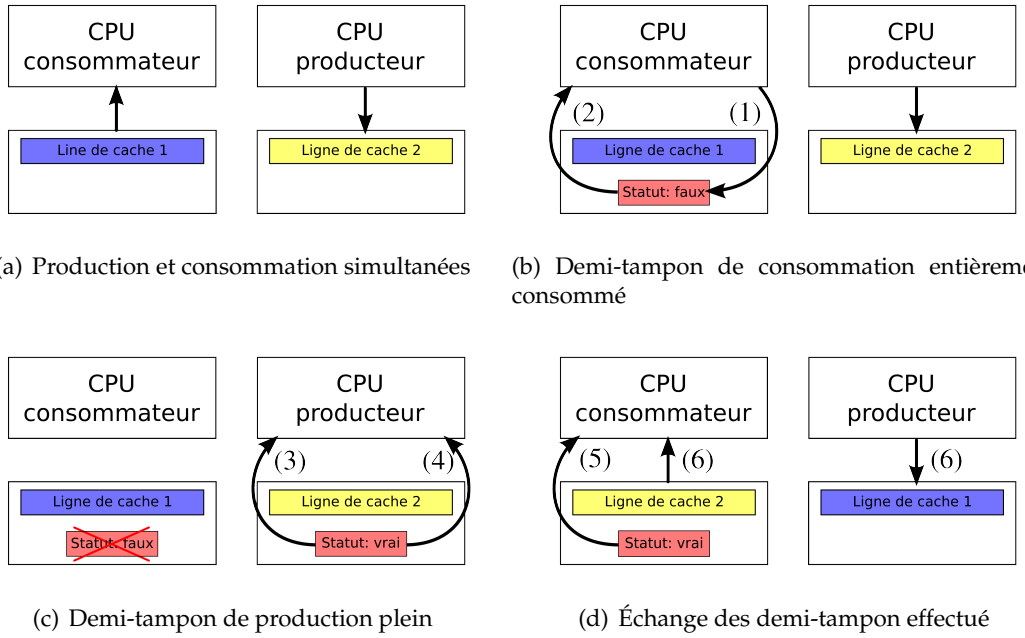
La synchronisation pour échanger les deux demi-tampons repose sur une unique variable partagée booléenne appelée $statut$. Le producteur ne peut permuter sa valeur que lorsque celle-ci est fausse tandis que le consommateur ne peut permuter sa valeur que lorsque celle-ci est vraie. Cet invariant garantit que le producteur et le consommateur ne mettent jamais à jour la variable en même temps.

Cette variable n'est pas modifiée pendant le traitement des demi-tampons. Elle est modifiée uniquement lorsqu'une synchronisation est nécessaire, c'est à dire lorsque les demi-tampons sont échangés. En d'autres mots, le partage réel de la variable ne se produit que pendant l'échange des demi-tampons. Puisque le débit de communication est lié à la quantité de synchronisation effectuée et que cette synchronisation ne se produit que lorsque les demi-tampons sont échangés, il est possible d'augmenter le débit en augmentant la taille des demi-tampons. Cependant, utiliser des demi-tampons plus grand implique que le temps nécessaire pour remplir un demi-tampon avant de l'envoyer est également plus grand, c'est à dire que la latence est plus importante. Choisir la bonne taille pour les demi-tampons relève donc d'un compromis entre débit et latence.

Comme la plupart des algorithmes de communication récents, BatchQueue rend possible la production et la consommation en parallèle sans induire de partage de donnée. La synchronisation entre producteur et consommateur ne s'effectue qu'une fois par demi-tampon. Ceci favorise le participant le plus lent : seule la variable de synchronisation est considérée quand l'un des participants finit de traiter son demi-tampon, l'autre participant n'est pas ralenti.

La figure 3.6 illustre la façon dont BatchQueue fonctionne. Il y a 4 étapes. Tout d'abord, la production et la consommation se produisent en parallèle, sans s'entraver l'un l'autre (figure 3.6(a)). Puis, quand le consommateur a fini de consommer son demi-tampon, il positionne $statut$ à **faux** et se met en attente jusqu'à ce que $statut$ soit **vrai** (figure 3.6(b)). Ensuite, lorsque le producteur finit d'écrire dans son demi-tampon car il est plein, il vérifie d'abord que $statut$ est **faux** ce qui signifie que le consommateur a fini de consommer son demi-tampon, puis positionne $statut$ à **vrai** (fig. 3.6(c)). Enfin, la synchronisation étant finie, les

demi-tampons sont échangés et le producteur et le consommateur commencent à nouveau à écrire et lire depuis leur nouveau demi-tampon (figure 3.6(d)).



(a) Production et consommation simultanées

(b) Demi-tampon de consommation entièrement consommé

(c) Demi-tampon de production plein

(d) Échange des demi-tampon effectué

FIGURE 3.6 – Fonctionnement de BatchQueue

L'algorithme de BatchQueue présenté ci-avant présente bien les caractéristiques annoncées dans la section précédente. BatchQueue permet, à l'aide d'un algorithme minimal, de transférer des données par lot avec une synchronisation reposant sur une unique variable booléenne partagée. Par ailleurs, la structure de donnée nécessaire pour la communication a une empreinte mémoire réduite : seuls deux tampons de communication et la variable de synchronisation *statut* sont nécessaires pour communiquer.

3.2.3 Gestion du préchargement

Comme vu dans le chapitre 2, les processeurs modernes contiennent tous un mécanisme de préchargement qui permet de charger des lignes de cache avant qu'elles ne soient accédées. Ce mécanisme apporte des gains considérables en performance dans le cas d'accès séquentiels en mémoire, tel le parcours d'un tableau par exemple. Son fonctionnement est le suivant :

1. détecter si plusieurs lignes de cache successives sont accédées ;
2. charger les lignes de cache suivantes au fur et à mesure des accès successifs.

La détection ne se limite pas aux lignes de cache successives, tout accès linéaire est également détecté. Cela permet ainsi de détecter l'accès, dans un tableau de structure de grande taille, à un champ particulier de ces structures.

S'il améliore les performances des applications, le préchargement peut occasionnellement avoir des effets négatifs en pré-chargeant des lignes de cache qui vont être modifiées.

Dans ce cas, la modification de la ligne de cache est rendue plus coûteuse car une invalidation doit être effectuée. Sans préchargement, la ligne serait restée dans l'état exclusif et la modification n'aurait impliqué aucun message. C'est le cas notamment avec BatchQueue lors de l'accès à la variable de synchronisation *statut*. L'accès du consommateur à son demi-tampon est susceptible de déclencher le préchargement de la ligne de cache qui contient la variable *statut* alors que celle-ci est modifiée dès que le demi-tampon est entièrement consommé. De même, lorsque le consommateur consomme le premier demi-tampon, l'unité de préchargement détecte les accès séquentiels et pré-charge le demi-tampon suivant, celui-ci étant consécutif en mémoire. Or ce demi-tampon est en train d'être modifié par le producteur.

Il est dès lors désirable de pouvoir éviter ce préchargement sans désactiver [3] celui-ci le reste du temps. La solution utilisée pour la variable *statut* est simple : il s'agit d'éloigner la variable des demi-tampons de plusieurs lignes de cache. De cette manière, le préchargement s'effectue toujours mais charge des lignes de cache vides. Celle-ci n'étant pas accédée, le préchargement ne continue pas jusqu'à la ligne de cache contenant la variable *statut*. Cette solution fonctionne mais possède deux inconvénients. Tout d'abord, cette solution est coûteuse en espace mémoire puisque l'espace inséré n'est pas utilisé. Ceci n'est pas forcément vrai dans le cas général car il est possible de réorganiser la structure partagée pour placer les données accédées uniquement en lecture à cet emplacement. En revanche, dans le cas de la structure de communication utilisée par BatchQueue, cet espace est perdu. De plus, cette solution nécessite qu'un espace puisse être introduit au milieu de la structure. Cette technique, applicable à la variable *statut*, ne peut donc pas être utilisée pour éviter le préchargement entre les deux demi-tampons puisque ceux-ci doivent être contigu en mémoire.

Une autre solution, utilisée pour réaliser le système à mémoire partagée répartie MultiView [IS98], consiste à associer plusieurs adresses virtuelles à l'adresse physique à laquelle se situe la structure de communication. En effet, pour des raisons d'efficacité la détection d'accès multiples qu'effectue le préchargement se fait sur les adresses virtuelles et non physiques. Un accès à plusieurs lignes de cache consécutives par des adresses virtuelles se trouvant dans des pages mémoires distinctes n'est ainsi pas détecté par l'unité de préchargement. Cela revient à désactiver le préchargement pour un accès particulier.

BatchQueue utilise cette dernière technique pour indiquer à l'unité de préchargement que l'accès à un demi-tampon de communication est indépendant de l'accès à l'autre demi-tampon. La méthode utilisée consiste à créer un segment de mémoire partagée avec shm_open et effectuer deux associations entre adresses virtuelles et adresses physiques en utilisant l'appel système mmap. Ensuite, il suffit d'accéder à un demi-tampon via la première association et au deuxième demi-tampon via la deuxième association.

L'algorithme BatchQueue tel que présenté dans la section précédente souffre en plusieurs occasions de l'effet négatif du préchargement. En effet, le préchargement charge des lignes de cache qui sont modifiées juste après par un cœur distant, nécessitant alors une invalidation de la ligne de cache par le cœur distant. Deux techniques sont utilisés dans BatchQueue afin d'éviter ces effets négatifs : éloigner la variable *statut* des demi-tampons, et accéder aux demi-tampons par deux adresses virtuelles différentes, afin que l'unité de préchargement ne puisse détecter un accès séquentiel.

3. Sur les processeurs Intel il est possible de désactiver le préchargement via les registres spécifiques au modèle de processeur, communément appelé MSRs (« Model-specific register »).

3.3 Mesures de performance des algorithmes de communication

La conception de BatchQueue présentée dans la section précédente suggère que BatchQueue requiert moins de synchronisation entre le producteur et le consommateur que les autres algorithmes et qu'il peut éviter les effets de bord indésirables du préchargement des lignes de cache. Cette section présente une étude des performances pour confirmer l'efficacité de cet algorithme, d'autant plus que dans le cadre du partage de données, le comportement du cache est toujours difficile à prévoir.

Les algorithmes de communication sont nombreux et d'efficacité variée. Il est dès lors souhaitable de présenter les résultats de façon hiérarchique. À cette effet, l'évaluation de BatchQueue est répartie au sein de trois séries de mesures, chacune correspondant à un degré d'efficacité différent. La première série montre les ordres de grandeur entre les différentes catégories de communication. La seconde série compare les différents algorithmes de communication optimisés pour le multi-cœurs. Enfin, la dernière série étudie l'influence de la configuration matérielle sur les performances de BatchQueue.

3.3.1 Conditions expérimentales

Les algorithmes de communication spécialement optimisés pour la communication inter-cœurs sont très sensibles aux conditions expérimentales. Cette sous-section est donc dédiée à la description de ces conditions. La description des tests réalisés vient en premier, suivi par les détails sur les plateformes matérielles utilisées.

Deux tests sont utilisés pour évaluer les performances des algorithmes de communication inter-cœurs, chacun évaluant un aspect différent de l'algorithme. Le premier test, « comm », évalue la bande passante maximale que permet l'algorithme évalué. Le second test, « matrice », évalue la bande passante atteinte lorsque le cache de premier niveau est également utilisé de manière intensive par le programme effectuant la communication.

Test de communication intensive (*comm*) Le test *comm* consiste à envoyer une grande quantité de données, près de 3 Gio, et de mesurer le temps mis pour envoyer l'ensemble de ces données. Le temps est ensuite converti en débit. Les données sont envoyées les unes après les autres ; aucun calcul n'intervient entre deux envois de données. Le débit mesuré est donc un débit maximal.

Test de calcul matriciel (*matrice*) Le test *matrice* est proche du test *comm* dans son fonctionnement. Comme lui, le principe est d'envoyer une grande quantité de données – 3 Gio comme pour l'autre test – et de mesurer le temps mis pour les envoyer. Cependant cette fois-ci un calcul matriciel est effectué entre deux envois. Le but du calcul est de remplir le cache de manière à mesurer l'influence de la quantité de cache utilisée par l'algorithme de communication sur l'efficacité du calcul.

Plateforme matérielle Afin d'étudier l'impact du partage de caches et l'impact du partage de module mémoire, deux machines sont utilisées pour faire les évaluations : *bossa* et *amd48*. La machine *bossa* est alors utilisée pour toutes les évaluations exceptée celle sur une machine NUMA – machine dont l'accès à la mémoire est non uniforme (« Non Uniform Memory Access ») – dans la dernière sous-section.

La machine *bossa* contient deux processeurs Xeon X5472 ayant 4 cœurs de calcul cadencés à 3 GHz et une mémoire vive de 10 Gio. Les cœurs de calcul ont une mémoire cache de premier niveau de 32 Kio et une mémoire cache de second niveau de 6 Mio partagée par paire de cœurs. Le système d'exploitation est Debian GNU/Linux 6.0 « Squeeze » installé en 64 bits, tant pour le noyau Linux 3.2.0 que l'espace utilisateur.

La machine *amd48* est une machine NUMA contenant 4 sockets, chaque socket ayant 2 nœuds mémoire avec sur chacun 6 cœurs de calcul AMD Opteron 6172 cadencés à 2.1 GHz. Les cœurs de calcul ont un cache de premier niveau de 64 Kio et un cache de second niveau de 512 Kio. Le cache de troisième niveau à une taille de près de 5 Mio et est partagé entre tous les cœurs d'un nœud mémoire. Chaque nœud mémoire a un espèce de 8 Gio, pour un espace total de 32 gio. Le système d'exploitation est Ubuntu 11.10 « Oneiric Ocelot » installé en 64 bits, tant pour le noyau Linux 3.0.0 installé que l'espace utilisateur.

Les programmes sont compilés avec gcc 4.6.3, le niveau d'optimisation maximum (-O3) et les options -finline-functions -finline-functions-called-once. En revanche, aucune optimisation n'est faite lors de l'édition de lien.

3.3.2 Ordre de grandeur des algorithmes de communication inter-cœurs

Dans l'état de l'art présenté section 3.1, les algorithmes sont regroupés en trois catégories correspondant aux trois ordres de grandeur d'efficacité de ceux-ci.

Files utilisants des appels systèmes La première catégorie correspond aux solutions où la communication est effectuée par des primitives systèmes. Les solutions utilisant la primitive permettant d'obtenir un espace mémoire partagé ne sont pas inclus dans cette catégorie car cette primitive ne fournit pas de communication mais un moyen de construire un mécanisme de communication. Ces solutions de communications sont distinctes des autres car l'utilisation de primitives systèmes nécessite des changements de contexte très coûteux entre mode utilisateur et mode noyau.

Files sans verrou en mémoire partagée La deuxième catégorie correspond à toutes les solutions en mémoire partagée n'utilisant pas de verrou mais ne prêtant pas d'attention particulière au système de cohérence des caches. Cette catégorie regroupe la file sans verrou de Lamport et toutes les variantes avec plusieurs producteurs et plusieurs consommateurs. Parce qu'ils n'utilisent pas d'appels systèmes, les algorithmes de cette catégorie ne nécessitent pas de changement de contextes et sont donc bien plus efficaces que ceux de la première catégorie. Cependant, l'efficacité est limitée par la latence que génère le système de cohérence des caches permettant le partage des données.

Files optimisées pour le multi-cœurs Enfin, la troisième catégorie regroupe toutes les autres solutions, de FastForward à BatchQueue, sans oublier la communication utilisée dans Barrelfish, DBLS, MCRingBuffer et CSQ. Ces algorithmes représentent l'état de l'art de la communication inter-cœurs. Leur conception vise à se rapprocher du cas idéal où le nombre de lignes de cache échangées entre un cœur et un autre correspond à la quantité de données que l'algorithme vise à transmettre d'un cœur à un autre.

Afin de mettre en évidence cette hiérarchie de famille d'algorithmes, un algorithme de chaque catégorie est choisi : une communication par tube anonyme, l'algorithme de Lamport

et BatchQueue. Le test effectué est le test *comm*, l'influence de la concurrence d'accès au cache étant faible par rapport à l'efficacité de la communication pour les algorithmes les plus lents. Les résultats sont présentés dans la figure 3.7.

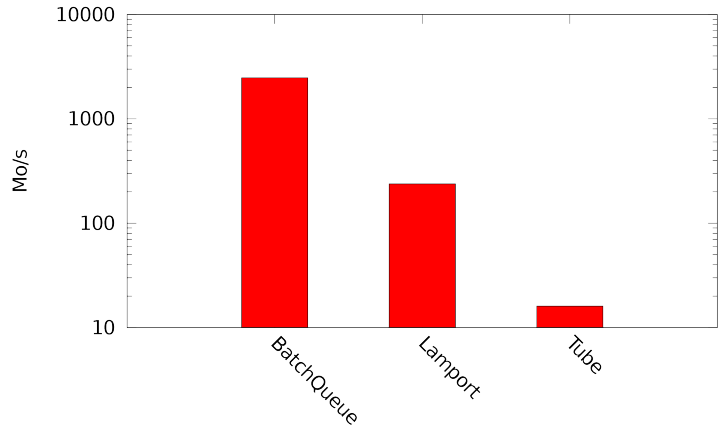


FIGURE 3.7 – Hiérarchie des performances dans les algorithmes de communication inter-cœurs

Comme attendu, les différences de performance entre les algorithmes des différentes catégories sont importantes. Le rapport de performance d'une catégorie à une autre oscille autour de 10. Une telle différence s'explique par le mode de fonctionnement général des différentes catégories – présenté en détail dans l'état de l'art, section 3.1. Les algorithmes fournis en tant que primitives système pénalisent les performances à cause des changements de contexte nécessaire pour chaque envoi de donnée. Les algorithmes dans la même famille que la file sans verrou de Lamport sont plus efficaces mais souffrent d'une synchronisation trop importante créant une communication excessive entre le cœur producteur et le cœur consommateur. Partant de cette constatation, la suite de cette section se focalise sur les algorithmes offrant des performances du même ordre de grandeur que celles de BatchQueue, c'est à dire les algorithmes optimisés pour les architectures multi-cœurs.

3.3.3 comparaison des algorithmes de communication optimisés pour le multi-cœurs

La hiérarchie qui existe entre les différentes catégories d'algorithmes justifie l'intérêt de disposer d'algorithmes de communication optimisés pour le multi-cœurs. Il existe cependant déjà de tels algorithmes ; la question se pose alors de savoir si BatchQueue présente un intérêt par rapport aux algorithmes existants. Cette section s'attache donc à présenter une évaluation des performances de ces différents algorithmes.

3.3.4 Paramètres des algorithmes de communication

Une des difficultés de l'évaluation réside dans la prise en compte des variations dans les tampons de communication utilisés par les algorithmes de communication. En effet, les algorithmes de communication ont par défaut – c'est à dire tel que décrit dans les articles

les présentant – un nombre de tampons et une taille de tampon qui leur sont propres. Ces différences ont un impact important sur les performances.

Impact de la taille des tampons Ainsi, avoir une grande taille de tampon permet d'améliorer les performances en réduisant la quantité de synchronisation entre le producteur et le consommateur. En effet, la synchronisation entre producteur et consommateur ne se produit que lorsqu'un tampon est rempli ou consommé. Plus un tampon est grand, plus le nombre de données envoyées par synchronisation est grand lui aussi. Un doublement de la taille d'un tampon permet donc de diviser par deux le nombre de synchronisations. Or une synchronisation représente un coût supplémentaire bien plus important que l'envoi d'une donnée. Une division par deux du nombre de synchronisations implique une amélioration importante des performances.

Impact du nombre de tampons Le nombre de tampons disponible a également un impact important sur les performances puisque cela permet de tolérer des variations plus importantes des vitesses de progression du producteur et du consommateur. Lorsque deux tampons uniquement sont utilisés, le producteur et le consommateur doivent traiter leur tampon à la même vitesse, sous peine que l'un doive attendre l'autre au moment de la synchronisation. Leurs vitesses peuvent varier mais au point de synchronisation ils doivent avoir mis le même temps pour traiter leur tampon. Dans le cas où trois tampons sont utilisés, la contrainte aux points de synchronisation est plus souple : le nombre de données traitées par le producteur et le consommateur ne doit pas dépasser le nombre de données contenues dans un tampon. Plus le nombre de tampons est élevé, plus la contrainte devient souple.

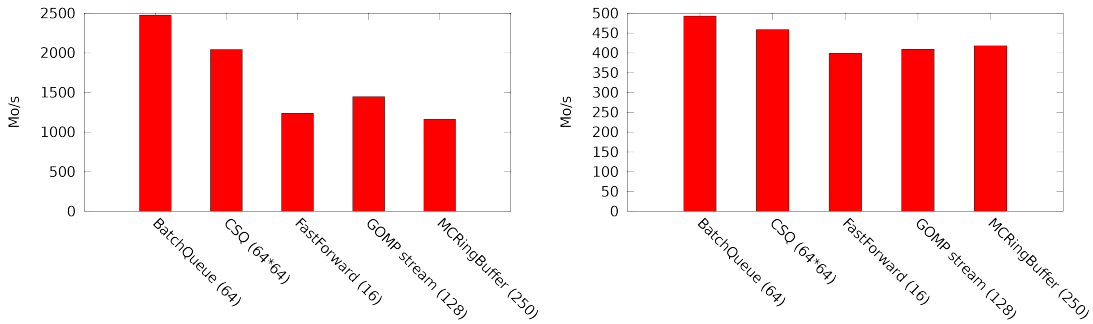
Impact de la taille globale des tampons Augmenter le nombre et la taille des tampons ne possède pas que des avantages. Cela a pour conséquence d'augmenter la taille globale de l'ensemble des tampons ce qui est pénalisant dans le cas où le cache de premier niveau est déjà utilisé intensivement. Plus la taille globale de tampons est importante, plus la fraction de cache utilisée pour stocker les tampons est grande. Il reste donc moins de place dans le cache pour d'autres données telle que celles manipulées par le programme effectuant la communication. La probabilité de créer un défaut de cache est augmentée en conséquence.

Afin de permettre une comparaison plus aisée, les algorithmes sont comparés dans deux paramétrages différents : l'un avec les valeurs par défaut suggérées dans les articles respectifs, l'autre où les tailles globales des tampons sont ramenées à une même valeur correspondant à 64 lignes de cache.

3.3.4.1 Test avec la configuration par défaut (tailles des tampons hétérogènes)

Les résultats pour le paramétrage avec les valeurs par défaut sont présentés dans la figure 3.8. Les nombres entre parenthèses indiquent le nombre de lignes de cache utilisées, avec dans le cas de CSQ le découpage de cet espace sous la forme $n * t$ où n est le nombre de tampons utilisés et t la taille des tampons.

Les résultats obtenus montrent que, lorsque les algorithmes sont dans leur paramétrage par défaut, BatchQueue est la solution la plus efficace. Pour une taille de tampon global de 64 lignes de cache, BatchQueue propose un débit plus important à la fois dans le cadre du débit



(a) Test *comm* (b) Test *matrice*

FIGURE 3.8 – Débits des algorithmes de communication optimisés pour le multi-cœurs dans leur paramétrage par défaut

maximal (test *comm*), et à la fois dans le cas où le cache de premier niveau est partagé entre l'algorithme de communication et un autre programme (test *matrice*). Ce résultat est d'autant plus remarquable considérant la taille globale de tampon nettement plus importante de CSQ. En effet, avec un tampon de 64 lignes de cache, BatchQueue obtient un débit 21% plus grand que CSQ alors que celui-ci utilise par défaut 64 tampons de 64 lignes de cache, soit un total de 4096 lignes de caches, 64 fois plus que BatchQueue. Ces résultats s'expliquent par le fait que BatchQueue propose, par rapport aux solutions concurrentes, (i) une synchronisation réduite entre le producteur et le consommateur et (ii) une empreinte mémoire faible.

Deux phénomènes expliquent que les différences de débits des différents algorithmes soient moins marquées dans le test *matrice* que dans le test *comm*. Tout d'abord, il se produit un recouvrement du temps de communication par le calcul. Le calcul matriciel ralentit le producteur permettant au consommateur d'être prêt pour la synchronisation plus tôt. Ce faisant, moins de temps est passé en communication ce qui tend à réduire les différences de vitesse d'exécution, en particulier entre BatchQueue et CSQ. Ensuite, la différence de débit entre BatchQueue et les autres techniques est moins visible sur le test avec calcul matriciel car le débit est calculé par rapport au temps total d'exécution. Or la proportion de temps passé à communiquer est faible au regard du temps passé à calculer. Une variation du temps de communication a donc un faible impact sur le temps d'exécution, a fortiori sur le débit. C'est ainsi que la différence de débit entre BatchQueue et CSQ est faible dans le test *matrice* bien que la différence des temps d'exécution est plus grande que dans le test *comm* : la différence des temps est de 0,28s pour le test *comm* et de 0,48s pour le test *matrice* pour une même quantité de données envoyées.

Un point à noter est que l'amélioration en performance pour CSQ ne provient pas d'une qualité intrinsèque de l'algorithme. En effet, tous les algorithmes proposés sont capables d'utiliser une taille globale de tampon plus grande ou plus petite que celle par défaut. Un avantage obtenu par un algorithme ayant une taille globale de tampon plus importante, comme c'est le cas pour CSQ, peut donc être obtenu également avec les autres algorithmes en augmentant leur tampon global de communication de la même façon.

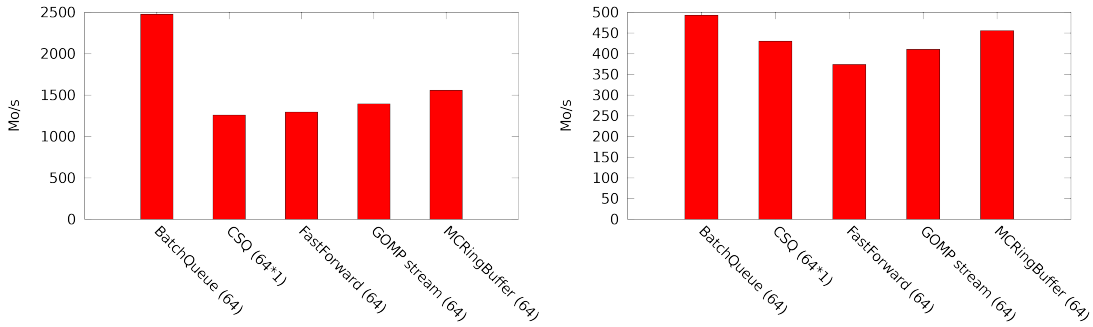
Par ailleurs, il ne faut pas oublier que l'augmentation de la taille globale des tampons a un effet sur les performances du reste du système. Un algorithme utilisant plus de cache laisse moins de place disponible à d'autres applications. Si la place disponible n'est pas suffisante,

cela crée une situation où l'algorithme et les autres programmes qui s'exécutent sur le même cœur se disputent les même lignes de cache. C'est cette situation que reproduit le test *matrice* et les résultats pour ce test mettent en évidence l'impact négatif sur les performances pour les algorithmes ayant une grande taille globale des tampons.

3.3.4.2 Test avec la configuration à taille des tampons fixée

La taille et le nombre des tampons de communication ont un impact important sur les performances des algorithmes de communication. En général, augmenter la taille et le nombre des tampons améliore les performances. Une taille de tampon importante réduit la quantité de synchronisation tandis qu'un nombre élevé de tampons permet de tolérer des variations plus importantes des vitesses de traitement des données par le producteur et le consommateur. Cependant, une taille globale des tampons plus grande est susceptible de créer plus de défauts de cache si par ailleurs le programme utilisant l'algorithme de communication utilise le cache de premier niveau de façon intensive. Dans un tel cas l'algorithme de communication et le programme sont en conflits pour utiliser les lignes de cache. Le test présenté figure 3.9 a pour but de comparer les algorithmes dans une configuration où la taille globale des tampons est la même pour tous.

Dans le cas de CSQ, la modification de la taille globale des tampons se fait en diminuant la taille des tampons individuels afin de garder le nombre de tampons à 64. En effet, l'article présentant CSQ étudie l'influence de plusieurs tailles de tampon et celle du nombre de tampons sur les performances mais le plus petit nombre de tampons étudié est 32, alors que le plus petit tampon suggéré est de la taille d'une ligne de cache.



(a) Test *comm* (b) Test *matrice*

FIGURE 3.9 – Débits des algorithmes de communication optimisés pour le multi-cœurs avec la taille globale de tampon fixé à 64 lignes de cache

Bien que sensiblement différente, la distribution des performances est toujours à l'avantage de BatchQueue dans la configuration avec taille globale de tampon uniforme. Pour une taille de tampon de 64 lignes de cache, BatchQueue propose un débit deux fois plus important que les autres algorithmes pour le test *comm*. Pour le test *matrice*, BatchQueue offre également un débit plus important que les autres algorithmes bien que le rapport entre BatchQueue et les autres algorithmes soit moins important dans ce test que dans le test *comm*. La différence entre les deux tests en ce qui concerne le rapport entre BatchQueue et les autres algorithmes s'explique en partie par la métrique mesurée comme expliqué dans la

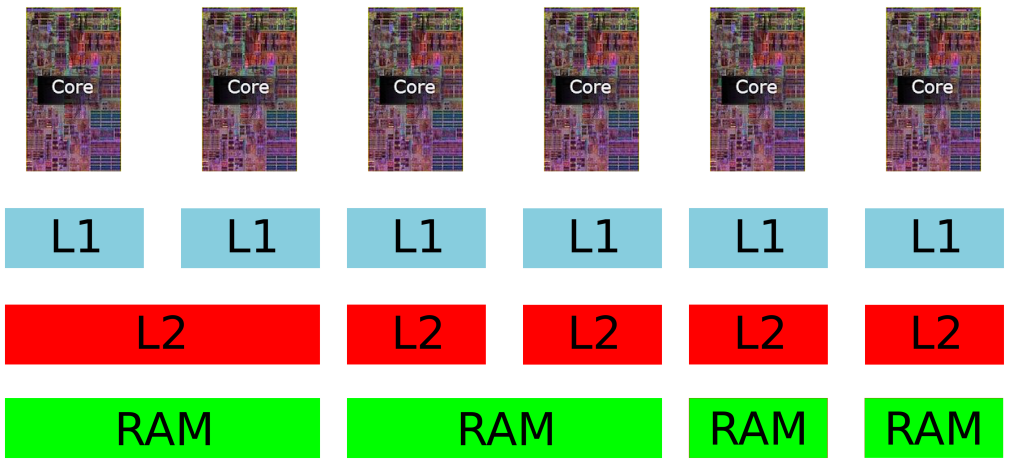
section 3.3.4.1, et en partie par le fait que tous les algorithmes utilisent la même taille globale de tampon. Les résultats de ce test sont d'autant plus intéressants que la différence de performances entre BatchQueue et les autres algorithmes ne provient dans ce cas que des qualités de l'algorithme et non de la taille des tampons réduite dans sa configuration par défaut.

Les performances que présente BatchQueue justifient de son intérêt par rapport aux algorithmes de communication existants. BatchQueue est la solution qui exploite le mieux une taille donnée de cache de premier niveau. En effet, à taille de tampon globale fixée, BatchQueue offre un débit bien plus important que les autres algorithmes tout en nécessitant moins de mémoire additionnelle pour son unique variable de synchronisation. De plus, BatchQueue est également une solution intéressante par rapport aux algorithmes existants dans leur configuration par défaut. BatchQueue obtient le meilleur débit maximal bien que la taille globale de tampon qu'il utilise soit plus petite que celle de CSQ et conserve également un meilleur débit lorsque la pression sur le cache de premier niveau augmente.

3.3.5 Influence du partage de cache

Les séries de mesure présentées ci-avant confirment l'intérêt de BatchQueue par rapport aux algorithmes de communication existants. En effet, la comparaison effectuée avec les autres algorithmes indique un débit environ deux fois plus important pour BatchQueue. De plus, celui-ci reste efficace avec une utilisation importante du cache.

Comme expliqué dans la section 3.2, les bons résultats obtenus par BatchQueue sont dûs à l'utilisation d'une unique variable de synchronisation ce qui limite la fréquence de synchronisation entre le producteur et le consommateur. Cette particularité a également un autre avantage : elle limite l'influence de la présence ou de l'absence d'un cache partagé. En effet, le faible partage de donnée entre le producteur et le consommateur rend BatchQueue relativement insensible à la hiérarchie matérielle des caches.



(a) Partage d'un cache matériel entre les cœurs (b) Absence de cache matériel partagé (c) Absence de mémoire centrale partagée

FIGURE 3.10 – Architectures multi-processeurs

Cette dernière série de mesure vise à vérifier cette hypothèse en comparant le débit obtenu par BatchQueue sur trois architectures de cache différentes, représentées par la figure 3.10 : une architecture avec un cache partagé (figure 3.10(a)), une architecture où seule la mémoire est partagée (figure 3.10(b)), et une architecture où les deux cœurs appartiennent à deux nœuds mémoire différents, c'est à dire étant rattachés à des mémoires différentes (figure 3.10(c)).

N'ayant pas accès à une machine présentant les trois configurations, les mesures sont effectuées sur deux machines différentes : *bossa* et *amd48*. Afin de pouvoir comparer les résultats, deux configurations différentes sont évaluées sur chaque machine de telle sorte qu'une configuration, celle à mémoire partagée, est évaluée sur les deux machines. La machine *bossa* est ainsi utilisée pour les configurations avec (3.10(a)) et sans (3.10(b)) cache matériel partagé mais disposant d'une mémoire centrale commune tandis que la machine *amd48*, ayant une architecture NUMA, est utilisée pour la configuration 3.10(a) à mémoire partagée mais sans cache partagé et pour la configuration 3.10(c) où les deux cœurs appartiennent à deux nœuds mémoire différents.

Les deux configurations avec mémoire centrale commune sont obtenues avec la machine *bossa* en clouant le producteur et le consommateur sur des cœurs partageant un cache ou non, suivant la configuration qui doit servir à l'évaluation. De la même façon, les deux configurations sans cache partagé sont obtenus avec la machine *amd48* en clouant le producteur et le consommateur sur des cœurs appartenant au même nœud mémoire ou non. Les résultats sur les quatre configurations avec les tests *comm* et *matrice* sont présentés dans les figures 3.11 et 3.12.

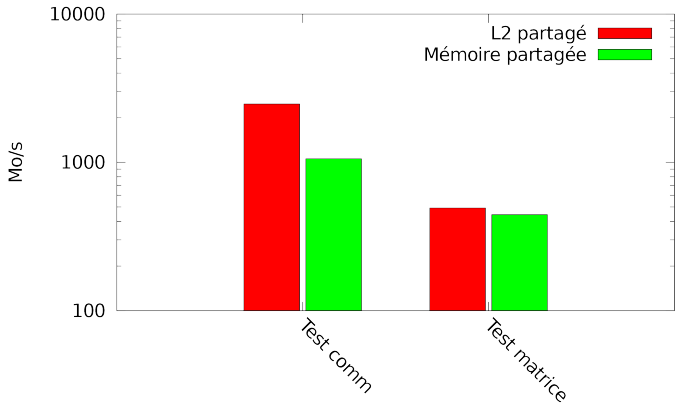


FIGURE 3.11 – Comparaison des débits de BatchQueue avec ou sans cache partagé

Il ressort de l'évaluation que le rapport de performances entre les différentes configurations est non linéaire par rapport au rapport du nombre de cycles nécessaire pour communiquer d'un cœur à un autre dans les différentes configurations. En effet, il faut environ 10 cycles pour communiquer entre deux cœurs ayant un cache de deuxième niveau partagé, environ 100 cycles lorsque la mémoire est partagée et environ 500 cycles si les cœurs sont situés sur des nœuds mémoire différent. Or les performances de BatchQueue ne sont dégradés dans le pire cas que de 30% en mémoire partagée mais en l'absence de cache de second niveau partagé et ne sont pas dégradés entre la configuration de référence et la configuration NUMA. La dégradation est moins importante dans le cas du test *matrice* car l'application

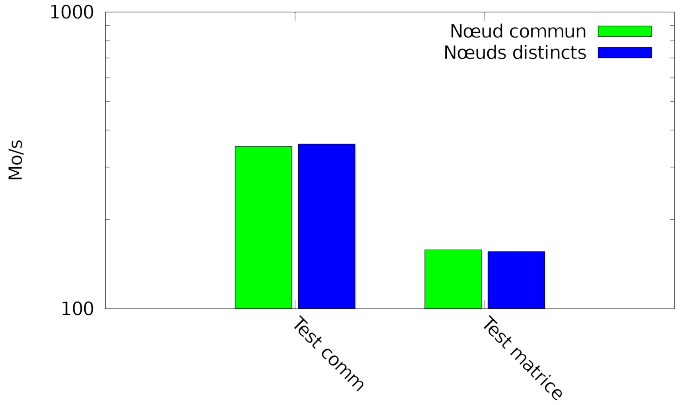


FIGURE 3.12 – Comparaison des débits de BatchQueue sur nœud mémoire distinct ou non

effectue le calcul matriciel la majeure partie du temps, la communication ne représentant qu'une petite fraction du temps d'exécution global.

Cette relative insensibilité à la latence d'accès aux données partagées montre que BatchQueue limite de façon efficace le partage de données entre le producteur et le consommateur. Un tel résultat pour l'architecture NUMA est une très bonne nouvelle car les architectures NUMA semblent devenir l'une des solutions matérielles de référence pour les machines ayant un grand nombre de cœurs. Ce résultat indique également que BatchQueue devrait bien se comporter dans le cadre de système NUCA [4].

3.4 Conclusion

Les résultats présentés dans la section précédente confirment les avantages de BatchQueue par rapport à l'état de l'art des algorithmes de communication. BatchQueue offre des performances améliorées par rapport aux algorithmes existants tant dans le cas du débit maximal que dans le cas où le cache de premier niveau subit une pression induite par un calcul matriciel. BatchQueue parvient à offrir un débit maximal deux fois plus grand que CSQ, la seconde solution la plus efficace, atteignant un débit de 2,5 Go/s. Enfin, le nombre réduit de synchronisations qu'offre sa conception permet également à BatchQueue d'être peu sensible à la latence d'accès aux données partagées de l'algorithme.

Ces caractéristiques font de BatchQueue une solution toute indiquée pour les applications communiquantes avec des contraintes de performance très forte. Ainsi BatchQueue est la solution adéquate pour les applications avec des pics fréquents de communication ainsi que les applications parallélisées sur un nombre de cœurs tel que la communication représente une part importante de son temps global d'exécution, comme c'est le cas des applications parallélisées sous forme de pipeline.

4. NUCA : Non Uniform Cache Architecture

Chapitre 4

Parallélisme de flux optimisé avec BatchQueue

Sommaire

DEPUIS plusieurs années, la fréquence des processeurs a cessé d'augmenter pour cause de contraintes physiques. Pour autant, la loi de Moore [M+98] continue de se vérifier pour la finesse de gravure. Les fabricants de processeurs profitent donc du nombre grandissant de transistors qu'il est possible d'intégrer par cm^2 pour augmenter le nombre de cœurs de calcul par circuit intégré [HP07, Sut05]. Cependant, de nombreuses applications ne sont encore composées que d'un nombre réduit de fils d'exécution. La parallélisation devient donc une condition nécessaire à l'utilisation efficace de ce nombre croissant de cœurs.

Plusieurs paradigmes peuvent être suivis afin de rendre une application parallèle. Parmi ceux-ci, le parallélisme de tâche et le parallélisme de donnée sont les deux principaux. Ces formes de parallélisme proviennent de la parallélisation de constructions de code couramment employées dans les programmes et sont donc très répandues. D'autres formes de parallélisme fondées sur la parallélisation de constructions moins courantes existent aussi, tel le parallélisme de flux. Celui-ci offre plusieurs avantages par rapport aux autres paradigmes

de parallélisation : il préserve la séquentialité des algorithmes, masque la latence d'accès à la mémoire et réduit l'utilisation du bus système permettant l'accès à la mémoire vive. Ces trois paradigmes de programmation parallèle sont présentés en détail dans la section 4.1.

Le fonctionnement du parallélisme de flux consiste à diviser le traitement effectué sur chaque donnée d'un flux en plusieurs étapes et exécuter chacune des étapes sur une unité de calcul différent. Les données migrent donc d'une unité de calcul à une autre pour subir l'ensemble du traitement. Dès lors, les performances de la communication inter-cœurs sont un élément clé de l'efficacité du parallélisme de flux.

Afin de simplifier la mise en œuvre du parallélisme de flux, les utilisateurs du parallélisme de flux peuvent se tourner vers des outils de parallélisation de haut niveau telle l'extension de calcul par flux pour OpenMP proposée par Pop et Cohen. Cette extension permet de mettre en œuvre le parallélisme de flux dans un programme en ajoutant des annotations dans son code. Afin de fournir de bonnes performances aux programmes utilisant cette extension, l'algorithme de communication utilisé est conçu de façon à être très efficace. Cependant, l'algorithme de communication utilisé permet la communication entre plusieurs producteurs et plusieurs consommateurs. Or il y a un coût important, inhérent à une solution aussi générale, qui peut être évité étant donné que le parallélisme de flux ne nécessite dans la plupart des cas qu'une communication entre un producteur et un consommateur.

Ce chapitre décrit ainsi l'utilisation de l'algorithme BatchQueue, décrit dans le chapitre précédent, afin d'améliorer les performances du parallélisme de flux que permet l'extension de calcul par flux pour OpenMP. La première section de ce chapitre est dédiée à la présentation des différents paradigmes de programmation parallèle, en particulier le parallélisme de flux, celui-ci étant moins répandu. Cette section aborde notamment l'utilisation de solutions de haut niveau pour mettre en œuvre ces différentes formes de parallélisme et offre également une comparaison des algorithmes entre eux. La seconde section décrit alors l'incorporation de l'algorithme BatchQueue au sein de l'extension de calcul par flux d'OpenMP afin d'en améliorer l'efficacité. Enfin, les deux dernières sections étudient en détail la performance de la solution ainsi obtenue tant d'un point de vue théorique que pratique.

4.1 Paradigmes de programmation parallèle

Plusieurs paradigmes de programmation parallèle sont à disposition des programmeurs [1] : le parallélisme de donnée, le parallélisme de tâche et le parallélisme de flux. Cette section est dédiée à la description de chacun de ces paradigmes. La première partie est consacrée à la description du parallélisme de donnée. Ensuite, la deuxième partie présente le parallélisme de tâche. Enfin, la dernière partie décrit le parallélisme de flux dont ce chapitre présente une amélioration des performances.

4.1.1 Parallélisme de données

4.1.1.1 Principe

Le parallélisme de donnée, aussi appelé parallélisme de boucle, est le paradigme de programmation parallèle qui consiste à permettre le traitement de plusieurs données à la fois.

1. il existe également des formes de parallélisme qui impliquent le matériel : le parallélisme de bit et le parallélisme d'instruction

Les boucles mènent très souvent à ce type de parallélisme car elles servent dans bien des cas à effectuer un traitement identique sur un ensemble de données indépendantes les unes des autres. Le parallélisme de donnée survient alors lorsque le matériel permet d'effectuer ce traitement sur plusieurs de ces données en même temps. Le parallélisme de donnée vise donc à augmenter le débit de traitement des données, c'est à dire le nombre de données qui peuvent être traitées en un temps donné.

Le traitement de plusieurs données peut s'effectuer au sein de la même unité d'exécution ou au sein de plusieurs unités distinctes. Le traitement au sein d'une même unité de calcul se présente sous la forme d'instructions dites SIMD [2] qui opèrent sur un ensemble de données en même temps. Par exemple, plusieurs données peuvent être multipliées par la même constante avec une seule instruction. Ce sont généralement des matériels spécialisés qui fournissent de telles instructions mais les processeurs d'ordinateurs personnels fournissent également un jeu d'instruction de ce type, tels les extensions de calculs de données par flots, appelées « Streaming SIMD Extensions ». À l'opposé, le traitement de plusieurs données en parallèle sur des unités d'exécution distinctes se pratique généralement sur des processeurs génériques, la seule condition requise étant d'avoir plusieurs unités d'exécution disponibles. Dans ce cas de figure, le traitement de plusieurs données s'effectue en traitant une donnée différente sur chaque processeur disponible, chaque processeur exécutant la même instruction.

Le fonctionnement des deux formes de parallélisme de donnée est représenté dans les figures 4.1 et 4.2.

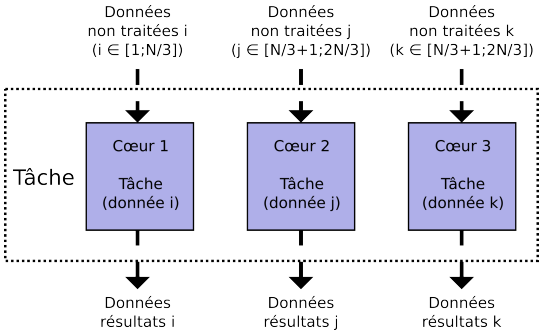


FIGURE 4.1 – Parallélisme de donnée sur un processeur multi-cœurs

4.1.1.2 Exemple

Afin de mieux comprendre comment appliquer le paradigme du parallélisme de donnée, considérons l'exemple présenté figure 4.1 :

Cet exemple montre la multiplication d'une matrice *mat* par une constante *a* dans le cas séquentiel. Chaque entrée est multipliée par la constante *a* l'une après l'autre. Chaque calcul est indépendant des précédents : la nouvelle valeur d'une entrée ne dépend que de son ancienne valeur et de la constante *a* mais pas des valeurs dans les autres entrées. Cette indépendance des calculs autorise donc à effectuer ces calculs en même temps, sans nécessiter

2. L'acronyme SIMD signifie « Simple Instruction Multiple Data » et désigne le fait qu'une instruction opère sur plusieurs données.

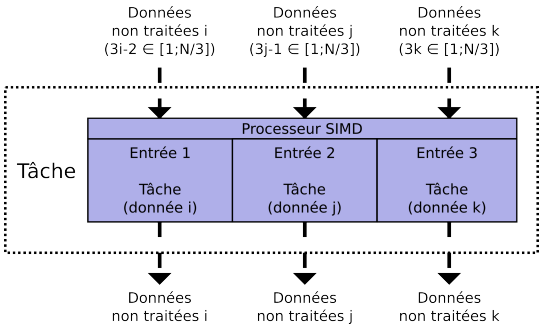


FIGURE 4.2 – Parallélisme de donnée sur un processeur SIMD

Listing 4.1 – Multiplication d'une matrice par un scalaire

```
1 for (i = 0; i < N; i++)
2   for (j = 0; j < N; j++)
3     mat[i][j] *= a;
```

aucune synchronisation. Il est ainsi possible de considérer la matrice comme étant une agrégation de plusieurs sous-matrices de taille M et d'effectuer la multiplication de chacune de ces sous-matrices sur des processeurs différents. Les processeurs possèdent tous un identifiant *id_proc* et exécutent le même code, présenté dans la figure 4.2 :

Listing 4.2 – Multiplication d'une matrice par un scalaire avec parallélisme de donnée

```
1 for (i = id_proc*M; i < (id_proc+1)*M; i++)
2   for (j = 0; j < N; j++)
3     mat[i][j] *= a;
```

Si les processeurs partagent une même mémoire, alors seul ce code est nécessaire pour effectuer la multiplication de la matrice par un scalaire. Dans le cas contraire, il faut auparavant envoyer les sous-matrices sur les processeurs qui vont les manipuler et transmettre les matrices résultat après le calcul. Cette distribution des données sur les différents processeurs permet ainsi de calculer les nouvelles valeurs de plusieurs entrées en même temps, celles-ci se trouvant dans différentes sous-matrices.

4.1.1.3 OpenMP

La mise en œuvre dans un programme du paradigme de parallélisme de donnée nécessite un effort non négligeable : il faut gérer la création de plusieurs fils d'exécutions, leur synchronisation ainsi que la distribution des données. Ajouter cette gestion à un programme, même lors son ébauche initiale, est susceptible de créer des bogues. De plus, la complexité du programme s'en trouve augmenté ce qui renforce d'autant les risques d'introduire des bogues lorsque celui-ci est modifié.

Pour toutes ces raisons, il existe des outils pour convertir un code séquentiel en un code parallèle avec un effort minimal. Ces outils visent à fournir une couche d'abstraction dans le but de gérer l'hétérogénéité des matériels et de cacher la gestion des fils d'exécution et la synchronisation. Cette abstraction est atteinte en fournissant une interface de programmation haut niveau et la bibliothèque logicielle qui va avec. L'interface peut prendre la forme d'un ensemble de fichiers d'en-tête, de nouveau mots clé du langage ou d'annotations à utiliser dans le code. De nombreux outils sont disponibles : OpenMP[Boa], Threading Building Blocks[Intc], Cilk Plus[Intb], Intel Array Building Block[Inta] pour en citer quelques uns. OpenMP est la solution de référence pour les systèmes à mémoire partagée. Ceci s'explique par les nombreux avantages qu'offre OpenMP par rapport aux solutions alternatives : OpenMP est une solution multi-plateforme, intégrée dans les compilateurs et relativement simple à utiliser.

L'interface que fournit OpenMP prends la forme d'annotations, appelés « pragmas », ajoutés dans le code. Ces pragmas indiquent au compilateur quelles parties du code peuvent être parallélisées et de quelle manière effectuer la parallélisation. Les pragmas OpenMP sont introduits par les mots clé « `#pragma omp` » en début de ligne suivi du nom d'une directive indiquant au compilateur quel comportement est souhaité. Les directives peuvent également avoir des clauses qui permettent de paramétrer plus finement le comportement d'une directive. Ainsi, OpenMP proposant un modèle mémoire où toutes les variables sont partagées par défaut, toutes les directives introduisant une section de code à exécuter en parallèle proposent une clause *private* pour indiquer des variables qui ne doivent pas être partagées. OpenMP est particulièrement simple à utiliser dans le cas du parallélisme de donnée car il propose une directive dédiée aux boucles de calcul : la directive *for*. Quant à la parallélisation proprement dite, elle est introduite par la directive *parallel*. Cette directive étant souvent combinée avec une directive indiquant la forme de parallélisation à utiliser, il est possible de combiner les deux directives sur la même ligne. La parallélisation de l'exemple présenté ci-avant est présentée figure 4.3.

Listing 4.3 – Multiplication d'une matrice par un scalaire avec OpenMP

```
1 #pragma omp parallel for private(j)
2 for (i = 0; i < N; i++)
3   for (j = 0; j < N; j++)
4     mat[i][j] *= a;
```

En utilisant OpenMP, l'ajout d'une seule ligne est nécessaire pour mettre en œuvre du parallélisme de donnée dans cette multiplication de matrice par un scalaire. Le code original ne nécessite alors plus d'être modifié et la création et la synchronisation entre les unités d'exécution est automatique. OpenMP parvient donc en effet à simplifier l'écriture d'un programme utilisant le parallélisme de donnée.

4.1.1.4 Avantages et inconvénients

Plusieurs paradigmes de programmation parallèle existent, il convient donc de s'interroger sur les avantages et inconvénients que chacun possède.

Avantage Dans le cas du parallélisme de donnée, sa force réside dans son passage à l'échelle au nombre d'unités d'exécution. En effet, prendre avantage d'un nombre plus important d'unités d'exécution consiste à traiter moins de données par unité d'exécution. Dans l'exemple de la multiplication de matrice, cela revient à traiter moins de lignes par unité d'exécution. Si le nombre d'unité devient très important, il est encore possible de paralléliser également la boucle interne. Avec OpenMP le passage à l'échelle est presque automatique : il est en effet possible d'indiquer à OpenMP le nombre d'unité d'exécution devant exécuter une section de code parallèle.

Inconvénient En revanche, le parallélisme de donnée est compliqué à utiliser en premier lieu à cause de la nécessité de disposer de calculs indépendants. L'existence de dépendances entre des calculs limitent fortement l'utilisation de parallélisme de donnée : elles forcent l'utilisation de synchronisation et réduisent l'efficacité du parallélisme résultant. Par exemple, le calcul consistant à remplacer chaque entrée *mat[i][j]* d'une matrice par la somme des entrées *mat[i][k]* pour k allant de 0 à j ne pourrait être réparti que sur N unités d'exécutions au maximum, N étant le nombre de lignes dans la matrice.

Le parallélisme de donnée est donc une solution intéressante pour traiter de grandes quantités de données car le traitement peut s'effectuer sur plusieurs d'entre elles en même temps en répartissant celles-ci sur un grand nombre d'unités d'exécution. De plus, l'utilisation d'outils de haut niveau tel OpenMP qui permettent de rendre une portion de code parallèle en ajoutant une annotation. En revanche, de nombreux problèmes ne se prêtent pas à cette forme de parallélisme car des dépendances existent entre les données.

4.1.2 Parallélisme de tâche

4.1.2.1 Principe

Le parallélisme de tâche est le paradigme de programmation parallèle qui consiste à permettre à plusieurs calculs de se dérouler en même temps. Alors que le parallélisme de donnée parallélise l'exécution d'un même traitement sur plusieurs données indépendantes, le parallélisme de tâche parallélise l'exécution de plusieurs tâches indépendantes. Cette forme de parallélisme est donc axée sur les calculs effectués plutôt que sur les données transformées par les calculs. En conséquence, plusieurs données peuvent être manipulées par une même tâche et plusieurs tâches peuvent manipuler des données communes, à condition que les calculs effectués par ces tâches restent suffisamment indépendants les uns des autres. Le parallélisme de tâche permet notamment de rendre une machine plus réactive puisque celle-ci peut traiter plusieurs tâches en même temps.

La quantité de synchronisation utilisée est une autre différence entre le parallélisme de donnée et le parallélisme de tâche. Puisque le parallélisme de donnée parallélise des calculs sur des données indépendantes, la synchronisation est généralement limitée à la fin du traitement de toutes les données pour ne continuer l'exécution du programme qu'une fois le calcul complet effectué. En revanche, le parallélisme de tâche parallélise des calculs qui peuvent avoir des impacts l'un sur l'autre. Il est donc assez fréquent d'avoir des synchronisations au sein des tâches parallélisées par ce paradigme.

Contrairement au parallélisme de donnée, le parallélisme de tâche s'effectue nécessairement au sein d'unités d'exécutions différentes. En effet, les tâches que ce paradigme vise à

paralléliser sont souvent des segments de code différents, c'est à dire que le code n'est pas le même pour les différentes tâches. Dès lors, le parallélisme de tâche requiert des systèmes pouvant gérer plusieurs fils d'exécution en même temps, qu'il s'agisse de système multi-processeurs, de systèmes multi-cœurs ou de système à multitraitement simultané (« Simultaneous MultiThreading », plus connu sous le sigle SMT). Ces systèmes ont donc un comportement proche des systèmes répartis, au partage des ressources mémoire près.

Le fonctionnement de cette forme de parallélisme est représenté dans la figure 4.3.

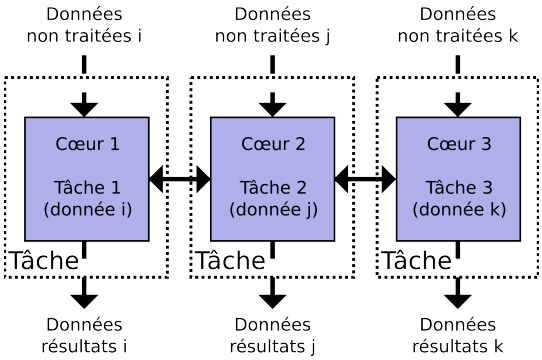


FIGURE 4.3 – Flux de données avec parallélisme de tâche

4.1.2.2 Exemple

L'exemple présenté figure 4.4 permet de mieux comprendre comment mettre en pratique le paradigme du parallélisme de tâche au travers de la parallélisation de deux calculs différents.

Listing 4.4 – Opérations scalaires sur deux matrices

```
1 mult_matrix_scalar(A, coeff);
2 add_matrix_scalar(B, cst);
3 add_matrices(A, B);
```

Cet exemple reprends le contexte de l'exemple utilisé pour le parallélisme de donnée et considère que la multiplication de matrice par un scalaire fait partie d'un plus grand programme. Le but de celui-ci est alors d'effectuer cette multiplication sur une matrice A avec un coefficient *coeff*, d'ajouter une constante *cst* à toutes les entrées d'une seconde matrice B puis d'ajouter les deux matrices résultats. Les deux premiers calculs sont indépendants et ne nécessitent donc pas de synchronisation pour être exécutés en parallèle. Par contre, ces calculs étant différents, ils ne relèvent pas du parallélisme de donnée mais du parallélisme de tâche. La parallélisation de ce programme consiste à exécuter les deux premiers calculs sur des unités d'exécution différentes puis d'effectuer une synchronisation avant de poursuivre le programme avec l'addition des deux matrices obtenues.

Le code correspondant devient ainsi séparé en trois segments de code distincts, présentés figures 4.5, 4.6 et 4.7 : un pour la multiplication de la matrice A par un scalaire, un pour

l'incrément des éléments de la matrice B par un autre scalaire et un qui attend les résultats des deux autres et effectue l'addition des deux matrices résultantes.

Listing 4.5 – Multiplication d'une matrice par un scalaire avec parallélisme de tâche

```
1 mult_matrix_scalar(A, coeff);
```

Listing 4.6 – Incrémentation des éléments d'une matrice avec parallélisme de tâche

```
1 add_matrix_scalar(B, cst);
```

Listing 4.7 – Addition de deux matrices calculées avec parallélisme de tâche

```
1 wait_computations(A, B);
2 add_matrices(A, B);
```

De la même façon qu'avec le parallélisme de donnée, si les processeurs partagent une même mémoire, alors seuls ces trois codes sont nécessaires pour effectuer la multiplication de la première matrice, l'incrément de toutes les entrées de la seconde et l'addition des deux matrices obtenues. La gestion des fils d'exécution reste néanmoins nécessaire, de même que la synchronisation qui doit être ajoutée explicitement. Il apparaît clairement avec cet exemple que les deux paradigmes peuvent être combinés : il est ainsi possible d'utiliser le parallélisme de donnée pour effectuer les calculs des deux matrices sur plusieurs entrées en même temps et tirer parti du parallélisme de tâche pour effectuer ces deux calculs en même temps.

4.1.2.3 OpenMP

Les outils permettant d'éviter la gestion bas niveau du parallélisme de donnée offre également une gestion du parallélisme de tâche. En particulier, OpenMP dispose de directives adaptées à ce parallélisme, évitant ainsi une gestion fastidieuse des fils d'exécution et de la synchronisation. OpenMP fournit ainsi deux directives permettant de faire du parallélisme de tâche : les directives *sections* et *task*. Dans le cadre de l'exemple ci-dessus, c'est la directive *sections* qui est la plus adaptée. Comme les deux calculs opèrent sur des données distinctes, le partage des variables effectué par défaut par cette directive est adapté. La parallélisation de l'exemple présenté ci-avant est présenté figure 4.8.

Une fois de plus, l'utilisation d'OpenMP permet de simplifier la mise en œuvre du parallélisme, le parallélisme de tâche dans le cas présent. Le fonctionnement de la directive *sections* consiste à exécuter chaque bloc de code annoté par la directive *section* sur une unité d'exécution différente. Dans le cas présent les deux calculs de matrice indépendants sont annotés par la directive *section* et sont donc exécutés en parallèle sur des unités d'exécution différentes. À la fin de la section, une barrière implicite est effectuée de telle sorte que l'addition des matrices A et B n'a lieu qu'une fois que le calcul de ces matrices est effectué. Bien que la syntaxe soit un peu plus complexe que dans le cas du parallélisme de donnée,

Listing 4.8 – Opérations scalaires sur deux matrices avec OpenMP

```
#pragma omp parallel sections
{
  #pragma omp section
  mult_matrix_scalar(A, coeff);

  #pragma omp section
  add_matrix_scalar(B, cst);
}
add_matrices(A, B);
```

OpenMP parvient malgré tout à simplifier l'écriture d'un programme utilisant le parallélisme de tâche.

4.1.2.4 Avantages et inconvénients

Avantage Le parallélisme de tâche vise à paralléliser des codes différents de ceux parallélisés par le parallélisme de donnée ; une comparaison des deux paradigmes est donc difficile. Néanmoins, leur facilité d'utilisation peut être comparée. Un avantage que possède le parallélisme de tâche, y compris par rapport au parallélisme de flux présenté ci-après, est la flexibilité qu'il permet. En effet, tout ce qui est fait avec du parallélisme de donnée ou du parallélisme de flux peut être fait avec du parallélisme de tâche bien que celui-ci soit généralement associé à des tâches relativement distinctes, c'est à dire ne participant pas au même calcul. En particulier, le parallélisme de tâche permet d'exécuter n'importe quel code et d'effectuer une synchronisation à tout moment. Il est ainsi possible d'avoir des dépendances entre les différentes tâches participant au parallélisme de tâche.

Inconvénient Une contrepartie à la flexibilité que propose le parallélisme de tâche est la complexité inhérente à son utilisation. En dehors des cas triviaux, il faut la plupart du temps recourir à de la synchronisation là où les autres formes de parallélisme y ont recours plus épisodiquement. De plus, le modèle mental que propose le parallélisme de tâche est moins structuré. Il est facile de penser au parallélisme de donnée en terme de boucles et le parallélisme de flux en terme de flux à gérer et ces types de code sont souvent un bon indice qu'une parallélisation est possible à l'aide de ces paradigmes. En revanche, il peut être plus difficile d'identifier des tâches parallélisables au sein d'un programme et encore plus difficile d'appréhender toutes les synchronisations qui ont lieu entre elles.

Une autre faiblesse du parallélisme de tâche est le découpage statique et manuel des tâches. Il faut en effet indiquer manuellement dans le code où se situent les blocs indépendants de code afin qu'ils puissent être parallélisés. En conséquence, un seul ensemble d'annotations ne peut gérer de manière optimale tous les systèmes puisque pour y arriver le nombre de blocs devrait varier en fonction du nombre d'unités d'exécution disponibles, et donc le découpage également. Or celui-ci doit être fait manuellement dans le code source et non automatiquement à l'exécution. Cette situation est différente du parallélisme de données où les données sont distribuées en nombre égal sur chaque unité d'exécution : seul

un nombre de données supérieur au nombre d'unités d'exécution est alors nécessaire. Le découpage statique et manuel fixe donc les performances maximales qui peuvent être obtenues pour un ensemble d'annotations. En effet, l'accélération maximale que permet le parallélisme de tâche correspond au nombre de tâches délimitées dans le code.

Le parallélisme de tâche est une solution adaptée pour le parallélisme gros grains, en particulier pour paralléliser des traitements de longue durée ayant des dépendances entre eux. Sa flexibilité le rend adapté à la parallélisation de programmes complexes constitués de plusieurs tâches plus ou moins indépendantes, au prix d'un résultat encore plus complexe. L'utilisation d'outil haut niveau est possible mais masque plus difficilement la complexité de cette forme de parallélisme. De plus, il est souvent nécessaire de combiner le parallélisme de tâche avec une autre forme de parallélisme afin de bénéficier d'une amélioration des performances lorsqu'un grand nombre d'unités d'exécution est disponible. Une classe d'application échappe néanmoins à ce problème de passage à l'échelle et se prête particulièrement bien à cette forme de parallélisme : les serveurs.

4.1.3 Parallélisme de flux

4.1.3.1 Principe

Le parallélisme de flux est le paradigme de programmation parallèle qui consiste à permettre d'exécuter plusieurs étapes d'un traitement séquentiel en parallèle en leur faisant traiter des données différentes. Le traitement complet est alors atteint pour les données en les faisant migrer d'une étape à une autre dans l'ordre dans lequel elles apparaissent dans le traitement afin de garder sa séquentialité. Les données se succèdent ainsi les unes aux autres dans les différentes étapes formant un flux de données – d'où le nom de cette forme de parallélisme. Le fonctionnement de cette forme de parallélisme est représenté dans la figure 4.4.

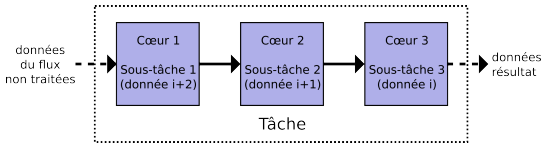


FIGURE 4.4 – Flux de données avec parallélisme de flux

Les données se succédant les unes aux autres dans les différentes étapes, le traitement effectué dans une étape donnée peut dépendre des traitements effectués par cette étape pour les données précédentes. Ce fonctionnement permet au parallélisme de flux de paralléliser des traitements avec de fortes dépendances entre les données sans avoir recours à de nombreuses synchronisations. Il cumule ainsi les avantages du parallélisme de donnée et du parallélisme de tâche.

Comme pour le parallélisme de tâche, le parallélisme de flux s'effectue nécessairement au sein d'unités d'exécution différentes. Les différentes étapes qui s'exécutent en parallèle dans cette forme de parallélisme sont composées de multiples instructions et sont différentes dans le cas général. Le parallélisme de flux requiert donc des systèmes de type multi-processeurs ou multi-cœurs.

4.1.3.2 Exemple

Le parallélisme de flux apparaît naturellement lors de multiples traitements d'un flux de données, aussi l'exemple présenté figure 4.9 prend pour contexte le traitement d'un flux audio provenant du réseau.

Listing 4.9 – Capture et réencodage d'un flux audio provenant du réseau

```
1 while (1) {
2   encoded_highrate_sample = recv_sample_from_network();
3   raw_highrate_sample = decode_sample(encoded_highrate_sample, &decoding_state)
4   encoded_lowrate_sample = resample(raw_highrate_sample, &sampling_state);
5   append_to_file(encoded_lowrate_sample, output_file);
6 }
```

Dans cet exemple, les variables *decoding_state* et *sampling_state* représentent la dépendance existant entre les échantillons audio successifs. Dans les flux audio et video compressés, les échantillons sont exprimés les uns en fonction des autres sous forme de « diff » pour être codés de façon plus concise. L'exemple présenté considère un flux avec des dépendances arrières, c'est à dire des dépendances entre chaque échantillon et les suivants. Il existe cependant des codages avec des dépendances avant et arrière ce qui rend leur parallélisation suivant ce paradigme plus complexe.

La parallélisation de cette portion de code consiste à exécuter chacune des lignes dans la boucle sur une unité d'exécution distincte et de transférer les résultats de l'exécution de chaque ligne vers l'unité d'exécution exécutant la ligne suivante. Les variables *decoding_state* et *sampling_state* n'étant utilisées que localement, il n'est pas nécessaire de les transférer.

Listing 4.10 – Réception d'un flux audio depuis le réseau avec parallélisme de flux

```
1 while (1) {
2   encoded_highrate_sample = recv_sample_from_network();
3   send_to_next_core(encoded_highrate_sample);
4 }
```

Listing 4.11 – Décodage d'un flux audio avec parallélisme de flux

```
1 while (1) {
2   recv_from_prev_core(encoded_highrate_sample);
3   raw_highrate_sample = decode_sample(encoded_highrate_sample, &decoding_state)
4   send_to_next_core(raw_highrate_sample);
5 }
```

À l'opposé du parallélisme de donnée et du parallélisme de tâche, la communication est explicite dans le cas du parallélisme de flux, y compris si les différentes unités d'exécution

Listing 4.12 – Réencodage d'un flux audio avec parallélisme de flux

```
while (1) {
  recv_from_prev_core(raw_highrate_sample);
  encoded_lowrate_sample = resample(raw_highrate_sample, &sampling_state);
  send_to_next_core(encoded_lowrate_sample);
}
```

Listing 4.13 – Capture d'un flux audio avec parallélisme de flux

```
while (1) {
  recv_from_prev_core(encoded_lowrate_sample);
  append_to_file(encoded_lowrate_sample, output_file);
}
```

partagent une même mémoire. La communication s'explique par la synchronisation qui a lieu entre les différentes étapes. En effet, les données doivent migrer toutes en même temps car une étape ne peut traiter deux données en même temps, sauf dans le cas d'une combinaison du parallélisme de flux avec une autre forme de parallélisme. Cette synchronisation explicite requiert donc des changements plus importants pour qu'un programme mette en œuvre le parallélisme de flux. Ce coût est cependant acceptable puisqu'un cas comme celui présenté dans cette section ne peut être parallélisé par les autres formes de parallélisme.

4.1.3.3 Extension de calcul par flux pour OpenMP

La présence importante de synchronisation dans le parallélisme de flux rend l'intérêt pour un outil plus haut niveau encore plus grand que pour les autres formes de parallélisme. Cependant, l'outil haut niveau de référence StreamIt [Wil02] est une solution dédiée au parallélisme de flux et n'est donc pas un outil familier au nombre important de programmeurs connaissant déjà le parallélisme de tâche ou le parallélisme de donnée. C'est dans ce but qu'une extension pour OpenMP a été mise au point [PC11] et il est probable qu'elle soit intégrée à OpenMP. Cette extension propose une interface semblable à celle des directives natives d'OpenMP de façon à proposer un cadre familier aux programmeurs habitués à utiliser OpenMP.

L'interface que propose cette extension se structure autour de la directive *task*, déjà présentée dans la section 4.1.2.3 sur le parallélisme de tâche. Cette extension ajoute les clauses *input* et *output* à la directive *task* pour indiquer le flux des données. Les clauses *firstprivate* et *lastprivate* sont également ajoutées pour permettre une interaction avec les données venant de l'environnement, c'est à dire non gérées par le parallélisme de flux. Les données migrent d'une clause *output* à la clause *input* de même nom. À la différence des directives *for* et *sections*, le bloc annoté par une directive *task* n'est pas partagé entre les différents fils d'exécution qui rencontre cette directive mais est au contraire exécuté dans son intégralité. Il est donc nécessaire d'avoir recours à la directive *single* afin qu'un fil d'exécution différent exécute chaque directive *task*. La parallélisation de l'exemple présenté dans la section

précédente est visible dans la figure 4.14.

Listing 4.14 – Capture et réencodage d'un flux audio provenant du réseau avec l'extension de calcul par flux pour OpenMP

```
#pragma omp parallel
#pragma omp single
while (1) {
  #pragma omp task output (encoded_highrate_sample)
  encoded_highrate_sample = recv_sample_from_network();

  #pragma omp task input (encoded_highrate_sample) output (raw_highrate_sample)
  raw_highrate_sample = decode_sample(encoded_highrate_sample, &decoding_state)

  #pragma omp task input (raw_highrate_sample) output (encoded_lowrate_sample)
  encoded_lowrate_sample = resample(raw_highrate_sample, &sampling_state);

  #pragma omp task input (encoded_lowrate_sample) shared (output_file)
  append_to_file(encoded_lowrate_sample, output_file);
}
```

Étant donné la quantité de synchronisation nécessaire pour le parallélisme de flux, l'utilisation de cette extension pour OpenMP est d'autant plus intéressante. De plus, la communication joue un rôle prépondérant dans l'efficacité d'une solution utilisant le parallélisme de flux. L'utilisation de cette extension permet d'utiliser l'algorithme de communication qui vient avec et de profiter de toute amélioration dont celui-ci pourrait bénéficier. La parallélisation manuelle en revanche nécessite d'implémenter son propre algorithme qui a toute les chances d'être ou de devenir à terme moins efficace que celui faisant partie de l'extension pour OpenMP. Il est donc vivement recommandé pour un programmeur envisageant d'avoir recours au parallélisme de flux d'utiliser cette extension.

4.1.3.4 Avantages et inconvénients

Avantages L'intérêt premier du parallélisme de flux est sans conteste sa capacité à paralléliser simplement des programmes possédant de fortes dépendances de données. Son utilisation est en effet simplifiée par le fait que cette forme de parallélisme conserve la séquentialité du traitement parallélisé. Le parallélisme de flux est donc le choix privilégié pour toute une catégorie de programmes complexes. Le parallélisme de flux est en particulier bien adapté pour le traitement de flux audio et vidéo ainsi que les traitements impliquant plusieurs filtres à exécuter à la suite les uns des autres, comme le traitement d'images.

De plus, le parallélisme de flux fait un bon usage de la mémoire pour l'application parallélisée elle-même mais aussi vis à vis des autres programmes du système. Du point de vue de l'application, le parallélisme de flux se distingue par sa capacité à masquer la latence d'accès à la mémoire. En effet, le parallélisme de flux permet d'effectuer naturellement un recouvrement de la communication par le calcul en découplant la production de la consommation. Du point de vue du système, la communication très locale du parallélisme de flux offre une

amélioration des performances sans ralentir les autres applications. À la différence du parallélisme de donnée qui effectue une dispersion des données depuis une unité d'exécution vers toutes les autres, le parallélisme de flux n'effectue une communication que point à point entre des unités d'exécution voisines. Sur un système NUMA [], la communication est donc interne au nœud, à l'exception de la dernière unité d'exécution d'un nœud qui communique avec l'unité se trouvant dans le nœud suivant.

Enfin, le parallélisme de flux permet un découpage dynamique et automatique du traitement séquentiel effectué sur les données du flux en sous-tâches lorsque le traitement prends la forme d'une boucle de calcul. Dans ces cas-là, la boucle peut être divisée en une séquence de boucles de tailles plus petites formant les différentes étapes du parallélisme de flux. Un seul jeu d'annotations est alors suffisant pour qu'un code donné utilise toutes les unités de calcul disponibles sur un système quel qu'en soit leur nombre en divisant la boucle de calcul en plus ou moins de sous-boucles, et donc en utilisant plus ou moins d'étapes.

Inconvénients En dépit de ses nombreux avantages, certains éléments sont à prendre en considération lors de la mise en œuvre du parallélisme de flux. Pour commencer, le parallélisme de flux n'offre des performances optimales que dans le cas où toutes les étapes ont une durée d'exécution identique. Si une étape est plus longue que les autres à s'exécuter, le débit du flux sera imposé par celle-ci. Il est impossible pour les étapes suivantes de traiter les données plus vite qu'elles ne sont traitées par l'étape la plus lente.

L'autre élément à prendre en compte en choisissant d'utiliser le parallélisme de flux est que l'efficacité de son passage à l'échelle dépend des performances de la communication entre les unités de calcul. L'accélération que permet le parallélisme de flux repose sur le fait que plus il y a d'unités d'exécution à disposition, plus le nombre de données à des étapes différentes du calcul est élevé. Quand une étape est achevée, la donnée qui était traitée migre vers l'étape suivante. Pour la donnée qui exécutait la dernière étape, cela signifie que son traitement est complètement fini. Étant donné (i) un temps de communication entre deux cœurs T_{comm}, (ii) une application dont le temps d'exécution sans parallélisation est T_{seq} et (iii) un nombre d'unités d'exécution disponibles n, une donnée finit son traitement toutes les $\frac{T_{seq}}{n} + T_{comm}$ secondes. Augmenter le nombre d'unités d'exécution ne réduit que le temps nécessaire pour exécuter une étape qui est de $\frac{T_{seq}}{n}$. Le débit de l'application est donc limité à une donnée toutes les T_{comm} secondes au mieux. Cela signifie également que le débit est amélioré de façon significative aussi longtemps que $T_{comm} << \frac{T_{seq}}{n}$. Il est donc essentiel d'avoir un algorithme de communication aussi rapide que possible afin d'avoir un passage à l'échelle aussi bon que possible. La figure 4.5 montre comment l'accélération varie pour plusieurs valeurs du rapport $\frac{T_{comm}}{T_{seq}}$.

Considérations autour du parallélisme de flux Le parallélisme de flux offre des avantages uniques par rapport aux autres paradigmes de programmation parallèle. Celui-ci est capable de paralléliser des programmes complexes avec de fortes dépendances entre les données. C'est également un modèle assez simple à utiliser car il préserve la séquentialité de l'algorithme parallélisé. Enfin, il parvient à utiliser la mémoire intelligemment en masquant la latence d'accès à celle-ci et en favorisant la communication locale, limitant ainsi l'impact sur les autres applications du système. La mise en œuvre du parallélisme de flux requiert néanmoins des précautions. Pour obtenir les meilleurs performances, il est en effet nécessaire de découper le traitement en étapes de durées les plus égales possible et de disposer d'un

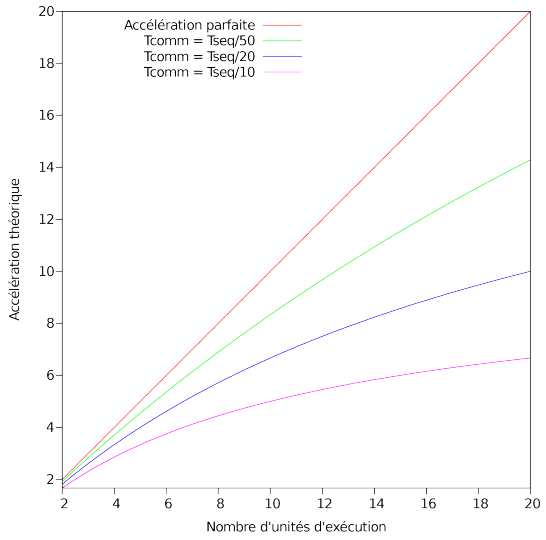


FIGURE 4.5 – Influence du débit de la communication sur l'accélération fournie par le parallélisme de flux

algorithme de communication offrant le meilleur débit possible.

L'efficacité de l'algorithme de communication présente un intérêt particulièrement grand car contrairement au découpage des étapes propre à chaque programme, il peut être le même pour tous. Il y a donc un intérêt à disposer d'un algorithme très efficace car celui-ci peut profiter à tous les programmes utilisant le parallélisme de flux. Ceci est d'autant plus vrai dans le cadre de l'extension de calcul par flux pour OpenMP car l'algorithme de communication compris dans celle-ci est utilisé automatiquement par tous les programmes ayant recours à cette extension. C'est la raison pour laquelle la section suivante présente l'intégration de l'algorithme BatchQueue, dont l'efficacité a été démontrée dans le chapitre précédent, au sein de l'extension de calcul par flux pour OpenMP.

4.2 Implémentation

La mise en œuvre automatique du parallélisme qu'offre OpenMP permet d'améliorer les performances d'un vaste ensemble de programmes en changeant le code qui est généré lorsque des annotations sont rencontrées. Dans le cadre de l'extension de calcul par flux pour OpenMP, une amélioration du parallélisme de flux est ainsi possible en remplaçant l'algorithme de communication inter-cœurs par un autre plus efficace. En effet, comme expliqué dans la section précédente, l'efficacité de cette communication est un élément clé de l'accélération possible avec le parallélisme de flux.

La section suivante présente l'intégration de l'algorithme de communication BatchQueue au sein de l'extension de calcul par flux pour OpenMP en vue d'améliorer les performances du parallélisme de flux proposé par cette extension. Une implémentation d'OpenMP se décompose en deux parties : une partie compilation pour analyser les annotations OpenMP et une partie exécution pour mettre en œuvre le parallélisme. Il en va de même pour l'extension de calcul par flux pour OpenMP. Celle-ci se compose de modifications dans la collection de

compilateurs GNU [GNU] pour supporter les nouvelles annotations et d'une bibliothèque dynamique offrant les primitives qui implémentent l'algorithme de communication. La section est divisée en deux parties où sont présentées les modifications au sein de chacun de ces composants.

4.2.1 Adaptation à l'interface de communication de l'extension de calcul par flux pour OpenMP

La décomposition d'une implémentation OpenMP et de l'extension de calcul par flux pour OpenMP en deux parties n'est pas une nécessité technique mais un choix de conception. Il est possible de générer tout le code de parallélisation depuis le compilateur et ainsi n'avoir pas besoin d'une bibliothèque dynamique à l'exécution. L'approche sous forme de deux composants se justifie cependant par les avantages qu'elle comporte. Un premier avantage est la simplification de l'implémentation. Le code source correspondant à la génération de code parallèle est un code complexe et verbeux aussi est-il préférable d'externaliser autant de code que possible dans la bibliothèque dynamique. La génération de code est ainsi réduite à la génération d'appels de fonctions présentes dans la bibliothèque dynamique, ce qui est moins difficile à effectuer.

Un second et plus important avantage est la possibilité de changer les algorithmes mettant en œuvre le parallélisme sans avoir à recompiler les programmes annotés via une mise à jour de la bibliothèque dynamique. Cela ouvre la voie à une amélioration des programmes existants sans avoir à les modifier ni même les recompiler. Changer la bibliothèque dynamique sans recompiler les programmes nécessite de conserver l'interface binaire – plus communément appelée ABI, pour « Application Binary Interface » – de la bibliothèque, c'est à dire de conserver l'ensemble des fonctions, leurs paramètres et leur comportement. Lorsque cela n'est pas possible, conserver l'interface de programmation – ou API, pour « Application Programming Interface » – permet de bénéficier d'un nouvel algorithme en recompilant le programme sans le modifier. De plus, aucune modification dans l'algorithme de génération de code parallèle n'est alors nécessaire. C'est pourquoi une adaptation de BatchQueue à l'interface de programmation a été réalisée.

L'adaptation de BatchQueue à l'interface de programmation de la bibliothèque dynamique repose sur trois ensembles de modifications. Une première partie des modifications consiste à adopter les structures utilisées dans l'extension. Celles-ci sont en effet utilisées en divers endroits de l'interface pour maintenir l'état de la communication entre les différents appels de fonction. Une deuxième partie des modifications provient de ce que la communication peut se faire sans copie de données. L'envoi et la réception se font en deux étapes : la réservation d'une partie du tampon de communication puis la notification que les données y ont été produites ou consommées. Enfin, le dernier ensemble de modifications a trait à la présence d'éléments dans l'interface pour garantir l'ordre des données dans le flux. Les trois ensembles de modifications sont décrits en détail dans les sections suivantes.

4.2.1.1 Nombre et nature des structures utilisées

L'interface de communication que propose l'extension de calcul par flux expose en partie les structures de données utilisées par l'algorithme de communication natif. Or, celui-ci dispose d'un nombre plus important de structures de données que BatchQueue. L'algorithme de communication natif comporte ainsi 5 structures dont les relations entre elles sont repré-

sentées figure 4.6 :

— les vues ;
— les listes de vues ;
— les gestionnaires de vues ;
— les flux ;
— les tâches.

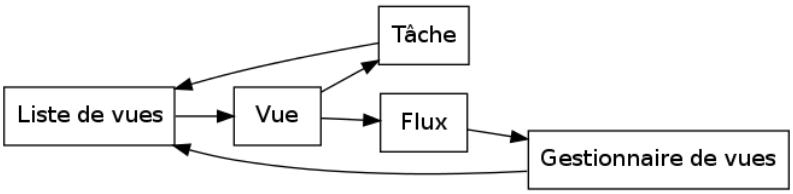


FIGURE 4.6 – Relations entres les structures

Sémantique des structures de données Tout comme pour BatchQueue, 2 structures sont utilisées pour représenter le canal de communication d'une part et les participants – producteurs ou consommateurs – d'autre part : il s'agit respectivement des flux et des vues. L'utilisation de 3 structures supplémentaires pour gérer la communication s'explique alors par deux raisons différentes. Tout d'abord, l'algorithme de communication natif se distingue par sa capacité à gérer plusieurs producteurs et consommateurs. Il faut donc pouvoir maintenir une liste des producteurs et consommateurs participant à la communication : c'est le rôle des listes de vues. De plus, la présence de multiples producteurs et consommateurs augmentant la concurrence d'accès au tampon de communication, une structure supplémentaire est utilisée pour stocker des copies locales des variables partagées, comme expliqué section 3.1.3.3. En ce qui concerne la structure de tâche, elle s'explique par le contexte dans lequel l'algorithme de communication natif est utilisé, à savoir l'extension de calcul par flux pour OpenMP. En effet, l'extension repose sur la création de tâches pour chaque étape du calcul à effectuer. La structure tâche est l'expression directe de cet aspect de l'extension et permet de contrôler le début et la fin de la communication.

Importance des relations entre structures Bien que les structures de données ne soient jamais manipulées en dehors de la bibliothèque dynamique, les adresses de celles-ci sont utilisées par le code généré par le compilateur pour identifier le composant sur lequel une action s'effectue. Par exemple, pour envoyer une donnée il est nécessaire de connaître le canal de communication où la donnée doit être envoyée ainsi que le producteur qui envoie cette donnée. Ces deux informations sont obtenues à partir du pointeur vers la structure « vue » associée à cette production. La structure « vue » enregistre plusieurs informations à propos de l'état du flux ainsi qu'un pointeur vers le canal de communication.

Les adresses sont obtenues elles-mêmes par le code généré en tant que valeur de retour des primitives créant les différentes composants d'une communication. Dans l'exemple précédent, l'adresse de la structure « vue » est obtenue lorsque la primitive créant un producteur

pour un canal de communication donné est appelée. Il importe donc que les adresses retournées par ces primitives pointent vers des structures suffisamment complètes pour que les primitives utilisant ces adresses puissent opérer. Le contenu et le nombre des structures est donc libre mais celles-ci doivent être organisées de manière à ce que l'adresse renvoyée lors de la création d'un producteur pour une communication donnée soit suffisante pour effectuer une production.

Liberté de contenu des structures Cela permet donc à BatchQueue d'utiliser des champs différents de l'algorithme de communication natif en utilisant des structures distinctes mais ayant les mêmes relations et fonctionnalités. Toutes les structures ne sont cependant pas distinctes puisque les structures de tâche, de liste de vues et de gestionnaire de vues utilisées par BatchQueue sont les mêmes que pour l'algorithme natif. L'ensemble de structures en résultant ainsi que leurs relations sont représentées figure 4.7.

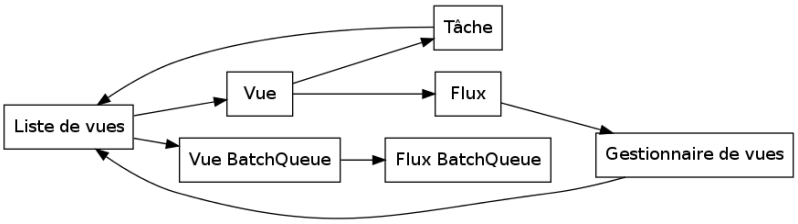


FIGURE 4.7 – Relations entres les structures

4.2.1.2 Transmission sans copie

Interface de BatchQueue Outre un ensemble de structures différents, une autre différence importante entre l'interface de communication native de BatchQueue et l'interface de communication de la bibliothèque dynamique tient dans le déroulement de l'envoi et de la réception. Le déroulement natif de l'interface pour BatchQueue consiste à ne permettre l'envoi et la réception d'une donnée ou d'un ensemble de données que lorsque celles-ci sont déjà prêtes. Le producteur transmet à BatchQueue un tampon depuis lequel les données à envoyer sont copiées dans le canal de communication. Respectivement, le consommateur transmet à BatchQueue un tampon dans lequel les données reçues sont copiées depuis le canal de communication.

Interface de l'algorithme natif L'interface de communication dans la bibliothèque dynamique en revanche sépare l'envoi et la réception chacun en deux étapes. La première étape consiste à mettre à disposition du producteur et du consommateur une partie du tampon servant de canal de communication afin d'y consommer ou produire les données directement. La deuxième étape a pour but la notification de fin d'utilisation de ce tampon. Cette séparation des primitives de communication en deux étapes permet d'éviter une recopie supplémentaire des données entre le canal de communication et les tampons utilisés par le producteur et le consommateurs. Les producteurs et consommateurs produisent et consomment ainsi les données directement dans le tampon.

Adaptation de l'interface de BatchQueue Bien que l'interface native de BatchQueue sépare le tampon du canal de communication des tampons où sont produites et consommées les données, cela n'est dû qu'à l'interface de programmation utilisée. En tant qu'algorithme de communication, BatchQueue gère l'envoi des données et la synchronisation entre le producteur et le consommateur. La façon dont une donnée produite est écrite dans le canal de communication est le domaine de l'interface de communication. Adopter l'interface de communication de la bibliothèque dynamique de l'extension de calcul par flux pour OpenMP est donc possible.

En revanche, la synchronisation opérée par BatchQueue repose sur le fait que le tampon du canal de communication est divisé à tout instant en deux demi-tampons et que ceux-ci sont utilisés entièrement. La production se produit dans l'un des deux demi-tampons tandis que la consommation s'effectue depuis le deuxième. L'étape de mise à disposition de tampon pour le producteur et le consommateur doit donc retourner l'adresse de chaque demi-tampon de façon alternative. De plus il faut s'assurer que les tampons soient entièrement utilisés. Cette contrainte peut également être satisfaite car le nombre de données envoyées et reçues à chaque communication est fixée par une macro. Changer la valeur de cette macro permet alors de satisfaire toutes les hypothèses sur lesquelles repose BatchQueue.

4.2.1.3 Ordre des données

La dernière différence notable entre l'interface de communication native de BatchQueue et l'interface de communication de la bibliothèque dynamique provient de la possibilité pour un nœud de traiter les données dans le désordre. L'interface de communication native de BatchQueue impose que les données soient envoyées dans l'ordre où elles apparaissent dans le flux. Chaque donnée envoyée est considérée comme suivant la donnée envoyée précédemment. À l'opposé, l'interface de communication de la bibliothèque dynamique permet de spécifier la position d'une donnée envoyée dans le flux auquel elle appartient. Ceci permet à un nœud de traiter les données d'un flux dans le désordre, l'algorithme de communication se chargeant de réordonner les données avant de les envoyer.

Cette capacité de l'interface de communication de la bibliothèque dynamique à accepter les données dans le désordre est une conséquence directe de la capacité de l'algorithme natif à gérer plusieurs producteurs et consommateurs. En effet, la présence de plusieurs producteurs ou plusieurs consommateurs nécessite que ceux-ci se synchronisent entre eux. Or une synchronisation est coûteuse en temps et dans le contexte d'un algorithme de communication est susceptible d'impacter lourdement les performances de celui-ci.

Pour limiter le coût de la synchronisation entre producteurs et consommateurs, l'algorithme de communication natif utilise les instructions d'incrément atomique que proposent les processeurs modernes, celles-ci ayant un coût moindre que d'autres formes de synchronisation. Ces instructions sont utilisées pour manipuler un compteur qui détermine la position des données à traiter pour les producteurs et consommateurs. Le fonctionnement du transfert des données par un nœud d'un canal de communication à un autre consiste alors à répéter les actions suivantes :

1. Obtention de la quantité de données disponibles pour consommation n et de la position de la première donnée *start_idx* ;
2. Obtention de l'adresse d'un tampon d'où consommer les données numérotées de *start_idx* à $start\_idx + n$;

3. Notification de fin de consommation ;
4. Obtention de l'adresse d'un tampon où produire les données numérotées de *start_idx* à *start_idx* + *n* ;
5. Notification de fin de production ;

La synchronisation se produit lors de l'étape 1 via un incrément atomique d'un compteur ce qui permet aux nœuds d'exécuter les autres étapes en parallèle.

Contrairement à l'algorithme de communication natif, BatchQueue ne supporte la communication qu'entre un unique producteur et consommateur. Aucun problème de cohérence ne peut alors survenir lors de la production ou la consommation des données : le maintien de l'ordre des données est garanti par la séquentialité du producteur et du consommateur. La position des données produites et consommées est donc une information inutile pour BatchQueue et il est possible de l'ignorer. Chaque donnée produite ou consommée peut être considérée sans risque comme suivant la dernière donnée produite (respectivement consommée) dans le flux de données.

4.2.2 Interchangeabilité des algorithmes de communication

Les adaptations apportées à BatchQueue et présentées ci-dessus permettent à celui-ci d'avoir une interface identique – au nom des fonctions près – à celle de la bibliothèque dynamique. Cependant, les deux algorithmes ont des fonctionnements différents et cette différence de comportement a un impact sur leur utilisation. Le code qui génère les appels aux primitives de communication doit donc être modifié pour prendre en compte ces différences. Deux aspects sont à considérer lors de la génération de code pour chaque algorithme : la possibilité pour celui-ci de gérer plusieurs producteurs ou consommateurs et la taille du tampon de communication nécessaire.

4.2.2.1 Communication à plusieurs producteurs ou consommateurs

Comme détaillé ci-avant, de nombreuses modifications sont nécessaires dans BatchQueue pour que celui-ci puisse être utilisé en lieu et place de l'algorithme de communication natif. Un nombre important d'entre elles sont dû à la capacité additionnelle que possède l'algorithme de communication natif à permettre la communication entre plusieurs producteurs et plusieurs consommateurs et qui se reflète au niveau de l'interface de communication. Or cette capacité découle de la possibilité qu'offrent les annotations de l'extension de créer des tâches ayant plusieurs entrées et/ou sorties. Il est donc nécessaire de pouvoir offrir une communication entre plusieurs producteurs et consommateurs pour que l'ensemble des programmes utilisant les annotations de cette extension puissent fonctionner.

BatchQueue ne proposant une communication qu'entre un unique producteur et un unique consommateur, il ne peut donc être utilisé inconditionnellement pour gérer toutes la communication nécessaire dans un programme utilisant le parallélisme de flux. Il est en revanche possible de l'utiliser dans la majorité des cas où la communication n'implique qu'un seul producteur et consommateur et déléguer les quelques cas restant à l'algorithme de communication natif. Toutes les applications continuent ainsi de fonctionner tout en bénéficiant de performances améliorées pour la majorité des communications ayant lieu entre deux nœuds uniquement.

Pour obtenir ce fonctionnement transparent, il est nécessaire de disposer dans la bibliothèque dynamique à la fois les primitives de BatchQueue et celles de l'algorithme de communication natif. Pour y parvenir, les primitives de BatchQueue doivent être renommées pour ne pas entrer en conflit avec celles de l'algorithme de communication natif. Ce faisant, les primitives de BatchQueue cessent de respecter l'interface de programmation de la bibliothèque dynamique. Une logique est donc ajoutée dans la génération de code afin de faire appel aux primitives de BatchQueue plutôt que celles de l'algorithme natif s'il s'agit d'une communication avec un unique producteur et consommateur. Ceci est fait juste avant que la création des flux ne soit effectuée en comptant le nombre de vues en lecture et écriture. Si le nombre total de vues est deux, alors BatchQueue peut gérer la communication pour ce flux et le reste du code généré pour ce flux contient des appels aux primitives de BatchQueue. Sinon, les primitives utilisées sont celle de l'algorithme de communication natif.

4.2.2.2 Taille des tampons de communication

Pour un algorithme de communication, le choix de la taille du tampon de communication relève d'un compromis entre fréquence des synchronisations et consommation mémoire. Plus un tampon de communication est grand, moins fréquente est la synchronisation entre le producteur et le consommateur. À l'inverse, plus un tampon est petit, plus la consommation mémoire est réduite. Chaque taille de tampon présentant des avantages et des inconvénients, il est donc peu surprenant que différents algorithmes utilisent des tailles de tampon différentes.

Le compromis choisi pour BatchQueue est en faveur de la consommation mémoire. La raison derrière ce choix est que pour une taille de tampon donné, BatchQueue ne nécessite que peu de synchronisations. Ainsi de bonnes performances sont obtenus avec un tampon relativement petit. À l'opposé, l'algorithme de communication natif utilise un tampon de communication de grande taille. Ce choix est une conséquence de la capacité de l'algorithme de communication natif de supporter plusieurs producteurs ou consommateurs. En effet, le tampon de communication doit être assez grand pour que plusieurs producteurs y produisent en même temps.

Bien que l'utilisation d'un tampon de communication de grande taille soit requise lorsqu'une communication implique de nombreux producteurs et consommateurs, cela n'est pas le cas lorsqu'un unique producteur et consommateurs communiquent. Il est donc possible de choisir une taille de tampon qui dépende du nombre de producteurs et consommateurs impliqués : moins il y a de participants, plus la taille du tampon peut être réduite. Cette flexibilité permet de profiter d'une limitation de la consommation mémoire dans le cas général tout en permettant le bon fonctionnement de l'algorithme de communication natif lorsque celui-ci doit être utilisé.

Ajouter le support pour cette flexibilité correspond alors au deuxième et dernier changement nécessaire dans le code générant les appels aux primitives de la bibliothèque dynamique. L'ajout du support consiste alors à réutiliser la détection du nombre de participants ajoutée pour utiliser l'algorithme de communication natif en cas de multiples producteurs ou consommateurs. Une fois le nombre de participants connu, il est ainsi possible de configurer la taille du tampon en conséquence.

De nombreux changements [3] sont donc nécessaires pour pouvoir utiliser BatchQueue avec l'extension de calcul par flux. Les changements sont nécessaires à la fois dans BatchQueue et dans la génération de code. Dans BatchQueue, les changements sont nécessaires pour qu'il adopte la même interface que l'algorithme de communication natif, ceci afin de minimiser les changements requis dans la génération. Conserver une interface différente pour BatchQueue demanderait une logique distincte de génération spécifique pour les deux algorithmes alors que l'utilisation d'une interface commune permet de limiter les modifications à l'ajout d'alternatives pour choisir le nom des fonctions à appeler. Les changements dans la génération de code ne se bornent cependant pas à quelques alternatives puisqu'il est nécessaire de détecter si une communication donnée implique plusieurs producteurs ou plusieurs consommateurs, indiquant que BatchQueue ne peut être utilisé et que la taille du tampon de communication doit être agrandie. In fine, bien que nombreux, les changements pour utiliser BatchQueue en lieu et place de l'algorithme de communication natif sont transparents pour les applications. De plus, ceux-ci sont principalement localisés dans la bibliothèque dynamique, laissant ainsi la partie génération de code peu modifiée.

4.3 Évaluation préliminaire

La présentation et l'évaluation détaillée de BatchQueue faite dans le chapitre 3 montrent que sa conception lui permet d'offrir des performances améliorées par rapport à l'algorithme de communication natif de l'extension de calcul par flux pour OpenMP. BatchQueue offre ainsi un débit près de deux fois supérieur à celui de l'algorithme de communication natif. Ce résultat suggère que les performances du parallélisme de flux tel que proposé par l'extension de calcul par flux pour OpenMP peuvent être améliorées en utilisant BatchQueue.

Cependant, l'évaluation effectuée dans la section 3.3 n'étudie les performances de la communication qu'entre deux unités d'exécution. Or, dans le cadre du parallélisme de flux, la communication peut intervenir entre un nombre bien plus grand de cœurs, ceux-ci étant chaînés entre eux et le chaînage de plusieurs canaux de communication crée des interactions supplémentaires qui peuvent impacter les performances de manière importante. Pour les nœuds en milieu de chaîne la situation est différente de celle évaluée puisqu'ils sont impliqués dans deux communications différentes : une fois en tant que producteur et une fois en tant que consommateur. Le blocage de l'un d'entre eux pour attendre des données ralentit alors les nœuds en aval dans la chaîne par effet de cascade. La performance de BatchQueue dans un tel contexte peut donc être différente.

Communication chaînée avec BatchQueue

Afin d'évaluer l'intérêt de l'utilisation de BatchQueue dans une configuration où plusieurs nœuds sont chaînés, cette section présente la réalisation d'un test en environnement contrôlé similaire à celui effectué pour évaluer BatchQueue dans une configuration à deux nœuds. Comme lors de l'évaluation initiale de BatchQueue, le test consiste à envoyer près de 3 Gio de données et de mesurer le temps mis pour envoyer toutes les données pour en déduire le débit. Le changement provient du fait que 4 nœuds sont utilisés afin de reproduire la structure d'un flux de communication tel qu'il se produit dans le cas du parallélisme de

3. L'ensemble des modifications ainsi effectuées est disponible en ligne [Git].

flux. Les résultats de l'exécution de ces deux tests sur la machine bossa, décrites dans la section 4.3 sont présentés dans la figure 4.8.

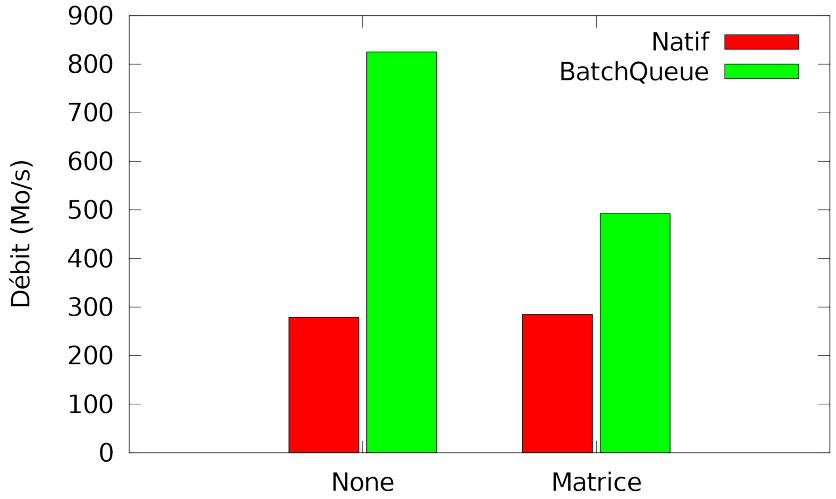


FIGURE 4.8 – Débit soutenu par les deux algorithmes pour une chaîne de communication de 4 nœuds

Les résultats montrent un fort avantage à utiliser BatchQueue : BatchQueue offre un débit 3 fois plus important lorsque la communication est intensive et 1.7 fois plus important lorsque le calcul est intensif. Le débit semble moins améliorée dans le cas du calcul matriciel car celui-ci est calculé à partir du temps global d'exécution du test. Or le calcul matriciel représente une grande partie de ce temps et ne dépend pas du type de communication utilisée. L'amélioration du débit est donc plus importante que le calcul ne l'indique. Il en résulte que BatchQueue est peu sensible à une utilisation importante du cache de premier niveau.

Il ressort donc de ce test en environnement contrôlé que les bons résultats de BatchQueue en terme de débit sont conservés lorsque celui-ci est utilisé dans une chaîne de nœuds. Comme dans la communication entre deux cœurs uniquement, BatchQueue présente un bon débit, que le cache de premier niveau soit fort utilisé ou non. Cela confirme donc l'intérêt d'utiliser BatchQueue comme algorithme de communication dans le cadre du parallélisme de flux.

4.4 Performances appliquées

Les résultats du test en environnement contrôlé confirment qu'y compris dans une configuration avec une chaîne de nœuds, BatchQueue propose un meilleur débit que l'algorithme natif. Par conséquent, de meilleures performances peuvent être attendues en utilisant l'extension de calcul par flux pour OpenMP si BatchQueue est utilisé pour la communication au lieu de l'algorithme de communication natif. Cependant, plusieurs modifications sont nécessaires pour intégrer BatchQueue à l'extension de calcul par flux pour OpenMP et chacune d'entre elles est susceptible de changer le comportement de BatchQueue par la même occasion. De même, l'utilisation conjointe de BatchQueue et de l'algorithme de communica-

tion natif lorsque le flux de communication n'est que partiellement linéaire peut créer des intéractions ayant des effets sur les performances.

Pour confirmer les performances de l'extension de calcul par flux en utilisant BatchQueue comme algorithme de communication, trois programmes sont annotés avec des pragmas de l'extension. Les programmes ainsi annotés sont alors compilés une fois avec l'extension telle que proposée par les auteurs et une fois avec BatchQueue intégré à celle-ci. L'accélération obtenue avec chacun des algorithmes est alors comparée : « Natif » dénote la version des programmes utilisant uniquement l'algorithme de communication natif tandis que « BatchQueue » correspond à la version des programmes utilisant BatchQueue pour les communications ayant un unique producteur et consommateur. Toutes les applications étudiées ont été exécutées sur la même machine, « bossa », décrite dans la section 4.3.

4.4.1 FMradio : une synchronisation excessive

La première application concrète évaluée est FMradio, un programme extrait du projet GNU radio puis modifié pour être parallélisé. FMradio est choisi car il fait partie des trois applications utilisées pour l'évaluation de l'extension de calcul par flux pour OpenMP dans [PC11]. Les deux autres applications sont un programme de transformée de Fourier rapide et un programme mettant en œuvre la norme 802.11a utilisé en production par Nokia. Ces applications ne sont pas évaluées car présentent des difficultés de reproductivité : la parallélisation de la transformée de Fourier rapide nécessite un réglage manuel du code de l'application et le programme mettant en œuvre la norme 802.11a n'est pas présenté dans l'article.

Deux versions annotées de FMradio ont été effectuées par Pop et Cohen dans leur évaluation de l'extension de calcul par flux pour OpenMP : une version avec uniquement du parallélisme de flux et une version combinant du parallélisme de flux et du parallélisme de donnée. La version avec uniquement du parallélisme de flux est utilisée pour l'évaluation car c'est l'impact de BatchQueue sur le parallélisme de flux qui est évalué. De plus, comme pour la transformée de Fourier rapide, la version combinant les deux formes de parallélisme ne peut pas être parallélisée de façon entièrement automatique. Le nombre de cœurs de calcul utilisé est fixé à 12 car les annotations ajoutés par les auteurs de [PC11] ne permettent pas de faire varier le nombre de cœurs utilisés. Les résultats sont présentés dans la figure 4.9.

En dépit des bonnes performances de BatchQueue dans le test en environnement contrôlé, l'accélération est la même quel que soit l'algorithme de communication utilisé dans le test avec FMradio. Étant donné l'extrême similarité des résultats, ce résultat suggère que la communication n'est pas le facteur limitant pour cette application. Deux explications sont alors possibles. La première est que le calcul reste le facteur limitant bien que celui-ci soit réparti sur douze cœurs de calcul. La seconde explication est que la parallélisation mène à une synchronisation excessive entre les différents cœurs de calcul utilisés.

Pour mieux identifier le problème, nous avons instrumentalisé le code de façon à obtenir le nombre de tâches créées ainsi que les échanges de données entre ces tâches. Le graphe du flux des données obtenu à partir de ces mesures est présenté figure 4.10. Cette analyse montre que la chaîne que forment les nœuds n'est pas linéaire, celle-ci contient au contraire plusieurs branches. Or, un canal de communication impliquant plusieurs producteurs ou plusieurs consommateurs, tel le canal C12, nécessite de la synchronisation entre les participants et est ainsi susceptible d'être un goulet d'étranglement.

Une analyse étendue, incluant le temps passé en attente active par chaque nœud et pour

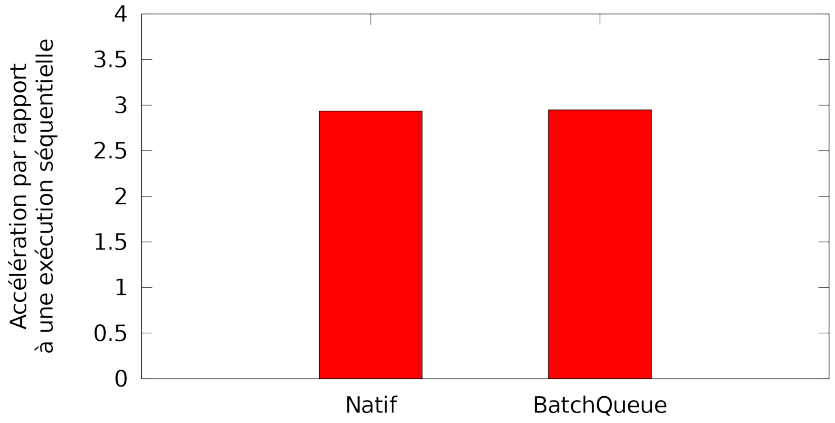


FIGURE 4.9 – Accélération obtenue par les deux configurations avec l'application FMradio

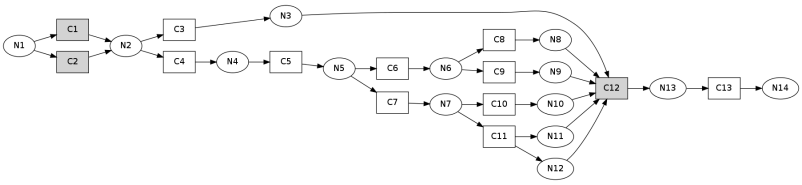


FIGURE 4.10 – Structure du flux de données avec FMradio

chaque flux, montre que le problème est en effet structurel mais réside principalement dans les canaux de communication C1 et C2. Cette analyse montre que le temps passé en attente active est très déséquilibré entre les canaux C1 et C2 : il apparaît que l'un des flux de données qui transite par ces canaux attend l'autre.

En définitive, FMradio présente deux limites qui l'empêche de passer à l'échelle lorsque le parallélisme de flux y est mis en œuvre, quel que soit l'algorithme de communication utilisé. La première limite est la synchronisation excessive dont fait l'objet un canal de communication, faisant ainsi office de goulet d'étranglement. La deuxième limite provient du déséquilibre entre deux canaux de communication reliant les deux même nœuds. En définitive, FMradio ne passe pas à l'échelle avec le parallélisme de flux car le chaînage des flux contient des branches et ne se conforme donc pas à la structure linéaire que le parallélisme de flux gère efficacement.

4.4.2 Décodage par treillis : jusqu'à 200% d'amélioration

L'absence d'amélioration des performances dans le cas de FMradio montre que le passage à l'échelle du nombre de cœurs ne peut pas toujours être atteint, à cause de limites intrinsèques au programme à paralléliser. En particulier, des chaînes de nœuds non linéaires ont tendance à présenter de piètres résultats pour cause de synchronisation excessive et de branches non équilibrées. En conséquence, les expériences présentés ci-après – décodage par treillis et modèle de code adapté au parallélisme de flux – ne considère que les programmes dont les dépendances sont linéaires.

Le décodage par treillis présenté dans cette section provient d'un travail mené à Alcatel

Lucent [MLKD10, HHK+10]. Le but du programme original est de décoder avec détection d'erreurs un signal correspondant à une entête de paquet AAC transmis via un canal de communication non fiable. Pour détecter une erreur dans un paquet, le programme calcule la somme de contrôle de l'entête et la compare à celle présente dans le paquet. Le programme présenté ici est une réécriture en C [4] du calcul de la somme de contrôle des paquets via l'utilisation d'un treillis.

Le calcul de la somme de contrôle d'une entête AAC se présente en deux parties. La première partie consiste à analyser les portions du signal reçu correspondant à chaque bit et à déterminer pour chaque portion à partir de l'intensité de ces portions de signal la probabilité que le bit envoyée était un 1. Le décodage par treillis correspond à la deuxième partie du calcul. Cette partie tire son nom du graphe de type treillis [5] utilisé pour calculer la somme de contrôle.

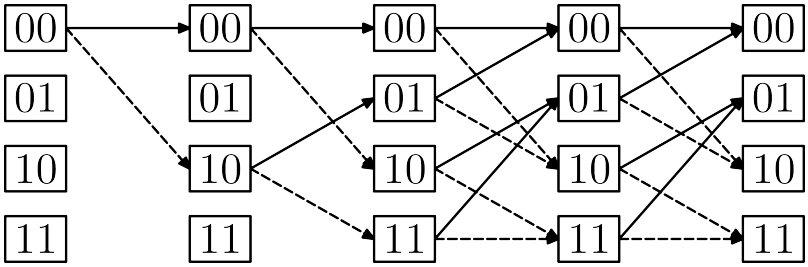


FIGURE 4.11 – Treillis permettant le calcul des sommes de contrôle

Dans le cadre présent, les colonnes du treillis contiennent un nœud pour chaque valeur possible de la somme de contrôle et les arcs représente les changements dans la somme de contrôle lorsqu'un nouveau bit est analysé. Le treillis contient donc autant de colonnes que de bits dans une entête AAC et chaque nœud possède exactement 2 arcs sortants, un pour chaque valeur possible du nouveau bit analysé. Un exemple d'un tel treillis est présenté figure 4.11. Le processus consiste alors à calculer la probabilité de chaque valeur possible de la somme de contrôle au fur et à mesure que les probabilités des bits d'entête qui ont été calculées précédemment sont analysées. La valeur de la somme de contrôle est alors obtenue en choisissant celle avec la plus grande probabilité.

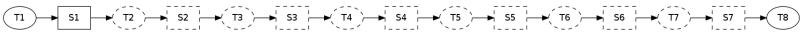


FIGURE 4.12 – Structure du flux de données avec le décodage de treillis

À l'opposé de FMradio, la structure de l'algorithme de décodage par treillis est typique de ce que le parallélisme de flux gère efficacement. Le flux de données subissant un traitement séquentiel et dont les dépendances entre les différentes étapes sont simples. Il en résulte qu'une fois parallélisé avec l'extension de calcul par flux pour OpenMP, le programme s'exécute sur un ensemble variable d'unités d'exécution dont le flux de communication

4. Le programme original est écrit en C++ et l'extension de calcul par flux pour OpenMP ne supporte que le language C

5. Pour rappel, un treillis est un graphe orienté dont les nœuds sont organisés en un ensemble de colonnes, chaque nœud étant relié à au moins un nœud de la colonne précédente et un nœud de la colonne suivante.

forme une chaîne, tel montré dans la figure 4.12. Il est à noter cependant que chaque paquet est traité indépendamment des autres. Aussi, la même technique que celle utilisée pour traiter les paquets réseau dans le patch RPS pour Linux [Rec] – Receive Packet Steering (Pilotage de la réception de paquets) – pourrait être utilisée à la place du parallélisme de flux. Les ensembles de probabilités seraient alors envoyés selon l'algorithme du tourniquet sur les différentes unités de calcul pour y être entièrement traité, évitant ainsi le surcoût lié à la communication.

La parallélisation du programme est effectuée de telle façon que chaque cœur de calcul gère une partie du calcul de la somme de contrôle de l'entête AAC et transmette le résultat au cœur suivant pour traiter les bits suivants. Ainsi, le traitement séquentiel peut être découpé en autant d'étapes qu'il y a de probabilités dans un ensemble, permettant d'observer le passage à l'échelle du décodage par treillis lorsque du parallélisme de flux est mis en œuvre. Le résultat est présenté dans la figure 4.13 où chaque histogramme représente l'accélération obtenue par rapport à l'exécution du code en séquentiel. Dès lors, les meilleurs résultats sont représentés par les histogrammes les plus grands.

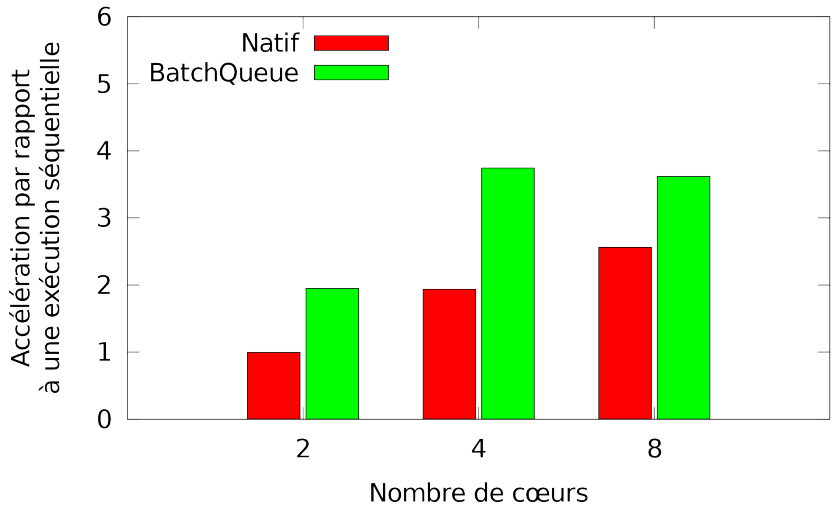


FIGURE 4.13 – Accélération obtenue par les deux configurations avec le décodage par treillis

Contrairement à FMradio, les performances du décodage par treillis sont améliorées lorsque BatchQueue est utilisée au lieu de l'algorithme de communication natif. L'accélération du programme une fois parallélisé par l'extension de calcul par flux pour OpenMP est améliorée d'un facteur 2 lorsque BatchQueue est utilisé et que le nombre de cœurs utilisés est inférieur à 8. De plus, l'utilisation de BatchQueue rend l'accélération presque linéaire : l'accélération est de 1,95 pour 2 cœurs de calcul, et de 3,75 lorsque 4 cœurs sont utilisés. En revanche, lorsque 8 cœurs sont utilisés, l'accélération avec BatchQueue est légèrement réduite pour atteindre 3,62 tandis que l'accélération avec l'algorithme natif s'améliore un peu et atteint 2,56.

La limite atteinte par BatchQueue, et dans une moindre mesure par l'algorithme de communication natif, s'explique par le rapport entre temps de communication et le temps nécessaire pour exécuter une étape du calcul. En effet, chaque cœur doit transmettre au suivant le résultat de son calcul, c'est à dire les probabilités obtenues pour chaque valeur possible

de somme de contrôle étant donné les probabilités des bits analysés jusque là. La somme de contrôle utilisée contenant 8 bits, le nombre de valeurs envoyées après chaque étape du calcul de la somme de contrôle est de 2^8. L'algorithme de communication natif n'est quant à lui pas aussi pénalisé car le surcroît de communication réduit la quantité d'attente active que celui-ci exécute. Cela s'observe en particulier au niveau du nombre de branchements effectués : celui-ci reste à peu près stable entre 4 et 8 cœurs alors qu'il est multiplié par 2,5 pour BatchQueue.

4.4.3 Modèle de code : accélération multipliée par 2

L'exemple du décodage par treillis est prometteur : il montre qu'une amélioration de l'accélération est possible en utilisant BatchQueue. De plus, l'évolution de l'accélération en fonction du nombre de cœurs de calcul montre qu'une accélération presque linéaire est possible si le programme parallélisé s'y prête bien. Cependant, le décodage par treillis ne présente pas de dépendances de données entre les paquets du flux : chaque paquet est traité indépendamment. Il est donc possible de paralléliser le traitement autrement en analysant chaque paquet sur des cœurs de calcul différents, à la manière de ce qui est fait dans Linux pour traiter les paquets [Rec].

La dernière expérience, présentée ci-dessous, concerne l'utilisation d'un modèle de code ayant toutes les caractéristiques pour profiter au mieux du parallélisme de flux, notamment des dépendances de données. L'utilisation d'un modèle de code permet d'obtenir la meilleure accélération pouvant être obtenue avec BatchQueue lorsqu'il est utilisé au sein de l'extension de calcul par flux pour OpenMP. Cela aide également les programmeurs qui considèrent la mise en œuvre du parallélisme de flux à estimer l'accélération qu'ils peuvent espérer pour leur propre programme, en fonction de la proximité de sa structure à celle du modèle de code.

Il y a une catégorie de logiciels dont la structure est parfaitement adaptée au parallélisme de flux : le traitement de flux audio et vidéo. Comme expliqué dans [GTA06], le traitement audio et vidéo est organisé sous la forme d'un graphe de filtres : les données évoluent d'un filtre à un autre en étant transformées à chaque étape. Les filtres peuvent être à état, ce qui signifie que leur résultat peut dépendre de la donnée en train d'être manipulée et des données précédemment traitées. Cette chaîne de dépendances implique que même la technique de pilotage de paquets – paquet steering – ne peut être utilisée pour paralléliser ce code, seul le parallélisme de flux le peut.

Cependant, paralléliser un tel code nécessite une analyse précise de celui-ci afin de déterminer où placer les annotations dans le code et quelles annotations utiliser. C'est pourquoi, le dernier exemple présenté fait appel à un modèle de code écrit pour l'occasion et inspiré de la structure des logiciels de traitement de flux existants plutôt qu'à un programme de traitement de flux complet qui aurait été parallélisé. Le modèle de code présente les même contraintes que les logiciels dont il s'inspire à savoir le traitement séquentiel d'un flux de données avec une chaîne linéaire de dépendances. Comme dans le cas du décodage par treillis, la chaîne de nœuds résultante est également complètement linéaire. Le traitement séquentiel effectué possède en outre la capacité d'être divisé en autant d'étapes que nécessaire, assurant au code la possibilité de passer à l'échelle du nombre de cœurs avec le parallélisme de flux. Le résultat peut être ainsi comparé à celui du décodage par treillis. La figure 4.14 présente les résultats de la parallélisation du modèle de code.

Deux points positifs apparaissent à la lecture de ces résultats : (i) une amélioration par

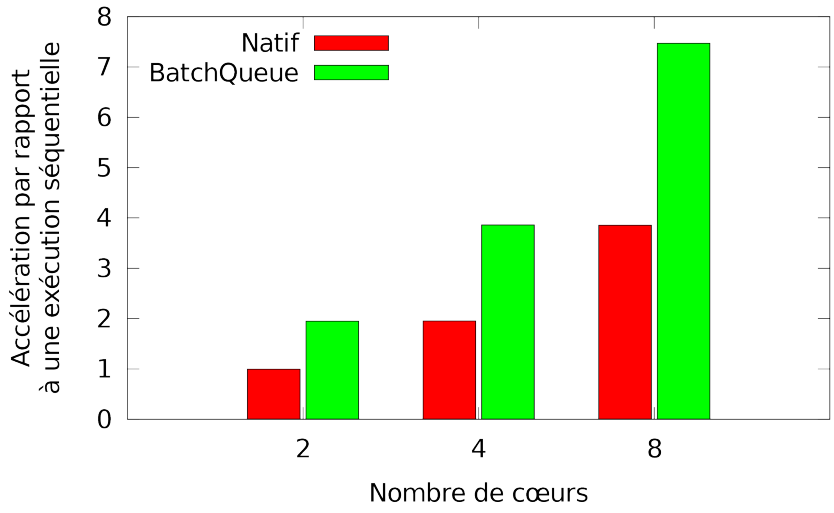


FIGURE 4.14 – Accélération obtenue par les deux configurations avec le modèle de code

deux des performances lorsque BatchQueue est utilisé pour la communication et (ii) un bon passage à l'échelle de l'application parallélisée. Cette forte amélioration des performances lorsque Batchqueue est utilisé confirme une fois de plus l'intérêt d'utiliser un algorithme de communication optimisé pour la communication entre un unique producteur et consommateur et situé sur des cœurs différents. La réduction de la synchronisation qui en résulte permet d'améliorer significativement les performances. De plus, les performances obtenues sont alors relativement linéaires vis à vis du nombre de cœurs : l'accélération obtenue en utilisant BatchQueue pour 8 cœurs de calcul est de 5,95.

Pour confirmer le passage à l'échelle de l'application parallélisée, l'expérience a également été menée sur une machine possédant un nombre de cœurs de calcul plus important : la machine *quadhexa*. Cette machine contient quatre processeurs Xeon X7460 ayant six cœurs de calcul cadencés à 2,66 GHz et une mémoire vive de 126 Gio. Les cœurs de calcul ont une mémoire cache de premier niveau de 32 Kio, une mémoire cache de second niveau de 3 Mio partagée par paire de cœurs et une mémoire cache de troisième niveau de 16 Mio partagée entre tous les cœurs d'un processeur. Le système d'exploitation est Mageia 3 « Cauldron » installé en 64 bits, tant pour le noyau Linux 3.8.1 que l'espace utilisateur. Les résultats obtenus avec cette machine sont présentés dans la figure 4.15.

Les résultats obtenus avec cette machine viennent confirmer ceux obtenus sur la machine *bossa* : les performances sont améliorées d'un facteur 2 lorsque BatchQueue est utilisé pour la communication et l'application passe à l'échelle du nombre de cœurs de façon surlinéaire. Ce passage à l'échelle de façon surlinéaire provient d'une particularité du processeur qui fait que les petites boucles de calcul sont exécutées de façon plus efficace que les boucles de calcul plus importante. Ceci a pour effet que l'exécution séquentielle est plus lente que deux fois l'exécution parallèle sur deux cœurs de calcul. Cet effet ne survient que lors du passage de un à deux cœurs de calcul : l'accélération est linéaire à partir de deux cœurs de calcul utilisés. Il apparaît donc que BatchQueue parvient à améliorer considérablement le passage à l'échelle vis à vis du nombre de cœurs d'un programme lorsque le traitement séquentiel effectué par celui-ci peut être divisé en un nombre arbitraire d'étapes.

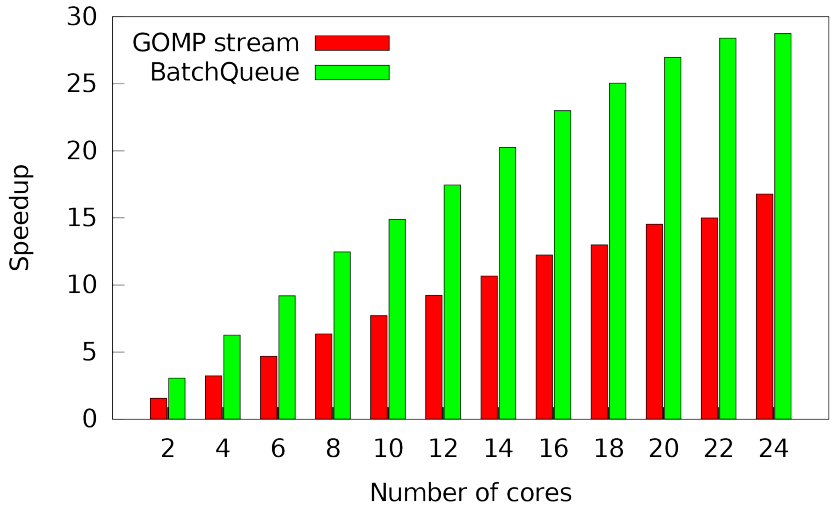


FIGURE 4.15 – Accélération obtenue par les deux configurations avec le modèle de code sur la machine *quadhexa*

Comme le démontrent les différents travaux sur le parallélisme de flux [PC11, GTA06], cette forme de parallélisme offre la possibilité de paralléliser de manière efficace des programmes qui ne peuvent être parallélisés par les techniques plus répandues que sont le parallélisme de tâche et le parallélisme de donnée. Cependant, pour passer correctement à l'échelle vis à vis du nombre d'unités de calcul utilisées, le parallélisme de flux requiert un algorithme de communication très efficace. Il ressort de l'évaluation présentée dans ce chapitre que BatchQueue peut être cet algorithme, permettant ainsi aux applications gérant un flux avec de fortes dépendances entre les données de prendre avantage du nombre croissant de cœurs de calcul dans les processeurs actuels.

Il n'en reste pas moins que pour profiter de cette amélioration du passage à l'échelle, les programmes mettant doivent être adaptés pour utiliser BatchQueue comme algorithme de communication dans la mise en œuvre du parallélisme de flux. À ce titre, ce chapitre présente également une intégration de BatchQueue au sein de l'extension de calcul par flux pour OpenMP. Cette intégration simplifie ainsi la mise en œuvre de cette forme de parallélisme avec BatchQueue comme algorithme de communication. Pour les programmes non parallélisés, cette intégration permet d'utiliser l'extension de calcul par flux pour OpenMP afin de simplifier leur parallélisation. Pour les programmes déjà parallélisés en utilisant cette extension, l'intégration de Batchqueue permet d'en profiter en effectuant une simple recompilation. La solution présentée dans ce chapitre permet donc de paralléliser simplement et efficacement les programmes de traitement de flux ayant des dépendances entre les données.

Chapitre 5

Conclusion

Sommaire

5.1 Synthèse

AUJOURD'HUI, avec l'augmentation du nombre de cœurs, les processeurs ressemblent de plus en plus à des systèmes répartis. La communication entre unités de calcul joue ainsi un rôle essentiel dans l'efficacité des programmes parallèles s'exécutant sur ces nouvelles architectures. Elle représente un des principaux goulots d'étranglement pour les applications parallèles.

Ceci est d'autant plus vrai dans le cas du parallélisme de flux, qui se distingue des autres paradigmes de parallélisation (parallélisme de tâche, de donnée, etc.) par son flux de données continu important entre les cœurs. Or, nombre des programmes mettant en œuvre le parallélisme de flux n'emploient pas d'algorithme de communication suffisamment efficace en terme de débit.

Une modification manuelle du mécanisme de communication de l'ensemble des applications parallèles parait peu envisageable. Cependant, de nombreuses applications parallèles reposent sur des outils de parallélisation automatique. Il suffit alors de modifier le code de communication généré par ces outils pour améliorer les performances d'un grand nombre d'applications patrimoniales. Parmi ces outils, nous nous sommes intéressés à l'extension de calcul par flux pour OpenMP.

Cette thèse apporte deux contributions.

Nous avons, dans un premier temps, proposé et évalué BatchQueue, un nouvel algorithme de communication inter-cœurs. Ce dernier s'est montré, lors des expérimentations, particulièrement efficace en comparaison des algorithmes existants dans la littérature.

Nous avons également amélioré l'extension de calcul par flux pour OpenMP afin d'utiliser BatchQueue pour réaliser les communications linéaires. Les résultats expérimentaux ont montré qu'on pouvait ainsi améliorer sensiblement les performances d'applications parallèles sans en modifier le code, prouvant ainsi la validité de cette approche.

Des enseignements sont à tirer de chacune de ces contributions. Le premier enseignement est l'importance que revêt le cache matériel dans les performances d'un algorithme de communication sur système multi-cœurs à mémoire partagée. En effet, en prenant mieux en compte les coûts associés à l'utilisation du cache matériel, l'algorithme BatchQueue parvient à obtenir un débit jusqu'à 2 fois supérieur à celui des algorithmes de communication existants.

Plusieurs aspects ont ainsi été améliorés concernant la prise en compte du cache matériel. Pour commencer, BatchQueue diminue le nombre de synchronisations nécessaires entre producteur et consommateur. Une synchronisation n'est effectuée que lorsqu'un nombre important de données sont prêtes à être envoyées et celle-ci ne nécessite que deux écritures pour être réalisée. Ce mécanisme permet ainsi de limiter le coût lié à la mise en cohérence des caches. Ensuite, BatchQueue met l'accent sur la limitation de son empreinte mémoire. Celui-ci n'utilise par défaut que 64 lignes de cache pour ses deux tampons de communication et seul un bit supplémentaire est nécessaire pour effectuer la synchronisation entre producteur et consommateur. De cette façon une plus grande partie du cache de premier niveau reste disponible pour l'application utilisant cet algorithme. Enfin, BatchQueue cherche à minimiser les effets négatifs que peut entraîner le préchargement matériel. Ceci est accompli en rendant le bit de synchronisation non contigu des tampons de communication et en accédant aux tampons au travers d'adresses virtuelles différentes. Ces dispositions permettent d'éviter le faux partage résultant du chargement prématuré d'une ligne de cache encore en train d'être modifiée.

Le second enseignement à tirer de cette thèse concerne l'importance de l'efficacité de l'algorithme de communication utilisé pour mettre en œuvre le parallélisme de flux. En effet, avec cette forme de parallélisme l'accélération obtenue est bornée par le temps de communication entre unités de calcul. Dès lors, l'utilisation intelligente d'algorithmes de communication optimisés, tel BatchQueue, permet d'améliorer de façon significative les performances des applications, comme l'a montré l'évaluation présentée chapitre 4.4.

La création d'un algorithme plus efficace est possible car l'algorithme de communication natif est conçu pour gérer un modèle plus général de communication : la communication entre plusieurs producteurs et plusieurs consommateurs. Un tel modèle est également plus complexe et se résout moins efficacement. En particulier, ce modèle implique plus de participants et demande donc plus de synchronisation. Pour cette raison, le parallélisme de flux est mis en œuvre de façon à obtenir des flux de données aussi linéaires que possible. En conséquence, la majeure partie des communications ayant lieu dans les programmes suivant le parallélisme de flux n'impliquent qu'un producteur et un consommateur. La communication est donc améliorée lorsque l'algorithme de communication plus spécialisé qu'est BatchQueue est utilisé pour toutes les communications linéaires. Cette approche permet alors d'améliorer le débit de traitement des données significativement. De plus, en choisissant dynamiquement l'algorithme de communication suivant le modèle de communication rencontré, il est possible de gérer autant de cas différents que l'algorithme natif tout en bénéficiant d'amélioration des performances.

La solution présentée dans cette thèse remplit l'objectif fixé d'améliorer les performances des programmes mettant en œuvre le parallélisme de flux par le biais de l'extension de calcul par flux pour OpenMP. Cette thèse propose ainsi un nouvel algorithme de communication améliorant le débit de communication d'un facteur 2 par rapport à l'état de l'art pour certaines configurations. De plus, son intégration au sein de l'extension de calcul par flux pour OpenMP permet d'en faire profiter facilement de nombreux programmes via une sélection automatique de l'algorithme de communication selon les caractéristiques du flux rencontré.

5.2 Perspectives

Les résultats obtenus sont encourageants et incitent à explorer plus avant les axes de recherche développés dans cette thèse. Plusieurs pistes sont envisagées pour perfectionner BatchQueue et améliorer ainsi les performances d'un plus grand nombre de programmes :

— amélioration de l'interaction avec l'ordonnanceur du système

— réduction du coût de synchronisation.

— réduction du nombre de recopie des données

— gestion de plusieurs producteurs et consommateurs

5.2.1 Perspectives à court terme

5.2.1.1 Interaction avec l'ordonnanceur du système

Dans une communication de type producteur/consommateur, un mécanisme de synchronisation est nécessaire pour synchroniser la production et la consommation dès lors qu'aucun signal d'horloge ne permet de rythmer ces deux activités. Ce mécanisme est nécessaire pour gérer le remplissage du tampon de communication partagé entre le producteur et le consommateur : la production doit être bloquée lorsque le tampon est plein et respectivement la consommation doit être bloquée lorsque le tampon est vide.

Des primitives de synchronisation sont fournies par le système afin de mettre en place ce type de synchronisation. Elles permettent de suspendre des processus tant qu'une condition n'est pas vérifiée et de notifier des processus quand celle-ci est vérifiée. L'utilisation de ces primitives permet de libérer les processeurs des processus mis en attente tant que la condition n'est pas remplie. Les processeurs peuvent ainsi être mis à profit par d'autres processus. Cependant, ces primitives système sont lentes car elles requièrent un appel système pour que l'ordonnanceur puisse libérer le processeur des processus mis en attente et en ordonnancer d'autres à la place. C'est pourquoi BatchQueue, visant une communication efficace, utilise à la place un système d'attente active.

L'utilisation d'attente active pour attendre qu'une condition soit vérifiée implique la perte de cycles processeurs. Par principe, le processus ou processus léger mis en attente continue de s'exécuter, empêchant par la même tout autre processus de s'exécuter dans l'intervalle. Ceci ne constitue pas en soit un problème si l'attente dure très peu de temps car les primitives permettant de relâcher le processeur ont un coût en instruction plus élevé. Par contre, dans le cas d'une attente prolongée, le coût de l'attente active devient plus important. Ceci rend BatchQueue inapproprié dans deux cas d'utilisation : (i) dans le cas des systèmes où l'énergie disponible est restreinte, tels les systèmes embarqué, et (ii) dans le

cas d'un nœud participant à plusieurs communications avec le même rôle, producteur ou consommateur.

L'inadaptation de BatchQueue à ces deux cas d'utilisation provient de ce que le processus effectuant l'attente est considéré comme actif par l'ordonnanceur du système. En conséquence, l'ordonnanceur ne prend aucune mesure pour réduire la consommation de l'unité de calcul sur laquelle le processus s'exécute, tel réduire sa fréquence d'horloge. De même, l'ordonnanceur ne peut remplacer le processus par un autre processus, en particulier un autre processus communiquant. Les autres types de processus souffrent moins du problème car ils ont la possibilité de s'exécuter sur un autre processeur. Afin de remédier à cela, il est nécessaire d'invoquer l'une des primitives permettant de relâcher le processeur en cas d'attente prolongée mais pas en cas d'attente courte sous peine de réduire les performances. Une heuristique doit donc être trouvée qui permette de détecter avec une bonne précision si une attente sera longue ou non.

Dans le cas d'une distribution des temps d'attente non uniforme, une heuristique possible est d'utiliser une valeur seuil. Si une attente dure plus longtemps que cette valeur seuil, alors il est considéré que celle-ci sera longue. Il est alors nécessaire de déterminer la valeur seuil de manière à ce que la classification qui est faite soit la plus exacte possible, de façon à ne pas mettre en sommeil un processus dont l'attente sera courte. Cette détermination pourrait alors être faite par une analyse statique de la distribution des temps d'attente d'un ensemble d'exécutions, ou par une analyse dynamique à l'exécution. On peut alors envisager l'utilisation d'un mécanisme à fenêtre glissante.

5.2.1.2 Réduction du coût de synchronisation

Le fonctionnement d'un algorithme de communication producteur/consommateur se compose de deux parties : l'envoi des données à proprement parler et la synchronisation entre le producteur et le consommateur. Chacune de ces parties a un coût qu'il convient de minimiser. Traditionnellement, la synchronisation est une partie sur laquelle se concentrent en premier les efforts. Tout d'abord, bien que nécessaire, cette partie n'est pas directement utile à la communication dans le sens où aucune des données à communiquer n'est transférée pendant cette phase, contrairement à la partie d'envoi des données. De plus, cette partie est la plus compliquée et la plus coûteuse en temps. Comme expliquée dans les chapitres 2 et 3.

BatchQueue contient déjà plusieurs optimisations afin de réduire le coût de la synchronisation entre producteur et consommateur, parmi lesquelles une réduction de la fréquence de synchronisation. Cette approche consiste à envoyer plusieurs données en un seul bloc afin de réduire la proportion du temps passé en synchronisation. BatchQueue envoie ainsi par défaut les données par bloc de 32 octets et il est possible de réduire encore plus le coût de la synchronisation en augmentant la taille des blocs de données. Néanmoins, cela nuit alors à l'empreinte mémoire de BatchQueue et donc à ses performances en cas de pression sur la mémoire cache de premier niveau.

Pour réduire d'avantage le coût de la synchronisation, BatchQueue se limite à l'utilisation d'une unique variable booléenne. Le producteur et le consommateur se notifient alors leur progression par une inversion de valeur de cette variable. Le mécanisme de synchronisation ne nécessite donc pas de connaître la valeur de cette variable : seule son changement d'état détermine la progression de l'algorithme. Or, le jeu d'instructions des processeurs ne permet pas d'obtenir cette information. Il est donc nécessaire d'effectuer une lecture de la variable

pour réaliser son changement de valeur, ce qui demande plusieurs cycles processeurs afin de récupérer sa valeur courante.

Une solution possible consisterai à tirer profit de la technique de multitraitement simultané (« Simultaneous Multithreading ») pour ne pas perdre de cycle processeur lors de la lecture d'une variable afin d'obtenir sa dernière valeur. La technique repose sur l'utilisation de deux processus léger par cœur de calcul de telle manière que lorsque l'un d'eux effectue une lecture d'une variable dont la valeur a changé, le processeur se mette à exécuter le second processus léger qui poursuit la synchronisation. Le processus léger qui procède à la lecture de la variable joue alors le rôle d'aide au second en lui évitant de rester bloquer. Une idée semblable a d'ailleurs déjà été mise en pratique[KST11] en utilisant un processus léger qui en aide un autre en préchargeant les données dont il aura besoin.

5.2.1.3 Réduction du nombre de recopie des données

Bien que constituant la partie utile de la communication, l'envoi des données n'en reste pas moins un coût et celui-ci doit être réduit au minimum. Dans cette optique, les algorithmes de communication emploient de nombreuses techniques afin de réduire l'impact de la mise en cohérence des données entre le producteur et le consommateur. Cependant, bien qu'important, le coût de cette mise en cohérence est moindre comparé à celui de copie des données vers et depuis le tampon de communication par le producteur et le consommateur.

Or cette copie des données n'est pas requise dans le cas où les données ne sont plus réutilisées après envoi. La copie est même un coût inutile si les données à envoyer sont le résultat d'une modification de données déjà existantes. C'est le cas par exemple si les données à envoyer sont produites par des opérations arithmétiques effectuées directement sur des données déjà existantes. Une solution pour éviter ce problème serait de pouvoir donner à BatchQueue l'adresse du tampon dans lequel les données sont produites et de permettre au consommateur d'y lire directement les données. De cette façon, aucune copie des données ne serait effectuée.

5.2.2 Perspective à long terme

Une des deux contributions présentées dans cette thèse porte sur l'élaboration d'un algorithme de communication inter-cœurs, appelé BatchQueue optimisé pour une communication entre un unique producteur et consommateur. La motivation derrière cette contribution est la nécessité d'écrire des algorithmes parallèles pour mettre à profit l'ensemble des ressources disponibles dans un système multi-cœurs. En effet, les divers composants d'un algorithme parallèle sont amenés à communiquer fréquemment deux à deux, d'où l'intérêt pour un tel algorithme. Pour preuve, les performances du parallélisme de flux tel que proposé par l'extension de calcul par flux pour OpenMP sont améliorées de façon significative lorsque BatchQueue est utilisé.

La communication point à point ne constitue cependant pas la seule façon par laquelle communiquent les composants d'un algorithme parallèle. De nombreux algorithmes reposent en effet sur d'autres modes de communication tel la diffusion ou encore la communication entre plusieurs producteurs et plusieurs consommateurs. Ces formes de communication étant également très répandues, elles présentent un intérêt à être optimisées.

Bien que fondamentalement plus complexes que la communication point à point, ces formes de communication pourraient être améliorées en suivant certaines des approches em-

ployées dans la conception de BatchQueue. Ainsi, l'approche consistant à utiliser des tampons résidant dans des lignes de cache différentes pour le producteur et le consommateur pourrait également être appliquée pour réaliser une diffusion performante. En effet, cette technique, efficace dans le cas de BatchQueue, empêche que des écritures se produisent en parallèle de lectures sur un même ligne de cache. Ceci évite le coût lié à la mise en cohérence des caches. Or, cette situation se produit aussi dans le cas d'une diffusion si le producteur et les consommateurs accèdent au même tampon ou à des tampons résidant dans la même ligne de cache, d'où l'intérêt de cette approche pour la diffusion.

Cette même situation se produirait en utilisant un compteur partagé entre les consommateurs pour déterminer si tous les consommateurs ont pris connaissance d'une information donnée, comme cela est fait traditionnellement. Le compteur étant incrémenté par chaque consommateur puis comparé au nombre total de consommateurs participant à la diffusion, des écritures et des lectures se produisent en effet en parallèle sur le compteur. Il faudrait donc adopter une autre stratégie où chaque consommateur dispose d'une zone de mémoire distincte où il peut indiquer les données qu'il a lu.

Enfin, les améliorations de performances obtenues pour la communication point à point dans BatchQueue reposent sur une analyse détaillée de l'origine du coût du mécanisme de cohérence entre les caches. Cette analyse permet d'identifier des situations à éviter et des solutions permettant de les éviter. En suivant cette même méthodologie, d'autres formes de communication répandues, telles la diffusion et la communication entre plusieurs producteurs et plusieurs consommateurs, pourraient alors voir leurs performances améliorées.

Publications

Conférences internationales

[ICPADS12] Thomas Preud'homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. An improvement of OpenMP pipeline parallelism with the BatchQueue algorithm. In *18th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2012)*, Singapore, Singapore, December 2012. IEEE Computer Society Press.

[SPACPAD10] Thomas Preud'homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. BatchQueue : Fast and Memory-thrifty Core to Core Communication. In *22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2010)*, pages 215–222, Petrópolis, Brazil, October 2010. IEEE.

Poster international

[ASPLOS11] Thomas Preud'homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. BatchQueue : Efficient core-to-core communication for pipeline parallelism. *Poster for the ASPLOS 2011 conference*, Newport Beach, California, USA, March 2011.

Conférence française

[CFSE11] Thomas Preud'homme, Julien Sopena, Gaël Thomas, and Bertil Folliot. BatchQueue : file producteur/consommateur optimisée pour les multi-cœurs. In *8th Conférence Française en Systèmes d'Exploitation (CFSE'08)*, Saint-Malo, France, Mai 2011.

Références

A

[AFI+09] J. Alglave, A. Fox, S. Ishtiaq, M.O. Myreen, S. Sarkar, P. Sewell, and F.Z. Nardelli. The semantics of power and arm multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 13–24. ACM, 2009.

[AGH+11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order : expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[AH90] Sarita Adve and Mark D. Hill. Weak ordering - a new definition. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, 1990.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[AMD12] AMD. AMD64 Architecture Programmer's Manual Volume 2 : System Programming. http://support.amd.com/us/Embedded_TechDocs/24593.pdf, 2012. Section 7.3.

B

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

[Boa] OpenMP Architecture Review Board. OpenMP API specification for parallel programming. http://openmp.org.

[ccb12] Cc by 3.0. http://creativecommons.org/licenses/by-sa/3.0, 2012.

D

[Dre07] Drepper, Ulrich. What every programmer should know about memory, 2007.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *ACM SIGARCH Computer Architecture News*, 14(2) :434–442, 1986.

[DSB88] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2) :9–21, 1988.

F

[Fur03] M.A. Furis. *Cache miss analysis of Walsh-Hadamard Transform algorithms*. PhD thesis, Drexel University, 2003.

[Git] Git repository of OpenMP stream extension with BatchQueue. git://git.celest.fr/rt_gccstream.git.

G

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM, 1990.

[GLR83] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2) :189, 1983.

[GMV08] J. Giacomoni, T. Mosely, and M. Vachharajani. Fastforward for efficient pipeline parallelism : A cache-optimized concurrent lock-free queue. In *Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008.

[GNU] GNU. GNU Compiler Collection. http://gcc.gnu.org/.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII*, pages 151–162, 2006.

H

[HAF+07] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 341–354, New York, NY, USA, 2007. ACM.

[HHK+10] Ruijing Hu, Xucen Huang, Michel Kieffer, Olivier Derrien, and Pierre Duhamel. Robust critical data recovery for mpeg-4 aac encoded bitstreams. In *ICASSP*, pages 397–400, 2010.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition edition, 2007.

[HSS07] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. *Principles of Distributed Systems*, pages 401–414, 2007.

I

[Inta] Intel. Array Building Blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[Intb] Intel. Cilk Plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[Intc] Intel. Threading Building Blocks. http://threadingbuildingblocks.org/.

[IS98] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage-fine-grain sharing in page-based dsms. *Operating systems review*, 33 :215–228, 1998.

K

[Kel95] P. Keleher. *Lazy release consistency for distributed shared memory*. PhD thesis, Rice University, 1995.

[KSB95] L.I. Kontothanassis, M.L. Scott, and R. Bianchini. Lazy release consistency for hardware-coherent multiprocessors. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 61–61. IEEE, 1995.

[KST11] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *ASPLOS*, pages 393–404, 2011.

L

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9) :690–691, 1979.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2) :190–222, 1983.

[LBC10] P.P.C. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *IPDPS '10 : Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[LDT+] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking : Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *USENIX Annual Technical Conference*.

[LMS04] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Proceedings of Distributed Computing*, pages 117–131, 2004.

M

[M+98] G.E. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1) :82–85, 1998.

[MC87] J.M. Mellor-Crummey. Concurrent queues : Practical fetch-and-ϕ algorithms. Technical report, Technical Report 229, Computer Science Department, University of Rochester, 1987.

[MLKD10] C. Marin, Y. Leprovost, M. Kieffer, and P. Duhamel. Robust mac-lite and soft header recovery for packetized multimedia transmission. *Communications, IEEE Transactions on*, 58(3) :775–784, 2010.

[MNSS05] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, page 262. ACM, 2005.

[Mos93] D. Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1) :18–26, 1993.

[MS98] M.M. Michael and M.L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1) :1–26, 1998.

[MS09] J. Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 282–288. IEEE, 2009.

N

[Net12] Microsoft Developer Network. Communication Inter-Processus sous Microsoft Windows. http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx, 2012.

P

[PC11] Antoniu Pop and Albert Cohen. A stream-computing extension to openmp. In *International Conference on High Performance and Embedded Architectures and Compilers*, pages 5–14. ACM, 2011.

[PLJ91] S. Prakash, Y. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical report, University of Florida, 1991.

[PLJ94] S. Prakash, Y.H. Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, pages 548–559, 1994.

[Pro12] The Linux Documentation Project. Communication Inter-Processus sous UNIX. http://tldp.org/LDP/lpg/node7.html, 2012.

[Rec] Receive packet steering. http://lwn.net/Articles/362339/.

S

[SSN+09] S. Sarkar, P. Sewell, F.Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M.O. Myreen, and J. Alglave. The semantics of x86-cc multiprocessor machine code. *ACM SIGPLAN Notices*, 44(1) :379–391, 2009.

[Sut05] H. Sutter. The free lunch is over : A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3) :202–210, 2005.

[SVS94] L.M. Silva, B. Veer, and J.G. Silva. The helios tuple space library. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 325–331. IEEE, 1994.

T

[TZ01] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, page 143. ACM, 2001.

V

[Val94] J.D. Valois. Implementing lock-free queues. In *In Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV*, pages 64–69, 1994.

W

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos) : the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2) :76–85, April 2009.

[Wil94] Gregory V. Wilson. The History of the Development of Parallel Computing. http://ei.cs.vt.edu/~history/Parallel.html, 1994.

[Wil02] William Thies and Michal Karczmarek and Saman Amarasinghe. Streamit : A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[WKWY07] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO '07 : Proceedings of the International Symposium on Code Generation and Optimization*, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

Z

[ZOYB09] Y. Zhang, K. Ootsu, T. Yokota, and T. Baba. Clustered Communication for Efficient Pipelined Multithreading on Commodity MCPs. *IAENG International Journal of Computer Science*, 36, 2009.

Thomas PREUD'HOMME

Communication inter-cœurs optimisée pour le parallélisme de flux.

Résumé

Parmi les différents paradigmes de programmation parallèle, le parallélisme de flux présente l'avantage de conserver la séquentialité des algorithmes et d'être ainsi applicable en présence de dépendances de données. De plus, l'extension de calcul par flux pour OpenMP proposée par Pop et Cohen permet de mettre en œuvre cette forme de parallélisme sans requérir de réécriture complète du code, en y ajoutant simplement des annotations. Cependant, en raison de l'importance de la communication nécessaire entre les cœurs de calcul, les performances obtenues en suivant ce paradigme sont très dépendantes de l'algorithme de communication utilisé. Or l'algorithme de communication utilisé dans cette extension repose sur des files gérant plusieurs producteurs et consommateurs alors que les applications mettant en œuvre le parallélisme de flux fonctionnent principalement avec des chaînes de communication linéaires.

Afin d'améliorer les performances du parallélisme de flux mis en œuvre par l'extension de calcul par flux pour OpenMP, cette thèse propose d'utiliser, lorsque cela est possible, un algorithme de communication plus spécialisé nommé BatchQueue. En ne gérant que le cas particulier d'une communication avec un seul producteur et un seul consommateur, BatchQueue atteint des débits jusqu'à deux fois supérieurs à ceux des algorithmes existants. De plus, une fois intégré à l'extension de calcul par flux pour OpenMP, l'évaluation montre que BatchQueue permet d'améliorer l'accélération des applications jusqu'à un facteur 2 également. L'étude montre ainsi qu'utiliser des algorithmes de communication spécialisés plus efficaces peut avoir un impact significatif sur les performances générales des applications mettant en œuvre le parallélisme de flux.

Mots-clefs : Multi-cœurs, parallélisme de flux, OpenMP, file producteur consommateur, caches matériels, MOESI

Abstract

Among the various paradigms of parallelization, pipeline parallelism has the advantage of maintaining sequentiality of algorithms, thus being applicable in case of data dependencies. More over, the stream-computing extension for OpenMP proposed by Pop and Cohen allows to apply this form of parallelization without needing a complete rewrite of the code, by simply adding annotations to it. However, due to the *importance* of the communication needed between the cores, the performances obtained by following this paradigm depends very much on the communication algorithm used. Yet, the communication algorithm used in this extension relies on queues that can handle several producers and consumers while applications using pipeline parallelism mainly works with linear communication chains.

To improve the performances of pipeline parallelism implemented by the stream-computing extension for OpenMP, this thesis propose to use, whenever possible, a more specialized communication algorithme called BatchQueue. By only handling the special case of a communication with one producer and one consumer, BatchQueue can reach throughput up to two time faster than existing algorithms. Furthermore, once integrated to the stream-computing extension for OpenMP, the evaluation shows that BatchQueue can improve speedup of application up to a factor 2 as well. The study thus shows that using a more efficient specialized communication algorithm can have a significant impact on overall performances of application implementing pipeline parallelism.

Keywords : Multi-cores, pipeline parallelism, OpenMP, producer consumer queue, hardware cache, MOESI