RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

BatchQueue : file producteur / consommateur optimisée pour les multi-cœurs

Thomas PREUD'HOMME, Julien SOPENA, Gaël THOMAS et Bertil FOLLIOT

Université de Paris 6,
LIP6, 4 place jussieu,
75005 Paris - France
thomas.preud-homme@lip6.fr

Résumé
Les applications séquentielles peuvent tirer partie des systèmes multi-cœurs en utilisant le parallélisme pipeline pour accroître leur performance. Dans un tel schéma de parallélisme, l'accélération possible est limitée par le surcoût dû à la communication cœur à cœur. Ce papier présente l'algorithme BatchQueue, un système de communication rapide conçu pour optimiser l'utilisation du cache matériel, notamment au regard du pré-chargement. BatchQueue propose des performances améliorées d'un facteur 2 : il est capable d'envoyer un mot de données en 3,5 nanosecondes sur un système 64 bits, représentant un débit de 2 Gio/s.

Mots-clés : communication, mémoire partagée, optimisation, cache matériel

1. Introduction

Depuis plusieurs années maintenant, la fréquence des processeurs a cessé d'augmenter pour cause de contraintes physiques. Les fondeurs de puces profitent maintenant du nombre croissant de transistors par cm^2 pour augmenter le nombre de cœurs de calcul sur une même puce [3]. Or, seules les applications parallèles peuvent tirer un réel profit de ces nouvelles architectures. Parmi les diverses techniques de parallélisation, celle dite du pipeline s'avère particulièrement intéressante du fait de ses propriétés intrinsèques : (i) indépendance du fonctionnement par rapport à l'application parallélisée, (ii) simple à mettre en place dans les codes patrimoniaux et malgré tout (iii) très efficace en terme d'accélération.

Le parallélisme pipeline consiste à découper une tâche séquentielle en plusieurs sous-tâches appelées étapes et exécutées en séquence. Chaque étape prend en entrée le résultat de l'étape précédente, à la manière des commandes UNIX chaînée à l'aide de tubes anonymes. Le parallélisme apparaît alors lors du traitement d'un flux de données : les différentes étapes sont exécutées en parallèle sur des données successives.

Une des limites de ce parallélisme réside dans le temps de communication entre deux processeurs, les stades s'exécutant sur différents cœurs de calcul. Plus le nombre de cœurs augmente, plus le temps de calcul sur chaque cœur devient petit et se rapproche du temps de communication. Avec l'augmentation du nombre de cœurs, il devient nécessaire d'avoir un temps de communication très court, de façon à conserver un rapport entre temps de calcul par cœur et temps de communication important.

Plusieurs solutions existent pour assurer une communication efficace entre deux cœurs. Lamport le premier [6] proposa un algorithme évitant toute primitive de synchronisation dans le cadre d'un unique

producteur et consommateur. Par la suite, de nombreuses optimisations [15, 7, 1, 16] ont été proposés pour améliorer les performances. Ces solutions reposent sur une meilleure prise en compte du fonctionnement des caches matériels, notamment du protocole de cohérence des caches.

Cet article présente l'algorithme BatchQueue, une file d'attente producteur / consommateur proposant de nouvelles optimisations de l'utilisation des caches avec un doublement des performances par rapport aux solutions existantes. BatchQueue présente un fonctionnement par flot et n'utilise qu'une unique variable partagée pour assurer la synchronisation entre producteur et consommateur. BatchQueue réduit également les effets de bord dûs au pré-chargement des données dans les caches matériels en éloignant les différentes données partagées en mémoire.

Ces caractéristiques permettent à BatchQueue d'obtenir des performances grandement améliorées par rapport à l'état de l'art qu'est CSQ [16]. Sur un architecture 64 bits, BatchQueue est capable d'envoyer une donnée de la taille d'un mot en 3,5 nanosecondes, permettant de découper une tâche en sous-tâches de l'ordre de quelques dizaines de nanosecondes en ayant toujours une accélération importante. Exprimé en débit, cela représente une communication à 2 Gio/s.

Cet article contient plusieurs contributions :

– un algorithme de communication cœur à cœur efficace optimisant l'utilisation du cache matériel ;
– une analyse détaillée de l'évaluation des performances, mettant notamment en relief : l'importance du mécanisme de synchronisation entre producteur et consommateur mais aussi l'influence considérable du pré-chargement matériel des données dans le cache.

Le reste de cet article est organisé de la façon suivante. Tout d'abord, la section 2 présente BatchQueue, l'algorithme de communication cœur à cœur. L'état de l'art est ensuite présenté section 3. Les mesures de performances sont alors détaillées dans la section 4. Enfin, la section 5 conclut cet article.

2. Algorithmes efficaces de communication inter-cœurs

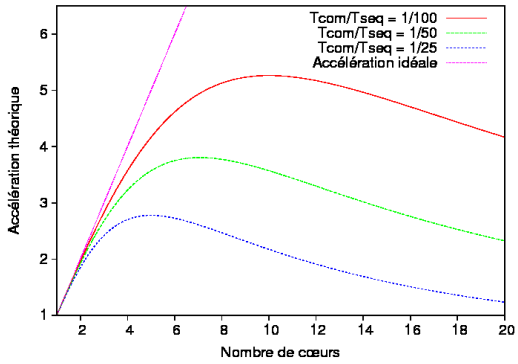


FIGURE 1 – Influence de la vitesse de communication sur l'accélération du parallélisme pipeline

Pour avoir une bonne accélération sur un nombre important de cœurs, la technique de parallélisme pipeline nécessite une communication particulièrement efficace. En effet, l'accélération théorique est donnée par la formule suivante : $\frac{1}{\frac{1}{x}+(x-1)\frac{T_{comm}}{T_{seq}}}$, où T_{seq} est le temps d'exécution en séquentiel de l'application, T_{comm} le temps de communication entre deux cœurs de calcul et x le nombre de ces cœurs. Autrement dit, plus le rapport entre le temps d'exécution séquentiel et le temps de communication est

grand, plus l'accélération théorique pour un nombre donné de cœurs est grande. La figure 1 nous montre qu'une diminution du rapport $\frac{T_{comm}}{T_{seq}}$ permet, d'une part d'augmenter l'accélération lorsque celle-ci est possible , d'autre part de retarder le moment où l'ajout de cœur dégrade l'accélération.

Pour une tâche donnée, il est donc important de réduire le temps de communication (T_{comm}) afin d'optimiser l'utilisation des cœurs disponibles. Or, concevoir un mécanisme de communication efficace dans les architectures multi-cœurs passe par la considération de contraintes matérielles. L'algorithme que nous avons mis au point se fixe donc trois contraintes : réduire le nombre de variables partagées, réduire la synchronisation et prévenir tout effet de bord du pré-chargement de lignes de cache.

Cette section présente tout d'abord la file sans verrou de Lamport, l'algorithme de référence en terme de communication inter-cœurs et les problèmes rencontrés en rapport avec le système de cohérence des caches. Dans un second temps, un ensemble de solutions est proposé pour répondre à ce problème, sous la forme de l'algorithme BatchQueue.

2.1. File sans verrou de Lamport

Considérons un instant l'algorithme classique composé des fonctions produit_lamport() et consomme_lamport() présentés figure 2 et 3 ci-dessous. Cet algorithme utilise 2 variables partagées : ind_{cons} et ind_{prod} et les lectures et écritures se font mot à mot.

FIGURE 2 – produit_lamport( )

Attendre ind_{prod} != ind_{cons} ;
tab[ind_{prod}] ⟵ data ;
ind_{prod} ⟵ case suivante(ind_{prod}) ;

FIGURE 3 – consomme_lamport( )

Attendre case suivante(ind_{cons}) != ind_{prod} ;
data ⟵ tab[ind_{cons}] ;
ind_{cons} ⟵ case suivante(ind_{cons}) ;
Result : data

Les figures 4(a) à 4(d) présentent l'exécution de cet algorithme dans une architecture multi-cœurs à deux niveaux de cache, l'un partagé entre les différents cœurs de calcul, l'autre propre à chaque cœur. Si les caches exclusifs sont plus près du processeur et donc plus rapides d'accès, ils sont également plus petits. Le cache partagé en revanche est plus gros, mais aussi plus lent. La mémoire centrale n'est pas présentée sur ces schémas car on suppose les données utilisées déjà chargées dans le cache partagé.

Le maintien de la cohérence entre ces différents caches cause une pénalité importante en terme de cycles processeur lorsqu'une variable est partagée en lecture / écriture par deux cœurs différents. Ce partage est coûteux aussi bien lors d'une écriture que d'une lecture. En effet, une écriture dans une variable partagée, effectuée sur un processeur P_i, implique l'invalidation de la variable dans les caches exclusifs des autres cœurs. Les accès en lecture sur les autres cœurs P_j nécessiteront donc de récupérer par la suite la dernière valeur sur le cœur P_i. Ceci nécessite bien plus de cycles que pour un simple accès au cache exclusif.

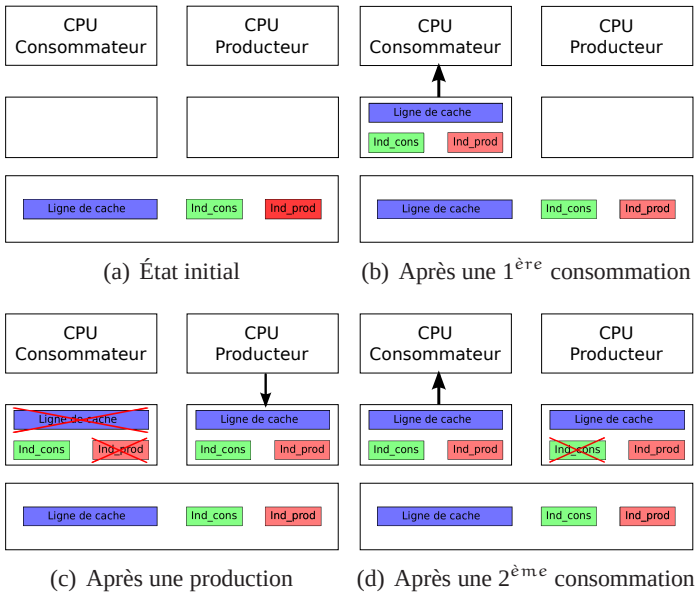


FIGURE 4 – Fonctionnement de l'algorithme producteur / consommateur classique

La figure 4(a) présente la situation à l'origine où : le tampon partagé entre le producteur et le consommateur et leurs indices se trouvent uniquement dans le cache partagé. 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 4(b)). La valeur courante des deux indices ainsi que du tampon sont alors chargés dans le cache exclusif. La récupération de ces valeurs depuis le niveau 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 exclusif (figure 4(c)) pour vérifier qu'il a la place d'écrire une nouvelle donnée. Lorsque, le producteur écrit la donnée dans le tampon et incrémente son indice de production, ces données sont invalidées dans le cache du consommateur. Cette invalidation coûte également cher car elle doit être effectuée sur tous les cœurs, qu'ils possèdent ou non la donnée.

Toutes les valeurs qui ont été invalidées devront être rechargées depuis le cœur du producteur lorsque le consommateur effectuera à nouveau une lecture (figure 4(d)). Contrairement à la figure 4(a) où les caches exclusifs étaient vides, cette consommation s'accompagnera d'une invalidation de l'indice du consommateur lorsque celui-ci est incrémenté.

Il apparaît ici que l'algorithme classique génère 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 : une lecture distante lorsque un des participants consulte l'indice de l'autre participant, une invalidation lorsque l'un des participants met à jour son indice et enfin une invalidation pour écrire une donnée dans le tampon puis une lecture distante pour lire cette donnée.

2.2. BatchQueue

L'algorithme que nous proposons est présenté dans les fonctions produit_batchqueue() et consomme_batchqueue(), figure 5 et 6. Le principe est de diviser le tampon en deux parties : à tout instant le producteur et le consommateur travaillent dans des tampons distincts. Une synchronisation se produit avant d'échanger de tampon entre le consommateur et le producteur. Son fonctionnement ne repose que sur une unique variable (*statut*)grâce au maintient d'un invariant. De plus, lorsque cette variable est utilisée pour une attente active, l'algorithme garantit qu'elle n'est pas utilisée par l'autre participant, diminuant ainsi son degré de partage.

FIGURE 5 – produit_batchqueue( )

```
tab[ind_prod] ⟵ data ;
ind_prod ⟵ case suivante ;
si ind_prod = début d'un tampon alors
    Attendre status = faux ;
    status ⟵ vrai ;
fin
```

FIGURE 6 – consomme_batchqueue( )

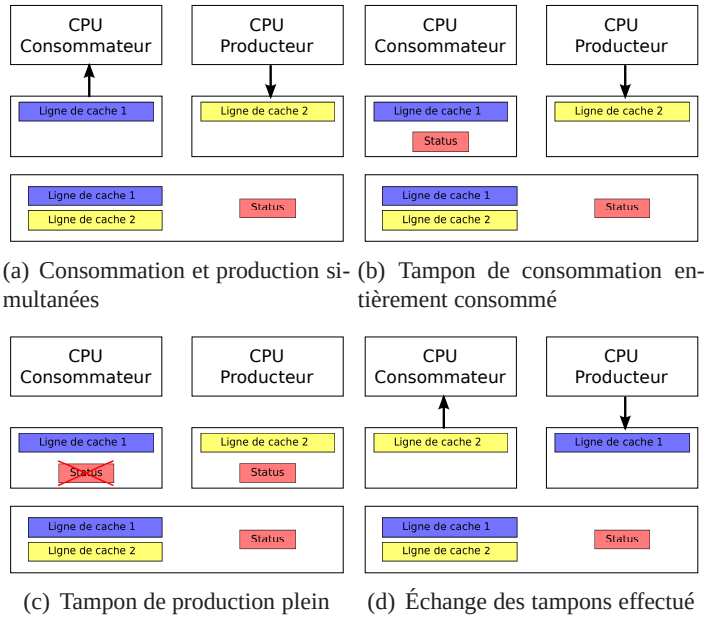
```
Attendre status = vrai ;
pour i ⟵ ind_cons to fin de tampon faire
    copy_buf[i] ⟵ tab[i] ;
fin
ind_cons ⟵ indice de l'autre tampon ;
status ⟵ faux ;
```

La figure 7 présente le déroulement de l'algorithme BatchQueue sur la même architecture que dans l'exemple précédent. La figure (figure 7(a)) présente le producteur et le consommateur alors qu'ils produisent et consomment sur des tampons distincts. Comme on peut le voir, la variable *status* n'est alors pas utilisée. Le producteur et le consommateur peuvent donc agir en parallèle sans occasionner de défaut de cache, contrairement à l'algorithme producteur / consommateur classique.

Une fois son tampon entièrement consommé (figure 7(b)), le consommateur met *status* à **faux**, récupérant la variable dans son cache exclusif au passage. Puis, il se met en attente jusqu'à ce que celle-ci ait pour valeur **vrai**.

Lorsque le producteur fini à son tour de produire (figure 7(c)), il vérifie tout d'abord que *status* ait pour valeur **faux**, récupèrant ainsi *status* dans son cache exclusif. Ce test permet au producteur de s'assurer que le consommateur a terminé la consommation de son propre tampon. Puis, il met *status* à **vrai**, invalidant la valeur de *status*.

La synchronisation est alors terminée (figure 7(d)) : le producteur et le consommateur recommencent à produire et consommer en ayant interchangé leur tampon de travail. Il y a donc avec cet algorithme 2 lectures distantes et 2 invalidations par tampon lu *et* écrit, en utilisant une unique variable partagée.



(a) Consommation et production simultanées

(b) Tampon de consommation entièrement consommé

(c) Tampon de production plein

(d) Échange des tampons effectué

FIGURE 7 – Fonctionnement de l'algorithme producteur / consommateur optimisé

3. Positionnement par rapport aux solutions alternatives

La section précédente présente les détails de l'algorithme BatchQueue, en particulier ce qu'il apporte par rapport à l'algorithme de Lamport. Cependant, d'autres solutions ont apporté des améliorations à l'algorithme de Lamport. Cette section présente ces différentes solutions de files producteur(s) / consommateur(s), leurs avantages et défauts, en incluant les files avec plusieurs producteurs et consommateurs.

Files avec producteurs et consommateurs multiples

À partir des travaux de Lamport sur les files sans verrous [6], un nombre important de 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 [13, 14], ou dynamique [14, 2, 5, 4, 8, 11, 12, 9, 10]. 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 verrous 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 une complexité accrue : les algorithmes sont plus complexes, reposent sur des opérations atomiques telles des compare-and-swap (comparaison et échange) et sont donc plus lents. De plus, tout comme la file à producteur et consommateur uniques de Lamport, ces algorithmes ne prennent pas de précautions particulières vis à vis du système de cohérence des caches.

FastForward

[15, 7, 1, 16] ont tous pour particularité de fournir un système de communication inter-cœurs avec producteur et consommateur uniques en tenant compte du système de cohérence des caches dans leur fonctionnement. Parmis ces algorithmes, FastForward [1] est le plus simple. La solution adoptée pour éviter

les invalidations de lignes de caches est de réserver une des valeurs que peuvent prendre les données pour indiquer quand une entrée du tampon est vide. De cette façon, le consommateur et le producteur ont uniquement besoin de manipuler leur propre indice et de lire la valeur de l'entrée suivante du tampon pour pouvoir consommer ou produire.

Cette élégante approche est malgré tout toujours sujette à des invalidations de lignes de cache si le producteur et le consommateurs travaillent sur des entrées du tampon se trouvant dans la même ligne de cache. Pour résoudre ce problème, les auteurs de ce travail proposent de temporiser la consommation par rapport à la production, c'est à dire d'attendre qu'une ligne de cache entière ait été produite avant de consommer une donnée. De cette façon, le producteur et le consommateur agissent toujours sur des lignes de cache distinctes.

Cette temporisation peut être maintenue en effectuant une attente active dès que la distance entre le producteur et du consommateur est trop faible. 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 invalidation de lignes de cache que cherche précisément à éviter l'algorithme FastForward.

BatchQueue résout ce problème en relaxant le contrôle de la temporisation entre production et consommation : la transmission des données est retardée grâce à une variable de synchronisation additionnelle uniquement utilisée lorsqu'un tampon entier de plusieurs lignes de cache a été traité.

DBLS and MCRingBuffer

DBLS [15] et MCRingBuffer [7] proposent de retarder l'envoi des données jusqu'à ce que N lignes de cache soient remplies, N étant plus grand que 1. Cet envoi retardé des données permet d'atteindre le cas optimal vis à vis du tampon de partage d'une invalidation de ligne de cache par ligne de cache envoyée.

Les invalidations dues au changement de valeur des indices de consommation et production sont également 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 : à chaque donnée produite (respectivement consommée) pour la copie locale et 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.

BatchQueue se differentie de DBLS et MCRingBuffer par le nombre de variables utilisées, en particulier le nombre de variables partagées : DBLS et MCRingBuffer utilisent 8 variables, dont 2 partagées indiquant le tampon en cours d'utilisation par le consommateur et le producteur alors que BatchQueue ne nécessite que 3 variables dont une unique variable partagée. De plus, BatchQueue tient compte du pré-chargement des lignes de cache dans l'organisation de ses données, afin d'éviter tout effet de bord indésirable.

File logicielle par lot

La « file logicielle par lot [16] » ou CSQ (pour Clustered Software Queue) est l'algorithme le plus proche de BatchQueue au niveau de son fonctionnement. Comme BatchQueue, CSQ utilise la technique d'envoi retardé des données pour minimiser les invalidations de lignes de cache et utilise le changement de valeur d'un bit pour indiquer quand un tampon est rempli ou vidé. Malgré un principe de fonctionnement similaire, les deux algorithmes diffèrent en deux points. Tout d'abord, avec les paramètres recommandés dans l'article CSQ comporte un nombre plus grand de tampons [1], chacun ayant un bit de synchronisation indiquant si le tampon est vide ou plein. L'idée est d'offrir plus de souplesse au niveau de la synchronisation entre le producteur et le consommateur. De plus, les différents éléments de la structure de donnée sont consécutifs en mémoire.
Chacune de ces deux différences ont un impact sur les performances de la solution CSQ. Tout d'abord, bien qu'offrant plus de souplesse au regard de la régularité du flux de données, le nombre important de tampons augmente le nombre de synchronisations. En effet, pour remplir l'ensemble des tampons, il est nécessaire d'effectuer autant de synchronisations que de tampons. Avec uniquement deux tampons, BatchQueue ne nécessite que deux synchronisations. Ensuite, les accès aux différents tampons et bits de synchronisation étant séquentiel, le pré-chargement cherche à récupérer les prochains éléments automatiquement. Ce faisant, des lignes de cache en train d'être modifiées sont chargées, entraînant une augmentation du nombre d'invalidations, ralentissant de ce fait le consommateur.

Il ressort de cet état de l'art que BatchQueue remplit parfaitement son objectif d'efficacité et tiens la comparaison avec les solutions alternatives. BatchQueue parvient à éviter les invalidations de lignes de cache en réduisant le nombre de variables partagées à un unique bit. BatchQueue parvient également à réduire les effets de bord indésirable du pré-chargement des lignes de cache en éloignant les différentes structures de données utilisées par l'algorithme.

4. Mesures de performance

La comparaison théorique de BatchQueue avec l'existant présenté dans la section précédente suggère que sa conception lui permet d'être plus efficace. En effet, la synchronisation nécessaire entre le producteur et le consommateur est faible et sa conception doit permettre d'éviter les effets de bord indésirables du pré-chargement des lignes de cache. Cependant, dans le cadre du partage de données, le comportement du cache est toujours difficile à prévoir. Des mesures de performance restent donc l'unique preuve d'efficacité.
Trois séries de mesure sont présentées dans cette section. Les deux premières séries comparent les performances de BatchQueue aux autres algorithmes de communication producteur / consommateur dans deux cas d'utilisation différents : les programmes dont la communication inter-cœur représente une proportion importante du temps d'exécution d'une part et les programmes effectuant principalement du calcul d'autre part. La troisième série se concentre sur BatchQueue et étudie l'influence de l'existence ou non d'un cache partagé entre le cœur producteur et le cœur consommateur.

4.1. Comparaison avec l'existant

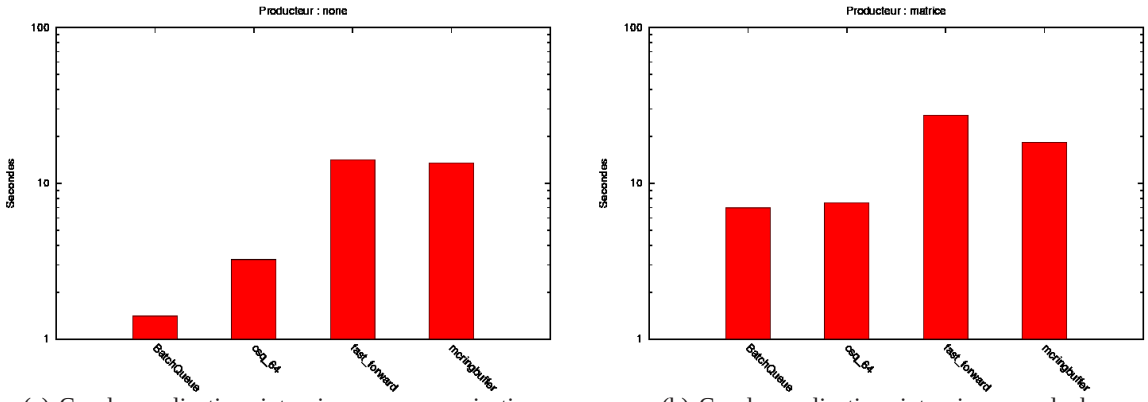
Les deux premières séries de mesure consistent à observer le temps nécessaire pour que le producteur effectue 400 millions écritures d'un mot mémoire et que le consommateur lise les 400 millions de données

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

correspondantes. C'est donc un débit qui est mesuré : le nombre de secondes pour 400 millions d'écritures. Le test est effectué pour deux types de programmes : les programmes intensifs en communication et les programmes intensifs en calcul.

Dans le premier cas le producteur ne fait qu'écrire des données en boucle, ce qui permet d'obtenir le débit maximal de chaque algorithme, lorsque seul les algorithmes de communication utilisent le cache. Dans le second cas, le producteur effectue un calcul matriciel entre deux écritures, ce qui nécessite d'utiliser le cache L1 et permet de mesurer l'influence de la quantité de cache utilisée par l'algorithme producteur / consommateur sur l'efficacité du calcul. Les différents algorithmes comparés sont : BatchQueue, Concurrent Shared Queue (CSQ), MCRingBuffer et FastForward. La somme des tampons de communication utilisés dans chacun des algorithmes est fixée à 64 lignes de cache comme proposé dans l'article [16].

Ces deux séries de mesure ont été effectuées sur une machine comprenant un processeur quad-cœur Xeon W3520 cadencé à 2.67 GHz avec 4 Gio [2] de RAM avec un système 64 bits. Les résultats sont présentés dans les figures 8(a) et 8(b).



(a) Cas des applications intensives en communication (b) Cas des applications intensives en calcul

FIGURE 8 – Comparaison des débits des différentes files d'attente producteur / consommateur

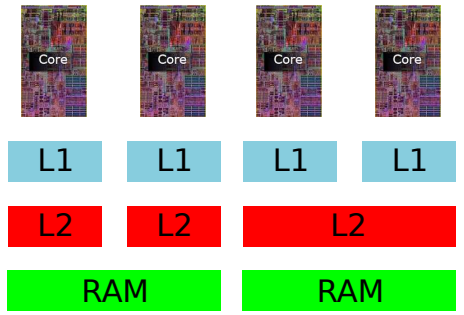
Les deux histogrammes montrent que BatchQueue présente de meilleures performances dans les deux cas de figure, avec une différence bien plus marquée dans le cas d'une production sans calcul matriciel. Cette différence s'explique par deux phénomènes. Tout d'abord, il se produit un recouvrement du temps 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 entre BatchQueue et CSQ en particulier. Ensuite, la différence de temps entre BatchQueue et les autres techniques est moins visible sur le test avec calcul matriciel car la proportion de temps passé à communiquer est faible au regard du temps passé à calculer. Ainsi l'écart entre BatchQueue est CSQ parait faible mais la différence de temps dans les deux cas est proche, autour de deux secondes sans calculs et autour d'une seconde avec calculs.

Par ailleurs, le temps de communication dans le cas d'une production sans calcul matriciel est le plus important dans le cadre du parallélisme pipeline. En effet, le temps de communication devient critique pour le parallélisme pipeline quand le nombre de cœurs est grand et que le temps de calcul par cœur est faible, autrement dit lorsque la proportion de temps de communication par rapport au temps total –

2. 1 Gio, ou gibioctet représente 2^{30} octets, par opposition à 1 Go qui représente 10^9 octets dans le système international

temps de calcul et de communication – est grand, ce qui est proche de la situation mesurée dans le test de communication sans calcul.

4.2. Influence du partage de cache



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

FIGURE 9 – Architectures multi-processeurs

Les deux premières séries de mesure montrent que BatchQueue tient la comparaison avec les autres algorithmes de communication. BatchQueue propose un débit environ deux fois meilleur et reste efficace avec une utilisation importante du cache. Comme le montre la section précédente, ces résultats sont dûs à 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 d'un cache partagé ou non. En effet, le faible partage de donnée entre le producteur et le consommateur rend BatchQueue relativement agnostique de la hiérarchie matérielle des caches. La troisième série de mesure vise à vérifier cette hypothèse en comparant le débit obtenu par BatchQueue sur deux architectures de cache différentes, représentées par la figure 9 : une architecture avec un cache partagé, et une architecture où seule la mémoire est partagée entre les différents cœurs.

Le test est exécuté sur une unique machine comprenant deux processeurs quad-cœur Xeon X5472 cadencé à 3 GHz avec 10 Gio de RAM. Chaque processeur comprends deux paires de cœurs, chaque paire comprenant un cache de niveau L2 partagé entre les deux cœurs la constituant. De cette façon, les deux architectures de caches peuvent être testées en plaçant le producteur et le consommateur sur deux cœurs d'une même paire ou non. Le même protocole expérimental est utilisé que pour les deux premières séries de mesures : l'envoi de 400 millions mots mémoire du producteur vers le consommateur avec un calcul matriciel effectué entre deux envois dans le cas de la seconde série de mesure. Les résultats sont présentés figure 10.

Il ressort des deux tests que la présence d'un cache partagé améliore systématiquement les performances mais de façon plus prononcée dans le cas d'un envoi avec calcul matriciel. En effet, un cache partagé permet d'éviter toute communication inter-cœur dans le cas d'une ligne de cache partagée en lecture et écriture entre deux cœurs de calcul. Chacun des cœurs consume et écrit directement dans la ligne de cache évitant ainsi toute mise en cohérence du cache avec d'autres caches distants.

L'effet est nettement plus marqué dans le cas du calcul matriciel : le gain en performance avec un cache partagé est de l'ordre de 30%. Ceci s'explique par l'utilisation plus importante du cache due aux calculs matriciels, créant des évictions de lignes de cache utilisées par BatchQueue.

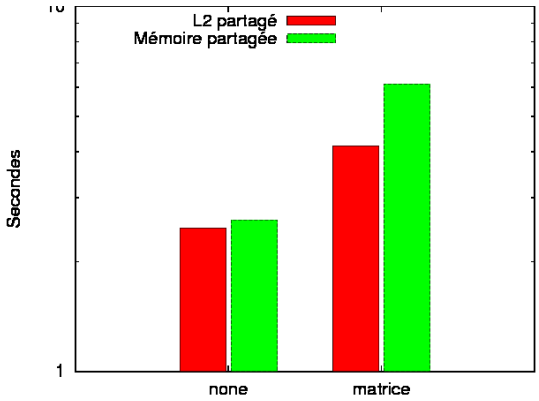


FIGURE 10 – Comparaison des débits de BatchQueue avec ou sans cache partagé

Dans le cas du test sans calcul matriciel, en revanche, la performance reste stable en l'absence d'un cache matériel partagé. 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 est très encourageant puisqu'il signifie que BatchQueue devrait fournir un bon débit de communication avec des latences importantes entre les différents cœurs, comme c'est le cas dans les systèmes NUCA [3]. Or comme dit précédement, les performances de BatchQueue dans le cas d'une communication sans calculs matriciels sont les plus intéressants car ils donnent une idée des performances possibles dans le cadre du parallélisme pipeline avec un grand nombre de cœurs, ou dans une utilisation de BatchQueue avec des pics (« bursts ») de communication.

Les résultats présentés dans cette section confirment les avantages de BatchQueue par rapport aux autres algorithmes de communication présentés dans la section précédente. Sur un machine 64 bits, sa conception lui permet, dans le cas d'une communication intensive, d'envoyer une donnée de la taille d'un mot mémoire en 3,5 nanosecondes, offrant ainsi un débit de plus de 2 Gio/s. Elle lui permet également d'être peu sensible à la latence d'accès aux données partagées de l'algorithme.

5. Conclusion

Cet article présente une nouvelle file d'attente producteur / consommateur sans synchronisation appelée BatchQueue. BatchQueue offre par rapport aux solutions existantes dans la littérature une meilleure prise en compte du système de cohérence des caches. Il est conçu de manière à éviter les invalidations de ligne de caches ainsi que les échanges de ligne de caches entre le producteur et le consommateur. Par rapport aux solutions existantes, BatchQueue propose une synchronisation réduite entre le producteur et le consommateur – seule un bit est requit pour synchroniser le producteur avec le consommateur – ainsi qu'une prise en compte du pré-chargement des lignes de cache.

Ces différentes caractéristiques font de BatchQueue une file d'attente producteur / consommateur particulièrement rapide. Lors des tests expérimentaux, il s'est montré 2 fois plus rapide que le plus efficace des algorithmes existants. BatchQueue est donc bien adapté pour toute tâche très communicative n'ayant

3. NUCA : Non Uniform Cache Architecture

pas de contrainte temps réel, telle que le parallélisme pipeline ou l'instrumentation de code. Des améliorations restent cependant possible, notamment l'ajout du support de plusieurs producteurs et écrivains. Ce qui permettrait d'augmenter son spectre d'utilisation.

Bibliographie

1. J. Giacomoni, T. Mosely, and M. Vachharajani. Fastforward for efficient pipeline parallelism : A cache-optimized concurrent lock-free queue. In *In PPoPP '08 : Proceedings of the The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 2008.
2. 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.
3. John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition edition, 2007.
4. M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. *Principles of Distributed Systems*, pages 401–414, 2007.
5. E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Proceedings of Distributed Computing*, pages 117–131, 2004.
6. Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2) :190–222, 1983.
7. 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.
8. J.M. Mellor-Crummey. Concurrent queues : Practical fetch-and-ϕ algorithms. Technical report, Technical Report 229, Computer Science Department, University of Rochester, 1987.
9. 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.
10. 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.
11. S. Prakash, Y. Lee, and T. Johnson. Non-blocking algorithms for concurrent data structures. Technical report, University of Florida, 1991.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.