RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

CLRC : un Support à Mémoire Partagée Répartie pour des Grappes de Stations Inter-connectées

Luciana Arantes, Pierre Sens

Université de Paris 6,
LIP6, 4 Place Jussieu, 75252 Paris Cedex 05 - France
{Prénom.Nom}@lip6.fr

Résumé
Dans cet article, nous proposons une mémoire partagée répartie (MPR), basée sur le modèle de cohérence relâchée paresseuse (LRC: Lazy Release Consistency), adaptée à des plates-formes composées d'un ensemble de stations de travail interconnectées. Cependant, nous considerons que le coût des communications entre des machines de grappes différentes peut être beaucoup plus élevé que celui entre des machines d'une même grappe. Par conséquent, une MPR sur une telle plate-forme doit réduire le coût des communications inter-grappes en privilégiant un partage de données au sein d'une même grappe. Dans notre approche, nous exploitons la localité des accès aux variables partagées au sein d'une même grappe ainsi que les informations de contrôle du protocole LRC pour minimiser le coût des échanges de messages entre les grappes. Pour cela, nous introduisons une gestion de cache au niveau de chaque grappe et nous adaptons les mécanismes de synchronisation à l'architecture multi-grappes.

Mots-clés : Mémoire partagée répartie, Cache, Grappes,cohérence relâchée paresseuse

1. Introduction

Les mémoires partagées réparties (MPR) fournissent l'abstraction d'une mémoire partagée au-dessus de systèmes ayant des mémoires physiquement distribuées. Cependant, la plupart des MPRs ont été implantées sur des réseaux de stations de travail ou des réseaux multi-processeurs symétriques (SMPs) et ne sont généralement adaptées qu'à des réseaux locaux. Nous proposons d'étendre le concept de MPR à un ensemble de grappes de stations inter-connectées (plate-forme multi-grappes). Cette approche permet d'exploiter un plus grand nombre de machines ou des machines distantes sous-utilisées. Cette solution est séduisante car elle offre aux applications une capacité de traitement plus importante à un faible coût.
Toutefois, les plates-formes multi-grappes ont des caractéristiques physiques qui peuvent limiter fortement les performances des applications. La latence entre les machines des grappes différentes peut être beaucoup plus élevée que celle entre machines d'une même grappe, et le débit est souvent plus faible. Le principal défi d'une MPR sur une telle plate-forme est donc de réduire le coût des communications inter-grappes. Elle doit exploiter la localité dans les références aux données faites par les processeurs au sein d'une même grappe afin de privilégier le partage de données parmi ces processeurs.
Nous proposons d'implanter une MPR multi-grappes en utilisant le modèle de cohérence relâchée paresseuse (LRC : Lazy Release Consistency). En relâchant le modèle de cohérence de mémoire, LRC réduit le volume de communication entre les processeurs. La propagation des modifications faites par un processeur n'a lieu que lors de certaines opérations de synchronisation. Le protocole de maintien de la cohérence relâchée paresseuse, qui a été initialement défini dans TreadMarks, est reconnu comme l'un des plus efficaces pour les réseaux locaux. Cependant, notre choix pour le protocole LRC ne repose pas uniquement sur ces performances mais aussi sur le fait que chaque processeur maintient en mémoire les mises à jour de données partagées même après les avoir appliqués sur leurs copies locales. Cet historique des mises à jour est fortement exploité dans notre solution.
Pour exploiter la localité des références au sein d'une même grappe, nous avons défini le *cache intra-grappe* qui permet de réduire la latence inter-grappe qui est le principal goulot d'étranglement pour des applications parallèles sur une plate-forme multi-grappes. Le cache est composé de deux autres caches :

le *cache inhérent* et le *cache étendu*. Le premier exploite les informations du protocole LRC présentes dans chaque grappe pour réduire le volume de données échangées. Le deuxième permet de collecter les informations de cohérence des autres grappes. Nous avons aussi adapté les mécanismes de synchronisation à l'architecture multi-grappes. Notre MPR est appelée **CLRC (Cluster Lazy Release Consistency)**.
La suite de cet article est organisée de la manière suivante. Dans la section 2, nous présentons les principes du protocole LRC. La section 3 décrit les extensions que nous avons ajoutées à TreadMarks pour implanter le cache intra-grappe et adapter ses algorithmes de barrières et verrous à des plates-formes multi-grappes non-uniformes. Dans la section 4, une discussion sur les résultats obtenus avec un prototype de la MPR CLRC est présentée. Nous évoquons les autres approches dans le domaine des MPRs dans la section 5. Enfin, la section 6 conclut sur nos travaux.

2. Protocole de maintien de la cohérence relâchée paresseuse

Dans le modèle de *cohérence relâchée* [1], les accès normaux à la mémoire sont isolés des synchronisations. Les synchronisations imposent un ordonnancement des accès à la mémoire partagée et peuvent être divisées en séquences d'acquisition et de libération. Une opération d'acquisition permet d'obtenir l'accès à une variable partagée tandis que la libération transmet le droit d'accès. Les modifications effectuées par un processeur sur les variables partagées sont propagées aux autres uniquement lorsque celui-ci exécute l'opération de libération.
La ***cohérence relâchée paresseuse*** (LRC: Lazy Release Consistency), originalement définie pour la MPR TreadMarks de l'Université de Rice [1], adopte les mêmes principes que la cohérence relâchée, mais la mise à disposition des modifications aux processeurs est reportée jusqu'à ce que ces derniers exécutent une opération d'acquisition. L'exécution des processus est divisée en intervalles. Un nouvel intervalle commence à chaque opération d'acquisition ou de libération.
L'ordre partiel des intervalles est contrôlé en utilisant une horloge vectorielle [9, 7] associée à chaque opération de synchronisation. Chaque processeur maintient une horloge logique locale représentée par un vecteur d'entiers de N entrées, où N est le nombre total de processeurs du système. Un processeur P_j contrôle les opérations de synchronisation qu'il effectue en utilisant la *jème* entrée du vecteur. Les autres entrées mémorisent la vision que le processeur a des mises à jour des autres sites.
Lors d'une opération d'acquisition, le processeur P_j doit recevoir du processeur P_r qui a effectué la libération, tous les intervalles dont les vecteurs d'horloges sont supérieurs à son horloge locale. Pour cela, P_j envoie la valeur courante de son horloge vectorielle au processeur P_r. Ce dernier renvoie alors la liste des intervalles et des pages modifiées qu'il connaît, mais que P_j ignore. La liste des pages modifiées est envoyée sous la forme de notifications d'écriture (*write-notices*). Une notification d'écriture est une structure, créée à la fin d'un intervalle, qui contient l'identification d'une page qui a été modifiée pendant l'intervalle. Les modifications sont elles-mêmes envoyées plus tard soit sous forme de *diffs* (comparaison mot à mot entre une copie de la page et sa dernière version), soit directement sous forme de pages modifiées. En recevant les notifications d'écriture, le processeur demandeur invalide les copies locales des pages correspondantes. Le prochain accès à l'une de ces pages provoquera un défaut de page et le processeur demandera alors les modifications. Dans le cas de *diffs*, l'acquéreur demande tous les *diffs* qu'il ne possède pas encore, puis il les applique dans l'ordre causal des intervalles.
La plupart des MPRs basées sur le modèle de cohérence relâchée paresseuse offrent deux variables de synchronisation : les ***barrières*** et les ***verrous***. Les barrières sont utilisées pour sérialiser l'exécution de l'application tandis que les verrous contrôlent l'accès à des régions critiques. Les opérations sur les verrous et les barrières peuvent être réalisées à l'aide des opérations d'acquisition et de libération : un verrouillage correspond à une acquisition, un déverrouillage à une libération, l'arrivée à une barrière peut être modélisée comme une libération tandis que son départ peut être représenté par une acquisition.

3. CLRC : Une MPR sur des plates-formes multi-grappes

Le fait que l'envoi d'un message entre deux machines de grappes différentes (connexion inter-grappe) soit plus coûteux qu'entre deux machines appartenant à la même grappe (connexion intra-grappe) limite les performances des applications au-dessus d'une plate-forme multi-grappes. Ce coût comprend la latence et le débit de communication. Toutefois, la latence est considérée comme le principal goulot d'étranglement

puisqu'elle est dépendante de l'éloignement entre les machines, et moins sensible aux avances technologiques que le débit. Par conséquent, nos choix lors de la conception de la MPR CLRC sont surtout destinés à réduire l'effet de la latence inter-grappe sur les performances des applications.
Nous avons modifié la MPR TreadMarks pour y introduire la notion d'un cache logique par grappe (cache intra-grappe) et adapter les mécanismes de synchronisation (gestion de barrière et de verrou) à la hiérarchie de latence de la plate-forme.

3.1. Le cache intra-grappe

Le but du *cache intra-grappe* est de faire en sorte que toutes les données stockées dans la mémoire des processeurs au sein d'une même grappe puissent être utilisées au maximum par les processeurs de cette grappe, réduisant alors le coût des communications entre machines de grappes différentes et, par conséquent, le temps qu'un processus attend pour obtenir une copie à jour d'une donnée partagée. Cette approche repose sur le principe de localité dans les références aux données partagées faites par les processus d'une même grappe que nous appelons *localité intra-grappe*.
Un cache intra-grappe est composé d'un *cache inhérent* et d'un *cache étendu* [2]. Le premier utilise directement les informations déjà maintenues en mémoire par le protocole LRC tandis que le deuxième stocke, dans un processus additionnel dédié, les informations sur les mises à jour des grappes distantes, non identifiables à l'aide du protocole LRC de base.

3.1.1. Cache inhérent

Dans le protocole LRC standard, toutes les modifications de pages (i.e., *diffs*) sont gardées en mémoire même lorsqu'elles ont été appliquées sur les copies locales des pages. Ces *diffs* sont maintenus car d'autres processus peuvent les réclamer. Dans notre approche multi-grappes, nous les utilisons pour réduire le volume de *diffs* échangés entre deux grappes. Quand un processeur envoie une requête à un processeur qui se trouve sur une grappe distante, le processeur demandeur ne recevra de la grappe distante que les mises à jour non détenues par les autres processeurs de la même grappe. Le processeur demandeur obtiendra les mises à jour locales qui lui manquent en interrogeant directement les processeurs locaux.
En appliquant le principe des chemins de causalité des intervalles, nous avons modifié le protocole LRC. Lors d'une requête des mises à jour d'une page (*diffs*), le processeur demandeur a la connaissance de l'ordre partiel des mises à jour, liées à cette page, qui lui manquent, ainsi que les processeurs qui les ont effectuées. Toutefois, dans le protocole LRC original, la requête pour les mises à jour est toujours adressée au dernier processeur qui a modifié la page, i.e., le responsable de la dernière mise à jour du chemin de causalité. Dans le cas de CLRC, si ce processeur appartient à une grappe distante, le processeur demandeur divise chaque chemin de causalité en deux parties. Il identifie le dernier processeur local de sa grappe ayant réalisé une mise à jour, et récupère de ce dernier les mises à jour que celui-ci possède et n'obtient du processeur distant que les mises à jour qui correspondent à la deuxième partie du chemin de causalité. De cette manière, le volume des données échangées entre les grappes peut être fortement réduit. De plus, cette approche parallélise le traitement des requêtes, ce qui peut réduire le temps total nécessaire à la mise à jour d'une page. Naturellement, le processeur demandeur applique les mises à jour reçues en respectant l'ordre établi par le chemin de causalité des intervalles.
Le principe du *cache inhérent* est illustré dans la figure 1. Les processeurs P_1 et P_2 appartiennent à la même grappe C_1, tandis que P_3, P_4 et P_5 font partie de la grappe C_2. Les processus utilisent le verrou 0 pour protéger l'accès à la variable y. La notation $\mathtt{lock}^j(0)$ et $\mathtt{unlock}^j(0)$ désignent respectivement la *jème* acquisition ou libération du verrou 0 faite par un processeur.
Par exemple, si après avoir acquis le verrou 0, ($\mathtt{lock}^5(0)$), P_3 tente d'écrire sur y, un défaut de page sera généré. P_3 requiert alors tous les *diffs* que lui manquent correspondant à la page contenant la variable y. Dans le protocole LRC original, P_3 s'adresserait à P_1 pour lui demander ces *diffs* (les *diffs* créés entre les intervalles $\mathtt{lock}^1(0)$-$\mathtt{unlock}^1(0)$ et $\mathtt{lock}^4(0)$-$\mathtt{unlock}^4(0)$). Cependant, à ce moment, P_3 sait que P_4 a été le dernier processeur de sa grappe à actualiser la variable y. Par conséquent, P_3 demandera à P_4 la liste de *diffs* jusqu'à intervalle $\mathtt{lock}^3(0)$-$\mathtt{unlock}^3(0)$ (quand P_4 a cédé le verrou à P_1) et à P_1 le restant des *diffs* (les diffs créés dans l'intervalle $\mathtt{lock}^4(0)$-$\mathtt{unlock}^4(0)$).

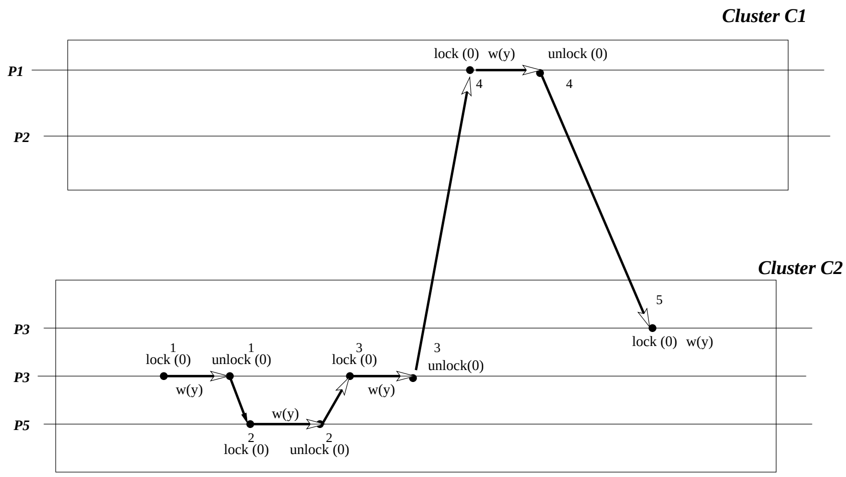

FIG. 1 – *Le Cache inherent*

3.1.2. Cache étendu

Le *cache inhérent* repose sur le fait que chaque processus peut identifier, lors d'un défaut de page, les relations de causalité des *diffs* qui lui manquent. Cependant, il est possible qu'un processus ne sache pas qu'un deuxième processus de sa propre grappe a déjà demandé ces mêmes *diffs*.

Cela arrive, par exemple, si l'application présente un comportement avec *"un seul producteur et plusieurs consommateurs"* et si ces derniers appartiennent à la même grappe. Après qu'un premier processus consommateur ait demandé une mise à jour distante, celle-ci est disponible dans sa mémoire, mais les autres consommateurs n'en sont pas au courant. Un deuxième exemple concerne le problème du *"faux partage"*, i.e. quand deux processus demandent le même *diff* sur une page contenant deux variables indépendantes. Si les deux processus se trouvent dans la même grappe, quand le premier demande à un processeur distant une page faussement partagée, le deuxième processus pourrait profiter de cette copie disponible s'il savait où elle se trouve. Pour cela, nous avons introduit dans chaque grappe un processus supplémentaire qui maintient dans sa mémoire toutes les informations, en provenance des grappes distantes, qui ne sont pas identifiables dans le cache inhérent. Ce processus, que nous appelons *processus cache*, est dédié pour garder des *diffs* distants et n'exécute pas le code de l'application.

Le mécanisme du cache étendu fonctionne de la manière suivante. Lorsqu'un processus réclame un *diff* appartenant à une grappe distante, il génère deux messages : un premier vers son processus cache local et un second vers le processus distant possédant le *diff*. Si le processus cache a déjà les *diffs*, il les renverra directement ; sinon les *diffs* proviendront du processus distant. Lorsque le processus distant reçoit la requête contenant la demande des *diffs*, il envoie les *diffs* au processus demandeur ainsi qu'à son processus cache.

La figure 2 illustre le fonctionnement du cache étendu dans une configuration à deux grappes avec cinq processus P_i et deux processus dédiés au cache EC_i. Supposons que le processus P_1 (le producteur) modifie une variable x qui est lue ensuite par P_3, P_4, P_5 (les consommateurs). Lorsque P_3 tente de lire x, un défaut de page est généré. Une requête $(\mathtt{req}_\mathsf{x}^1)$ est alors envoyée au producteur P_1. Celui-ci appartenant à une grappe distante, P_3 envoie également sa requête au cache de sa grappe EC_2. Quand la requête arrive sur P_1, ce dernier renvoie à P_3 les *diffs* correspondants ($diffs_\mathsf{x}$). De plus, comme c'est la première fois que P_1 reçoit une demande pour $diffs_\mathsf{x}$, il les envoie également à EC_2. P_4 et P_5 essayant, eux aussi, de lire la variable x, ils iront demander les mêmes $diffs_\mathsf{x}$ au processus P_1 ainsi qu'à EC_2. EC_2 ayant déjà reçu une copie de $diffs_\mathsf{x}$, il pourra directement renvoyer la réponse à P_4 et P_5, qui n'attendront pas la réponse de P_1. Quand celle-ci arrivera, elle sera ignorée.

Pool de requêtes de diffs en attente

En considérant que les connexions inter-grappes sont plus lentes que les connexions intra-grappes, il se peut qu'un processus cache reçoive une ou plusieurs requêtes pour le même ensemble de *diffs* avant

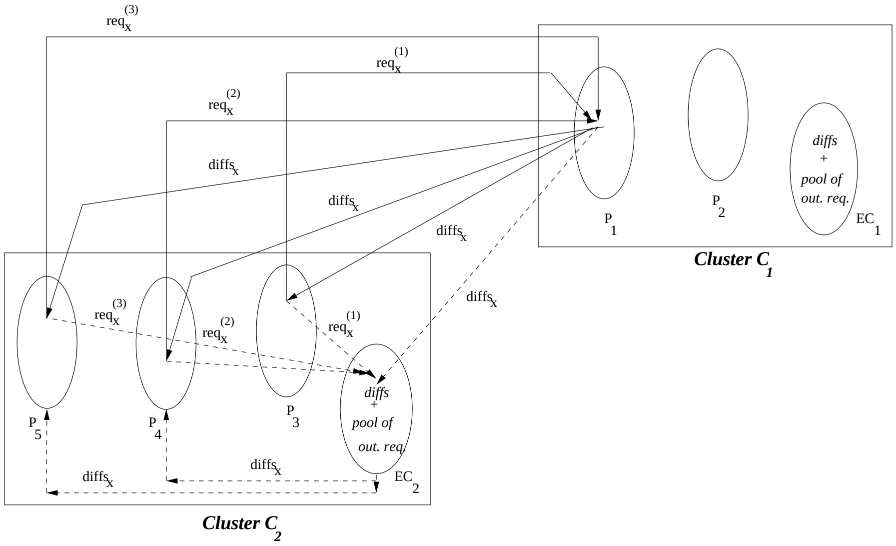

FIG. 2 – *Le cache étendu*

l'arrivée des *diffs* eux-mêmes. Pour cette raison, le processus cache stocke dans une structure interne (pool de requêtes de *diffs* en attente) toutes les requêtes pour des *diffs* non encore reçus. Lorsque le processus cache recevra les *diffs*, il parcourera ce pool afin de vérifier quelles sont les requêtes qui peuvent être satisfaites. Par exemple, reprenons la même figure 2. Si les processus P_4 et P_5 émettent leur requêtes, (req_x^2 et req_x^3, respectivement) quand les *diffs* demandés (*diffs*$_x$) sont en transit, le processus cache EC_2 sauvegardera les requêtes dans le pool. Quand EC_2 recevra *diffs*$_x$, il parcourera le pool et répondra alors aux requêtes req_x^2 et req_x^3.

Agrégation de requêtes de diffs

Pour les requêtes de *diffs*, nous appliquons une optimisation supplémentaire. Avec l'algorithme à écrivains multiples, une page peut avoir plusieurs copies, qui sont mises à jour simultanément. Dans ce cas, lors d'un défaut de page, un processus demande en parallèle les *diffs* de la page à chaque processus possédant une copie (i.e., les derniers écrivains). Cependant, si les écrivains sont situés dans la même grappe distante, plusieurs requêtes seront envoyées au processus cache de cette grappe. Pour limiter l'engorgement de ce processus, l'émetteur assemble les requêtes dans un seul message. Cette agrégation permet de réduire le trafic intra-grappe et de limiter la contention du processus cache.

Pré-chargement du cache étendu

L'efficacité du cache étendu est directement liée aux *diffs* maintenus par le processus cache. Le pré-chargement des *diffs* permet d'augmenter le taux de succès du cache étendu en anticipant les futures demandes. De nombreuses études ont démontré que la localité temporelle permet de concevoir des stratégies de pré-chargement efficaces.

Nous avons donc conçu une heuristique de pré-chargement où chaque processus compte le nombre de demandes de *diffs* émanant de chaque grappe distante entre deux appels à une barrière. Lors du deuxième appel à la barrière, si plusieurs processus d'une même grappe distante ont réclamé des *diffs* sur une même page, chaque processus qui a modifié le page envoie au cache étendu de la grappe distante les nouveaux *diffs* concernant cette page, sans attendre qu'un processus de la grappe distante les réclame. Ainsi, les derniers *diffs* distants sur les pages souvent réclamées deviennent accessibles plus rapidement. Le nombre de *diffs* pré-chargés est limité par un paramètre de configuration défini par l'utilisateur.

La figure 3 illustre le concept du mécanisme de pré-chargement et l'heuristique adoptée. Elle montre un scénario avec un processus producteur, P_1, qui modifie la variable x et trois processus consommateur de x (P_3, P_4 et P_5) appartenant à une même grappe distante. Pendant l'intervalle défini entre la barrière

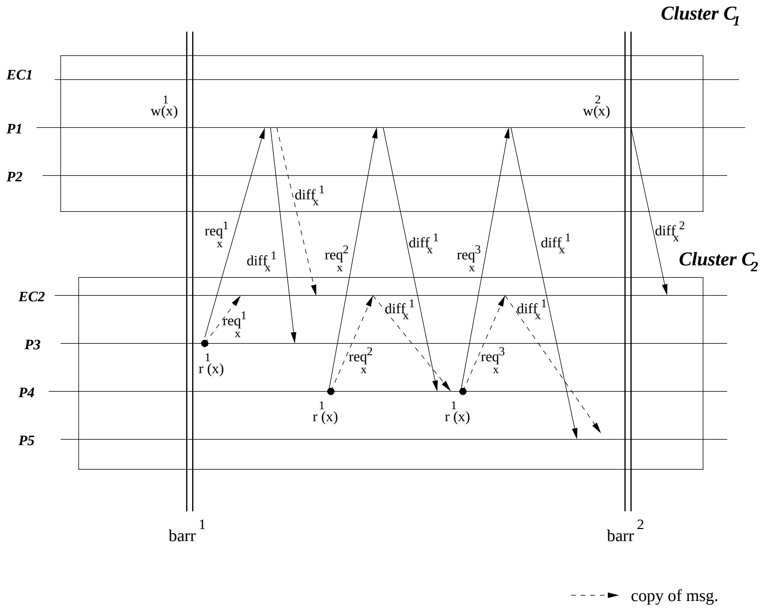

FIG. 3 – *Pré-chargement de diffs : scénario avec 1 producteur et 3 consommateurs*

barr^1 et barr^2, le processus cache EC_2 n'a pas pu satisfaire les requêtes de P_4 et P_5 pour $diffs_x^1$, qui en fait ont été satisfaites par le processus distant P_1. Cependant, pendant cet intervalle, P_1 a pu détecter que la page page_x avait été demandée par plusieurs processus de la grappe C_2. Par conséquent, lors de l'appel à la barrière barr^2, P_1 enverra à EC_2 les diffs ($diffs_x^2$), correspondant aux dernières modifications sur la page_x (w_x^2 dans la figure). De cette façon, si dans un future proche, les processus de C_2 répètent leur comportement de consommateur de la variable x, leurs requêtes seront satisfaites plus rapidement par EC_2.

3.2. Opérations de synchronisation

Les caches permettent de diminuer le coût des échanges inter-grappe. Cependant, pour avoir une gestion plus efficace, il est important de modifier l'implantation des mécanismes de synchronisation pour les adapter à la structure multi-grappes. A cette fin, nous proposons les deux optimisations suivantes.

3.2.1. Barrières parallèles en grappe

Lors des barrières, TreadMarks propage des informations de cohérence (pages modifiées et intervalles) de manière centralisée. Un gestionnaire de barrière regroupe les informations et les diffuse à tous les processeurs. Les processus de l'application sont alors bloqués tant qu'ils n'ont pas reçu le message du gestionnaire. Ce protocole en deux phases présente une forte latence ce qui limite grandement ses performances lorsque le nombre de noeuds devient important. Nous avons donc conçu et implanté un protocole moins centralisé ("en grappe") qui permet de réduire la latence des barrières. Arrivé à une barrière, chaque processus envoie à tous les processus cache, y compris celui de sa propre grappe, les informations de cohérence. Dès qu'un processus cache a collecté les informations de tous les processus, il diffuse aux processus de sa grappe les informations de cohérence. Pour des raisons de performance ce message est diffusé en utilisant IP-multicast. En fait, l'ensemble des processus cache joue le rôle du gestionnaire de barrière.

3.2.2. Migration des gestionnaires de verrous

Dans TreadMarks, tous les verrous ont, parmi les processus de l'application, un gestionnaire fixé statiquement au démarrage de l'application. Toutes les demandes d'acquisition sont envoyées au gestionnaire correspondant. Ceci peut poser de graves problèmes de performance dans une approche multi-grappes. Ainsi, même si le verrou est possédé par un processus situé dans la même grappe, il peut arriver que le gestionnaire soit localisé sur une autre grappe. Pour remédier à ce type de situation, nous avons introduit la migration des gestionnaires de verrous. Pendant l'exécution, chaque gestionnaire trace les demandes de verrous. Si le nombre de requêtes successives émises d'une même grappe distante dépasse un seuil et que le nombre de requêtes successives en provenance des processus de la grappe locale est en dessous d'un second seuil, le gestionnaire est déplacé dans la grappe distante. L'utilisation de deux seuils permet de limiter un effet de ping-pong où un gestionnaire ne cesserait de migrer.

4. Evaluation des performances

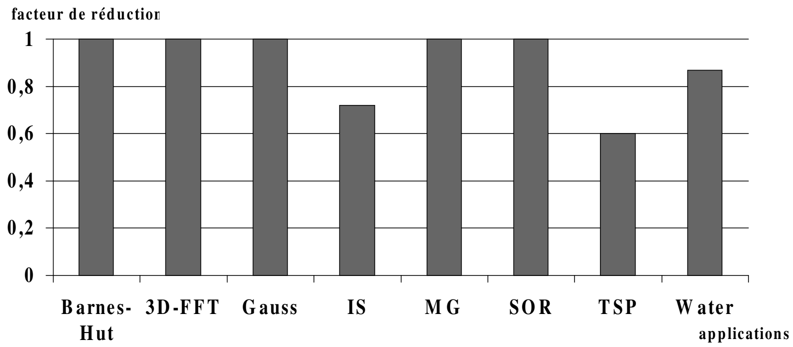

FIG. 4 – *Effet du cache inhérent sur la taille des diffs*

L'efficacité de notre solution a été évaluée en utilisant neuf applications de granularité grosse, moyenne et faible. La plupart appartiennent à des benchmarks SPLASH [14, 16] ou NAS [3], ou sont des applications appropriées pour tester les performances des applications sur une MPR. En voici la liste : Red-Black SOR, TSP et Gauss (distribuées avec TreadMarks); Water and Barnes-Hut du benchmark SPLASH; IS, 3D-FFT, EP et MG du benchmark NAS.
Nous avons fait nos tests sur un ensemble de machines PC, exécutant Linux, et reliées par un réseau Ethernet à 100 Mbit/s. Les machines possèdent soit un processeur Celeron cadencé à 433 MHz avec 128 Mégaoctets de mémoire soit un processeur K6 cadencé à 233 MHz avec 64 Mégaoctets.
Nous avons divisé les machines en groupes pour former deux grappes logiques distinctes. Chaque grappe est composée au plus de cinq machines Celeron, plus rapides, dédiées au code applicatif et un K6 dédié au processus cache.
Pour simuler la latence entre ces grappes logiques, les communications inter-grappes transitent par des processus situés sur des machines supplémentaires. En faisant varier le nombre de machines intermédiaires, nous avons simulé deux latences inter-grappes, nommées *lat1* et *lat2*. Lors de la simulation avec la latence *lat1*, le temps moyen pour obtenir un *diff* distant est *doublé*, tandis que pour la latence *lat2* il est multiplié

par *dix*.

4.1. Cache inhérent

En limitant le nombre d'informations demandées aux grappes distantes (*diffs*) lors des défauts de page, le cache inhérent est directement responsable de la réduction de la taille des messages échangés entre les grappes.

Dans la figure 4, nous indiquons le rapport: *nombre d'octets des messages de diffs inter-grappes par message échangés en ativant le cache inhérent / nombre d'octets des messages de diffs inter-grappes par message sans le cache inhérent*. Le nombre de messages échangés dans les deux cas est le même. Aucune latence n'est simulée pour ces mesures. Nous pouvons remarquer que les applications présentant des données migratoires telles que IS, TSP et Water profitent vraiment de cette réduction (rapport inférieur à 1).

4.2. Cache étendu

Pour évaluer l'efficacité des caches étendus, nous avons mesuré le pourcentage de requêtes satisfaites par les processus cache par rapport au total des requêtes de *diffs* distants ("*diff request hit factor*").

Appl.	**Nb. de processeurs par grappe**		
	3 procs.	**4 procs.**	**5 procs.**
Barnes-Hut	22.2%	27.3%	27.3%
EP	0%	0%	0%
3D-FFT	10.3%	23.4%	31.2%
Gauss	58.5%	65.6%	77.6%
IS	38.7%	48.4%	50.3%
MG	9.3%	16.5%	22.5%
SOR	0%	0%	0%
TSP	31.4%	39.4%	42.0%
Water	34.2%	38.0%	49.2%

TAB. 1 – *Efficacité du cache*

Le Tableau 1 présente ces résultats lorsque nous avons fait varier le nombre de nœuds de chaque grappes (de trois à cinq processus applicatifs).

Nous observons que plus la taille des grappes augmente, plus l'efficacité du cache est importante. Les applications avec un comportement producteur/consommateur ou celles qui présentent des pages faussement partagées tirent profit du cache étendu (e.g. Gauss, MG, TSP, Water, IS). Néanmoins, dans le cas des applications EP et SOR, un processus d'une grappe ne bénéficie pas des *diffs* distants demandés auparavant ou simultanément par un deuxième processus de la même grappe. Par conséquent, le cache étendu n'a aucune efficacité sur ces deux applications, contrairement à ce qui arrive avec les autres applications. En fait, cela arrive parce que ces applications ne présentent pas de localité intra-grappe de références dans les accès aux données partagées.

La localité intra-grappe est donc exprimée par le "*diff request hit factor*". Toutefois, ce taux comprend le pourcentage de requêtes satisfaites promptement ("*prompt diff request hit factor*"), ainsi que le pourcentage de requêtes satisfaites plus tard avec le mécanisme du *pool de requêtes de diff en attente* ("*delayed diff request hit factor*"). La figure 5 illustre l'efficacité du cache et de la stratégie de pré-chargement. La latence inter-grappe considérée pour cette figure est égale à *lat1* et le nombre de processus applicatifs par grappe égal à cinq. Le nombre de pages dont les *diffs* peuvent être pré-chargés a été limité à 20. Des applications comme Gauss et MG dont les processus font des accès concurrents à la même page tirent un bénéfice du mécanisme du *pool de requêtes de diff en attente* tandis que des applications dont les accès à une page partagée ne sont pas concurrents mais protégés par un même vérrou, comme TSP, n'en profitent pas.

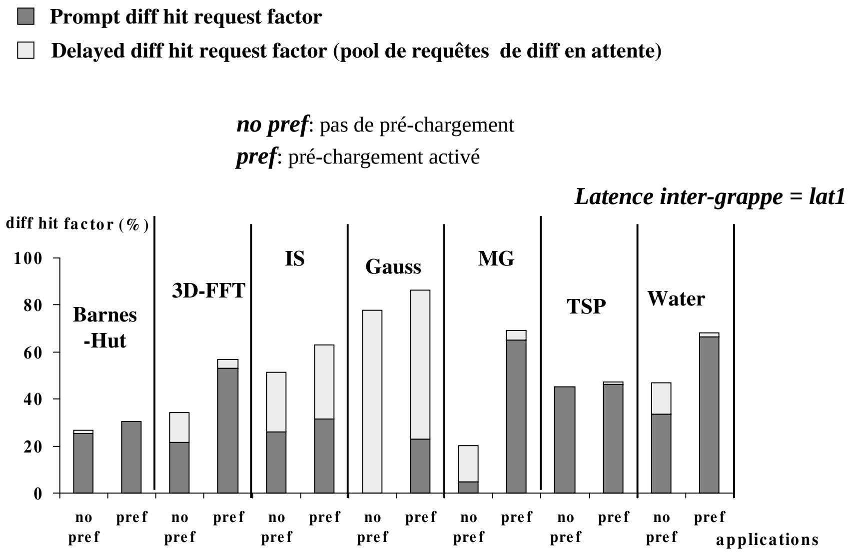

FIG. 5 – *"Diff hit factor" du cache étendu*

Le mécanisme de pré-chargement, en général, augmente le taux de requêtes satisfaites par un processus cache pour la majorité des applications. En outre, il réduit l'effet du *pool de requêtes de diff en attente.* Il est responsable aussi, pour la plupart des applications (e.g. 3D-FFT, IS, Gauss, MG, Water), de l'augmentation du pourcentage de requêtes satisfaites promptement par un processus cache ("*prompt diff request hit factor*").

4.3. Temps d'attente pour un diff distant

Nos tests ont montré que pour la plupart des applications, à l'exception de 3D-FFT qui est une application présentant un volume très grand d'échanges de message entre les processeurs, le cache intra-grappe permet de réduire le temps d'attente d'un *diff* distant.

La figure 6 montre le rapport : *temps moyen d'attente pour obtenir un diff distant en utilisant TreadMarks / temps moyen d'attente pour obtenir un diff distant en utilisant CLRC.* Nous avons considéré cinq processus applicatifs par grappe et les deux latences inter-grappes. Nous pouvons remarquer que pour toutes les applications, sauf 3D-FFT, que le rapport est supérieur à 1.

4.4. Performances des applications

La figure 7 montre l'avantage d'exécuter les applications sur deux grappes de cinq stations par rapport à une grappe de cinq stations, considérant les deux latences inter-grappes *lat1* et *lat2.* Le gain de performance est illustré par le rapport : *temps d'exécution de l'application sur une grappe / temps d'exécution de l'application sur deux grappes* pour les MPRs TreadMarks et CLRC. Nous remarquons que pour les applications Gauss, MG et TSP notre approche apporte un gain de performance lorsque la latence inter-grappe *lat1* est simulée.

Cependant, lorsqu'une plus forte latence est simulée (*lat2*), les performances ont été décevantes (à l'exception de EP qui est une application à gros grain). Bien que le cache intra-grappe réduise le temps d'attente d'un *diff* distant, nos tests nous portent à croire que les opérations de synchronisation minimisent l'effet

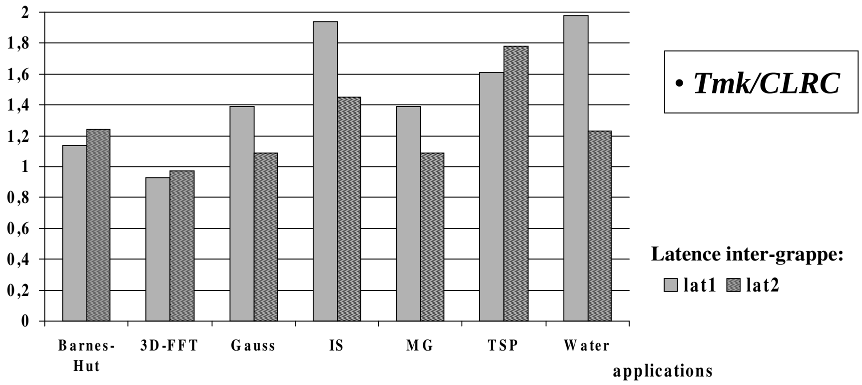

FIG. 6 – *Temps d'attente pour un diff distant*

de ce gain. Une solution serait de restructurer les applications afin qu'elles puissent utiliser des primitives de synchronisation plus flexibles (e.g. plusieurs lecteurs, ayant acquis le même verrou en lecture, peuvent lire concurremment la même donnée) et éviter les synchronisations globales.

5. Travaux apparentés

Plusieurs travaux ont proposé d'étendre des MPR à des réseaux de machines SMP. Les principaux projets dans ce domaine sont **Shasta** de l'Université de Rutgers [13], **Cashmere-2L** de l'Université de Lancaster [15], **HLRC-SMP** de Princeton [12], **SoftFlash** de Standford [6] et **MGS** du MIT [17]. Ces approches profitent des protocoles de cohérence cache matériels au sein de chaque nœud tandis que des protocoles implantés de façon logicielle gèrent les échanges entre nœuds. Ainsi Hu et al. présentent dans [5] une version de TreadMarks pour réseaux SMPs. Cashmere-2L implémente un protocole à écrivains multiples tandis que Shasta supporte une mémoire partagée avec une granularité plus fine. HLRC-SMP propose une extension du protocole HLRC aux réseaux de machines SMP. Notre prototype présente des similarités certaines avec ces approches. En effet au sein d'un même cluster toutes les informations de cohérence en provenance de l'extérieur sont disponibles pour tous les nœuds du clusters. La différence est que, dans notre cas, cette disponibilité est assurée de façon logicielle.
Le pré-chargement des *diffs* pour des protocoles LRC a été proposé par de nombreux auteurs [10, 4, 8] dans le but d'absorber en partie la latence des communications. Ces travaux visent à rapprocher les données des processeurs qui les utilisent. Comme dans notre approche, les opérations de synchronisation, particulièrement les barrières, sont utilisées pour diviser l'exécution des programmes en intervalles sur lesquels sont définis des historiques d'accès. Dans ces approches, les données pré-chargées sont stockées dans la mémoire des processeurs susceptibles d'exiger ces données dans un futur proche. Cela diffère de notre approche où un processus dédié, le processus cache, stocke toutes ces données. Ainsi, nous exploitons l'architecture multi-cluster : tous les processus d'une même grappe profitent des données pré-chargées du

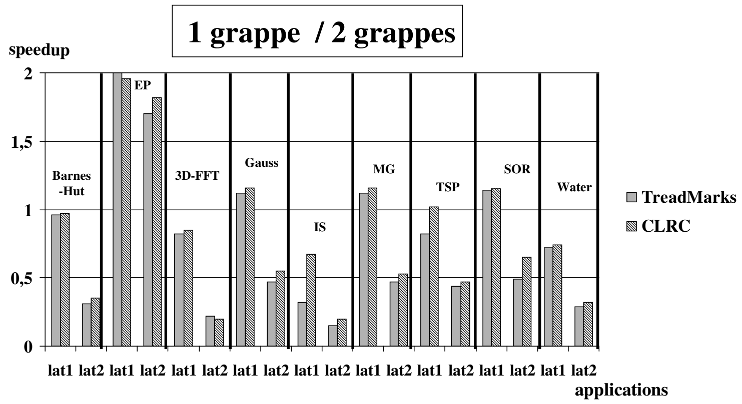

FIG. 7 – *Gain des applications*

cache.
Plaat et al. (projet Albatross) [11] proposent la restructuration des applications afin de les adapter à des plates-formes multi-grappes avec de latences et débits non-uniformes. Des expérimentations sur plusieurs applications parallèles du benchmark NAS et SPLASH ont montré l'efficacité de leur solution. Cependant, contrairement à notre approche, les optimisations ont été introduites dans le source de chaque application et pas dans le support à mémoire partagé répartie, comme nous avons fait dans CLRC.

6. Conclusions

Nous avons étendu la mémoire MPR TreadMarks pour une meilleure adaptation à des plates-formes composées de grappes de stations interconnectées. TreadMarks est un support à mémoire partagée répartie, développé par l'Université de Rice, qui est connu pour ses bonnes performances principalement grâce à son protocole de maintien de la cohérence relâchée paresseuse. Une des difficultés de son adaptation à une plate-forme multi-grappe réside dans le goulot d'étranglement que les communications inter-grappes représentent. Dans la MPR CLRC, les machines d'une grappe partagent leurs informations liées au protocole de cohérence ainsi que les copies à jour de données partagées afin de réduire le coût d'une requête à une donnée distante. Des expérimentations sur plusieurs applications parallèles ont montré que ces dernières présentent une *localité intra-grappe* et que CLRC exploite cette localité en réduisant le temps d'attente pour un *diff* distant. Cette réduction assure l'efficacité de notre approche. Cependant, même si nous avons proposé des algorithmes pour adapter les opérations de synchronisation à l'architecture de la plate-forme, ceux-ci ne permettent pas de minimiser l'impact de la latence inter-grappe, responsable pour les mauvaises performances des applications à grain petit ou moyen. Dans ces cas là, des stratégies plus flexibles, comme un modèle transactionnel ou des accès concurrents réalisés par plusieurs processus lecteurs sur des données protégées par le même verrou, doivent alors être adoptées. D'autre part, il ne faut pas oublier que ces applications sont représentatives du comportement d'accès des applications sur les MPRs, mais ne sont pas adaptées pour des plates-formes avec des latences non-uniformes.

Bibliographie

1. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of worksations. *IEEE Computer*, 29(2):18–28, February 1996.
2. L. Arantes, P. Sens, and B. Folliot. The impact of caching in a loosely-coupled clustered software DSM system. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 27–34, Chemintz (Germany), November 2000.
3. D. Bailey, J. Barton, T. Lansinski, and H. Simon. The NAS parallel benchmark. Technical Report Technical Report 103863, NASA, July 1993.
4. R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software DSMs. In *Proceedings of the 1998 International Conference on Supercomputings*, July 1998.
5. A. Cox, Y. Hu, H. Lu, and W. Zwaenepoel. OpenMP on networks of SMP. In *13th International Parallel Processing Symposium*, April 1999.
6. A. Erlichson, N. Nuckolls, G. Chesson, and J. Henessy. SoftFLASH: Analysing the performance of clustered distributed virtual shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, October 1996.
7. C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, July 1991.
8. M. Karllson and P. Stenstrom. Effectiveness of dynamic prefetching in multiple-writers distributed virtual shared memory systems. *Journal of Parallel and Distributed Computing*, 42(7), July 1997.
9. F. Mattern. Virtual time and global states in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, Elsevier (Holland), October 1989.
10. T. Mowry, C. Chan, and Lo. A. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, February 1998.
11. A. Plaat, H. Bal, and R. Hofman. Sensitivity of parallel applications to large differences in bandwidth an latency in two-layer interconnects. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, February 1999.
12. R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proceedings of the 4th Symposium on High Performance Computer Architecture*, February 1998.
13. D. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, February 1998.
14. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.
15. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis, Parhasarathy, and M. Scott. Cashmere-2L: software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.
16. S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological characterizations. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, June 1995.
17. D. Yeung, J. Kubiatowicks, and A. Agarwal. Mgs: A multi-grain shared memory system. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–55, April 1996.