CFSE'1
Rennes, 8 juin - 11 juin 1999
1ère Conférence Française sur les Systèmes d'Exploitation

Une Mémoire Partagée Répartie Extensible basée sur la Cohérence Relâchée Paresseuse

Luciana Arantes*, Bertil Folliot et Pierre Sens.
LIP6 - Université Paris 6
4, Place Jussieu 75252 Paris Cedex 05.
Adresse électronique: *[arantes,folliot,sens]@src.lip6.fr*

Résumé

Dans cet article, nous proposons une implantation extensible du protocole de maintien de la cohérence relâchée paresseuse (LRC pour Lazy Release Consistency). Les implantations existantes du protocole de maintien de la LRC utilisent des vecteurs de temps, qui dépendent du nombre de sites, pour contrôler l'ordre partiel des mises à jour de la mémoire partagée. Nous proposons de nouvelles horloges logiques, les horloges vectorielles *barrière-verrous*, pour implanter le protocole LRC. Le principal avantage de ces horloges logiques par rapport aux vecteurs de temps traditionnels est que leur taille n'est pas proportionnelle au nombre de sites, mais au nombre de variables de synchronisation utilisées par l'application. Enfin, nous introduisons une approche hiérarchique utilisant les horloges *barrière-verrous* permettant d'adapter le protocole LRC à des plates-formes multi-réseaux.

Abstract

In this paper we present our proposal for a scalable Lazy Release Consistency (LRC) Distributed Shared Memory (DSM). Firstly, we discuss the scalability problem of current interval-based LRC DSM implementations related to the use of vector clocks for controlling causality of shared memory updates. We propose, then, a new logical clock, *the barrier-lock* clock, for implementing the LRC protocol. The size of the *barrier-lock* clock does not depend on the number of nodes of the system. It is proportional to the number of synchronization variables used by the application. In the second part of this paper, we introduce the hierarchical approach that is given to the LRC protocol for adapting it for a multi-LAN platform. The *barrier-lock* clocks provide support for such an implementation.

---

1 Introduction

Les mémoires partagées réparties (MPR) fournissent l'abstraction d'une mémoire partagée sur des systèmes ayant des mémoires physiquement réparties. C'est un support très attractif qui simplifie le développement d'applications parallèles. Cependant, la plupart des MPRs ont été conçues pour des réseaux locaux de stations de travail ou pour des

* financement : bourse d'études CAPES (Brésil)

multi-processeurs symétriques (SMPs) et ne sont généralement pas adaptées à des réseaux plus étendus.

Le nombre de machines d'un réseau local étant relativement faible, l'idée d'étendre une MPR à un ensemble de réseaux inter-connectés (des multi-réseaux) est séduisante pour fournir une plus grande capacité de traitement et un espace de mémoire physique plus important.

Cependant, ces plates-formes multi-réseaux ont des caractéristiques physiques qui peuvent limiter fortement les performances des applications. La latence entre des machines des réseaux distants peut être beaucoup plus élevée qu'entre des machines d'un même réseau, tandis que le débit est souvent plus faible. Si les réseaux sont hétérogènes, le traitement de l'hétérogénéité peut entraîner un surcoût supplémentaire à chaque transfert de données entre les réseaux. Il faut donc limiter au maximum l'échange de données entre les réseaux. Cela signifie qu'une MPR multi-réseaux n'est pas adaptée à tout type d'applications. Elle vise particulièrement des applications parallèles à gros grain présentant une forte localité dans les références aux données avec des rares opérations de synchronisation ou mises à jour globales. Cependant, des travaux récents [5] ont montré qu'avec des optimisations appropriées, prenant en compte la latence et le débit, des applications de granularité moyenne peuvent s'exécuter de façon performante sur des plates-formes multi-réseaux.

Nous proposons d'implanter une MPR multi-réseaux en utilisant le modèle de cohérence relâchée paresseuse (LRC : Lazy Release Consistency). Le protocole de maintien de la LRC, qui a été initialement défini dans TreadMarks [1], est reconnu comme l'un des plus efficaces pour des réseaux locaux. En relâchant le modèle de cohérence mémoire, LRC réduit le volume des données transférées entre les processeurs. Les opérations de synchronisation des programmes établissant un ordre d'accès à la mémoire sont utilisées pour propager les informations sur les données partagées.

La plupart des implantations du protocole de maintien de la LRC sont peu extensibles. Ceci est en partie dû à l'utilisation des horloges vectorielles, définies par **Mattern** [11] et **Fidge** [7], pour contrôler la causalité des opérations de synchronisation. Le fait que les vecteurs de temps aient une entrée pour chaque noeud du système limite leur extensibilité. De plus, pour contrôler la propagation des modifications, les horloges vectorielles sont stockées dans la mémoire des processeurs et incluses dans de nombreux messages. Donc, si le nombre de processeurs est important, comme ce peut être le cas dans les multi-réseaux, la grande taille du vecteur aura une influence directe sur le volume des données échangées entre les processeurs de la MPR. Nous proposons donc une horloge logique dont la taille est indépendante du nombre de noeuds du système. Nous l'appelons l'horloge *barrière-verrous*. Sa taille est proportionnelle au nombre de variables de synchronisation utilisées par l'application (très souvent ce nombre reste petit). Nous montrons par la suite que le protocole de maintien de la LRC peut être implanté en utilisant les horloges *barrière-verrous* pour contrôler l'ordre partiel des opérations de synchronisation.

Il y a d'autres points qui limitent l'extensibilité du protocole LRC, mais qui ne seront pas abordés dans cet article. Parmi eux, nous pouvons signaler : la synchronisation globale (e.g., barrières, mécanisme de ramasse-miettes), l'ordonnancement des accès aux verrous, la distribution et l'envoi des pages et le fait que chaque processeur garde en mémoire l'historique des mises à jour des autres processeurs.

Nous présentons également dans cet article notre prototype d'une MPR multi-réseaux qui offre la cohérence relâchée paresseuse et utilise les horloges *barrière-verrous*. Nous adoptons une approche hiérarchique où chaque réseau local est vu comme une machine virtuelle unique. Le protocole de maintien de la LRC est appliqué hiérarchiquement. Cette approche permet d'avoir une MPR extensible et de réduire le trafic entre les réseaux.

La section 2 présente le principe du protocole LRC. Dans la section 3, nous définissons les horloges *barrière-verrous* et montrons comment elles peuvent être utilisées pour contrôler la causalité des mises à jour des données partagées. La section 4 décrit le protocole hiérarchique de maintien de la LRC. Enfin, dans la section 5, nous évoquons les autres approches dans le domaine des MPRs.

2 Cohérence relâchée paresseuse

Dans le modèle de *cohérence relâchée* [8], 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. Toutes 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.

La *cohérence relâchée paresseuse* [10] adopte les mêmes principes que la cohérence relâchée, mais la mise à disposition des modifications pour chaque processeur est reportée jusqu'à ce que ce denier exécute une opération d'acquisition.

L'exécution des processus est divisée en *intervalles*. Un nouvel intervalle commence à chaque opération d'acquisition ou libération. Les intervalles sont ordonnés partiellement de la manière suivante :

1. si i_1 et i_2 sont deux intervalles d'un même processus et que i_1 a lieu avant i_2 dans l'ordre du programme alors i_1 précède i_2, i.e., $i_1 \rightarrow i_2$;

2. si i_1 est un intervalle se terminant à la libération d'une variable de synchronisation et que i_2 est l'intervalle commençant à la prochaine opération d'acquisition sur cette variable alors $i_1 \rightarrow i_2$;

3. si $i_1 \rightarrow i_2$ et $i_2 \rightarrow i_3$, alors $i_1 \rightarrow i_3$.

Selon cet ordre partiel, lorsqu'un intervalle i_1 précède un intervalle i_2, toutes les mises à jour effectuées pendant i_1 sont visibles par i_2.

L'ordre partiel des intervalles est contrôlé en utilisant une horloge vectorielle associée à chaque opération de synchronisation [11] [7]. Chaque processeur maintient un vecteur d'entiers de N entrées, où N est le nombre total de processeurs du système. Un processeur j contrôle les intervalles qu'il crée en utilisant la *jème* entrée du vecteur. Les autres entrées mémorisent la vision que le processeur a des mises à jour des autres sites. Le processeur j met à jour son vecteur v_j à chaque opération de synchronisation de la manière suivante :

1. Lors d'une opération d'acquisition, si j est différent du processeur qui a exécuté la dernière libération, appelé le processeur l, j met à jour son vecteur local avec le

maximum de son propre vecteur et celui du vecteur de l:
$1 \leq k \leq N$; $v_j[k] = max(v_j[k], v_l[k])$;

2. Le nouvel intervalle est enregistré :
$v_j[j] = v_j[j] + 1$.

Ces deux règles garantissent l'ordre partiel des opérations d'acquisition et de libération. En effet, **Mattern** [11] définit la propriété de cohérence forte qui peut être appliquée aux opérations de synchronisation. Soit $v(s_1)$ la valeur de l'horloge vectorielle associée à l'opération de synchronisation s_1 et $v(s_2)$ celle associée à l'opération s_2, la cohérence forte définit que :

- $v(s_1) < v(s_2) \Leftrightarrow s_1 \rightarrow s_2$;

- $\neg(v(s_1) < v(s_2))$ et $\neg(v(s_2) < v(s_1)) \Leftrightarrow s_1 \| s_2$.

Lors d'une opération d'acquisition, le processeur j doit recevoir du processeur l, celui qui a effectué la libération, tous les intervalles dont les vecteurs d'horloges sont supérieurs à son horloge locale. Pour cela, j envoie la valeur courante de son horloge vectorielle au processeur l. Ce dernier, renvoie alors la liste des intervalles et des pages modifiées qu'il connaît, mais ignorée de j. 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. Ainsi, 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 processus 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.

3 Protocole extensible de maintien de la LRC

Les horloges vectorielles capturent précisément la causalité entre les opérations d'acquisition et de libération. Cependant, leur taille est proportionnelle au nombre de processeurs du système ce qui peut limiter leur extensibilité. Dans une MPR adaptée au multi-réseaux, le nombre de processeurs peut être très important exigeant donc des vecteurs très grands.

Nous proposons une nouvelle horloge logique, l'horloge *barrière-verrous*, dont la taille est indépendante du nombre de noeuds. Pour contrôler complètement l'ordre partiel des intervalles, l'horloge vectorielle traditionnelle (avec une entrée par noeud) est remplacée par un simple compteur de barrière et un vecteur ayant une entrée par verrou, i.e., le doublet (b, vl). Comme dans la version originale du protocole de maintien de la LRC, des *diffs* et des notifications d'écriture sont associés à chaque intervalle.

Le nombre de verrous utilisés par les applications étant souvent très petit (un ou deux ou même aucun pour des applications n'utilisant que des barrières), les horloges *barrière-verrous* peuvent réduire considérablement la taille de l'estampillage par rapport aux horloges vectorielles traditionnelles. Ceci est d'autant plus vrai si le nombre de sites est important. Par exemple, si une application s'exécute sur une plate-forme avec un grand nombre de sites, mais réclame au maximum 2 verrous, un vecteur avec 2 entrées et une simple variable scalaire sont suffisants pour implanter le protocole de maintien de la LRC.

3.1 Gestion des verrous

Pour contrôler les mouvements des verrous entre les processeurs, nous utilisons le concept du diagramme événementiel ("poset-diagram") défini par **Mattern** [11]. Ce diagramme reflète la relation logique entre les événements en termes de causalité. Graphiquement, si $a \rightarrow b$, il existe un *chemin de causalité* de a vers b dans le diagramme temporel qui est plus facilement visible dans le diagramme événementiel. Les deux diagrammes sont isomorphes [11].

Nous utilisons l'idée du diagramme événementiel pour contrôler l'ordre partiel des opérations sur des verrous. Ce diagramme montre qu'il est possible de tracer des *chemins de causalité* constitués des opérations d'acquisition et de libération. Avec un seul verrou, l'acquisition et la libération tracent un simple flux des mises à jour des données partagées. Ceci représente, en fait, la façon dont les processeurs sérialisent leurs accès aux variables partagées.

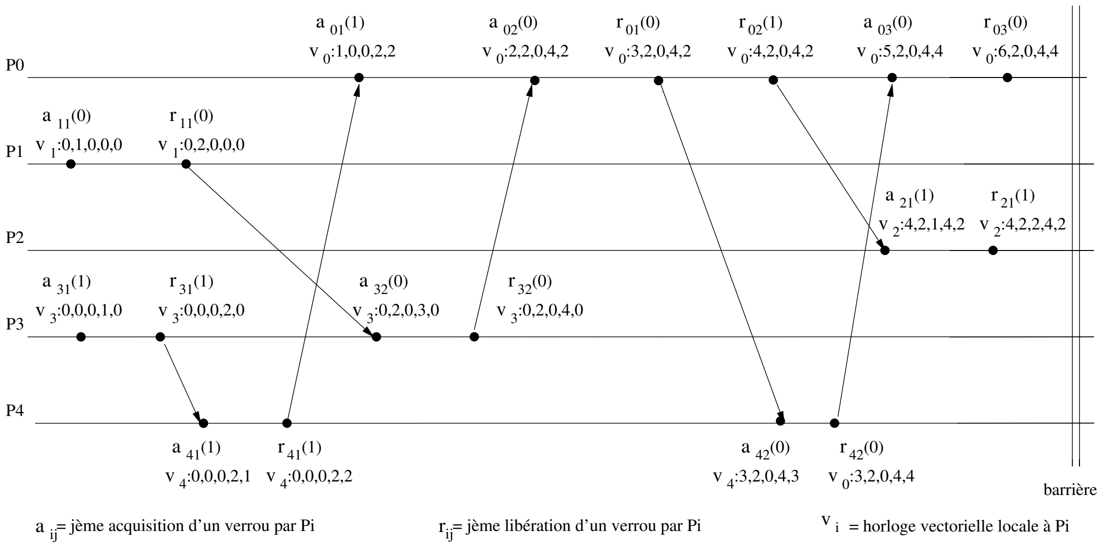

FIG. 1 – *Diagramme temporel des acquisitions/libérations suivant le modèle LRC classique.*

La figure 1 montre le diagramme temporel lié à des opérations d'acquisition et de libération sur deux verrous faites par cinq processeurs. Pour simplifier, les opérations d'écriture ou de lecture ainsi que les mises à jour des variables partagées ne sont pas représentées. Les notations a_{ij} et r_{ij} désignent respectivement la *jème* acquisition ou libération d'un verrou faite par le processeur P_i, tandis que v_i correspond à la valeur de l'horloge vectorielle de P_i lors de l'opération sur un verrou. La figure 2 montre le diagramme événementiel correspondant.

Les verrous étant utilisés pour contrôler les sections critiques lors des accès aux variables partagées, le protocole de cohérence relâchée assure qu'à un instant donné un verrou n'est

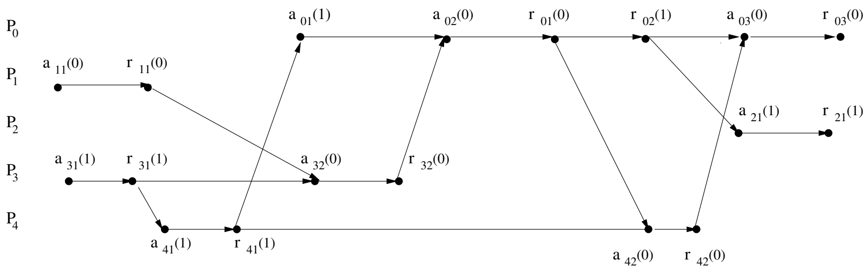

FIG. 2 – *Diagramme événementiel.*

possédé que par un seul processeur. Le comportement est similaire à un jeton. Ceci signifie que si deux opérations (acquisition ou libération) sur un verrou sont concurrentes, alors il s'agit de verrous différents. Ainsi, pour estampiller les opérations sur des verrous d'un diagramme événementiel, nous pouvons utiliser un vecteur où chaque entrée est associée à un verrou et non pas à un processeur. Grâce à cette propriété d'exclusivité dans l'utilisation des verrous, il est possible de garantir que deux processeurs différents n'incrémentent pas parallèlement la même entrée sur leur vecteur local.

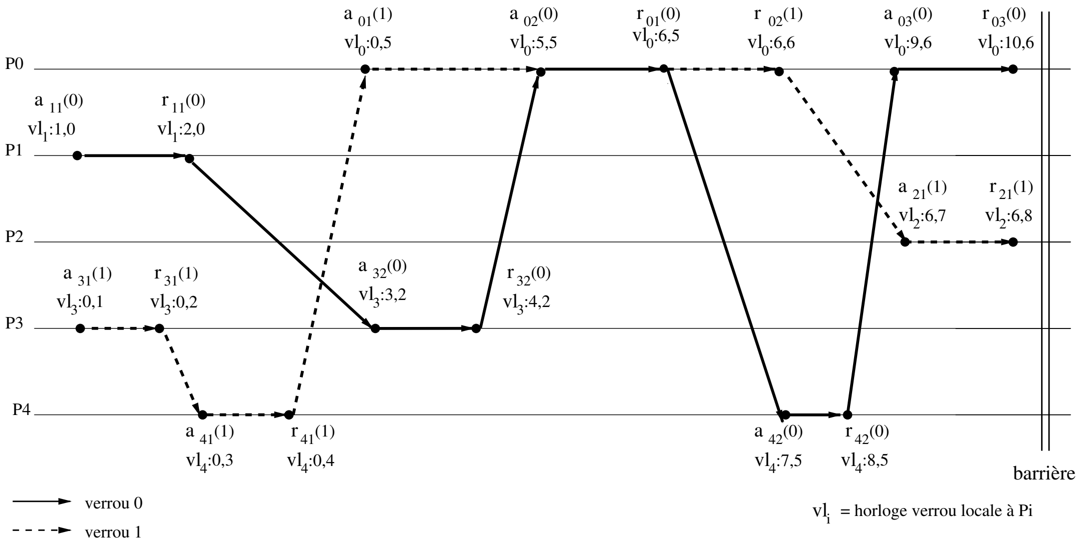

FIG. 3 – *Diagramme événementiel en utilisant les horloges verrous.*

Ainsi, lors de l'opération d'acquisition du verrou i, le processeur acquéreur j met à jour son vecteur de verrous vl_j de la manière suivante :

1. Si j est différent du processeur l, celui qui a libéré i, j met à jour les entrées de son vecteur local avec le maximum de son vecteur et celui du vecteur vl_l, transmis par l: $1 \leq k \leq N_v$; $vl_j[k] = max(vl_j[k], vl_l[k])$; N_v étant le nombre de verrous;

2. Le nouvel intervalle est enregistré :

$vl_j[i] = vl_j[i] + 1$.

La figure 3 reprend le même exemple en utilisant ces nouvelles horloges logiques. Dans la suite, nous appellerons ces horloges, les horloges *verrous*.

3.2 Gestion des barrières

Une barrière est un point de synchronisation où tous les processeurs positionnent leur vecteur à la même valeur et collectent les informations de cohérence (les notifications d'écriture) sur tous les accès en écriture effectués. Cette nouvelle valeur du vecteur de temps recouvre donc toutes les valeurs précédentes. Selon ce principe, les horloges vectorielles locales des processeurs pourraient être remises à zéro pour la prochaine séquence d'opérations sur des verrous. Une telle approche n'est pas possible à cause de la stratégie d'ajournement d'envoi des *diffs* du protocole de maintien de la LRC. Toutes les informations sur l'ordre partiel doivent être maintenues car les *diffs* d'une page invalidée doivent être appliqués dans le même ordre. Cela n'est pas réalisé lors des opérations de synchronisation mais lors des futurs défauts de page. Dans le protocole LRC original, cette "remise à zéro" des horloges vectorielles est faite lors des opérations de ramasse-miettes qui récupèrent la mémoire utilisée par les structures de notifications d'écriture, d'intervalles et de *diffs* [10].

Puisque lors des barrières tous les processeurs égalisent leur vecteur de temps et qu'entre deux barrières les verrous sont responsables de l'ordre partiel, l'exécution d'une application peut être divisée en *intervalles de barrière*, eux mêmes subdivisés en *intervalles de verrou*. L'ordre des *intervalles de barrière* peut être simplement contrôlé en lui associant une variable scalaire. Ainsi, à chaque appel à la barrière, les vecteurs *verrous* sont remis à zéro et la variable barrière est incrémentée. Le concept d'*intervalle de barrière* est valide à condition qu'une barrière soit appelée par tous les processus. Cette contrainte emporte une certaine perte de généralité aux horloges *barrière-verrous*.

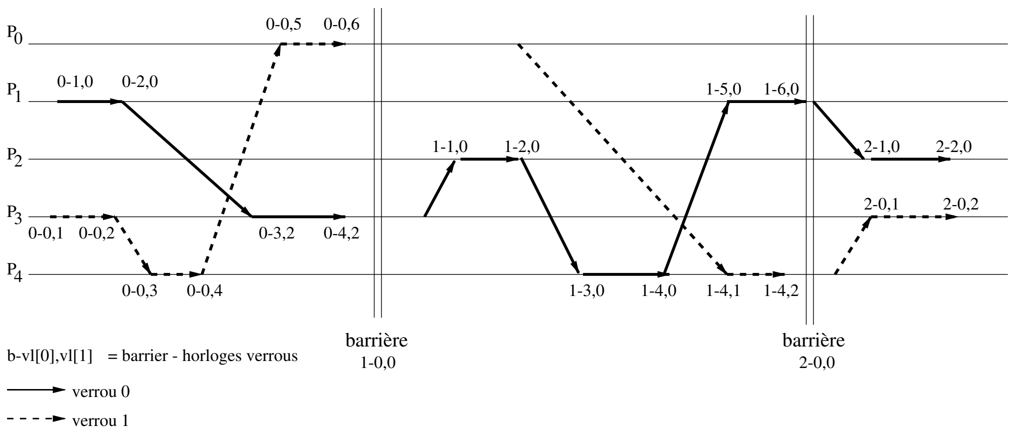

FIG. 4 – *LRC basé sur les horloges vectorielles barrière-verrous.*

Les estampillages des horloges *barrière-verrous* sont représentés, alors, par la paire (b, vl) où b contrôle les appels à la barrière et vl les opérations sur les verrous. Pour comparer deux estampillages appartenant à des *intervalles de barrière* différents, il suffit de comparer la valeur de leur variable b.

La figure 4 illustre un exemple où une barrière est exécutée à deux reprises définissant trois *intervalles de barrières*. La valeur de l'estampillage (b, vl) est signalée à chaque opération sur un verrou ou la barrière.

De manière similaire aux horloges vectorielles classiques, les horloges *barrière-verrous* permettent de capturer complètement la causalité des opérations de synchronisation du protocole de la maintien de la LRC. Il y a un isomorphisme entre les deux. La preuve de cet isomorphisme est donnée dans [3].

3.3 Mesures de performance

Pour valider notre approche, nous avons remplacé, dans TreadMarks (version 0.10.1), les horloges traditionnelles par les horloges *barrière-verrous*. Des tests de non régression ont été faits en utilisant cinq applications distribuées avec TreadMarks : SOR et TSP, applications fournies par TreadMarks [8]; IS et 3D FTT du benchmark NAS [4]; Barnes-Hut du benchmark SPLASH [15].

Le nombre de verrous acquis et d'appels faits à la barrière par chaque application est indiqué dans le tableau 1.

applications	nombre de verrous	nombre d'appels à la barrière
SOR	0	401
TSP	2	3
IS	0	82
3D FFT	0	104
Barnes-Hut	0	13

Tab.1 - *Comportement des applications.*

Nous avons comparé la taille des messages de synchronisation de notre MPR, la MPR *barrière-verrous*, avec celle de la MPR TreadMarks original. Les tests ont été effectués sur 7 stations de travail Sun Sparc-5 reliées par un réseau Ethernet à 100 Mbit/s.

Dans la figure 5, nous indiquons le rapport: *nombre d'octets des messages de synchronisation échangés en utilisant la MPR barrière-verrous/ nombre d'octets des messages de synchronisation en utilisant TreadMarks*. Pour illustrer l'extensibilité de la taille des horloges, nous avons fait varier la constante qui définit le nombre de processeurs du système dans TreadMarks. Cela permet de simuler des opérations de gestion du protocole de maintien de la LRC avec un nombre variable de machines.

Nous pouvons observer une diminution importante du volume des données échangées lors des opérations de synchronisation. Cette réduction peut apporter un gain de performance surtout pour des réseaux à faible débit (e.g., connexions entre deux réseaux). Naturellement, notre modèle est moins adapté aux applications qui utilisent un grand nombre de verrous, surtout si les données sont partagées avec un fort taux de migration. Par exemple, pour l'application Water du benchmark SPLASH [15], nous avons observé une nette augmentation de la taille des messages de synchronisation. Dans ce cas, le chevauchement des estampillages non concurrents ou l'utilisation de la technique différentielle de **Singhal-Kshemkalyani** [16] pourraient être appliqués afin de réduire la taille des messages de synchronisation.

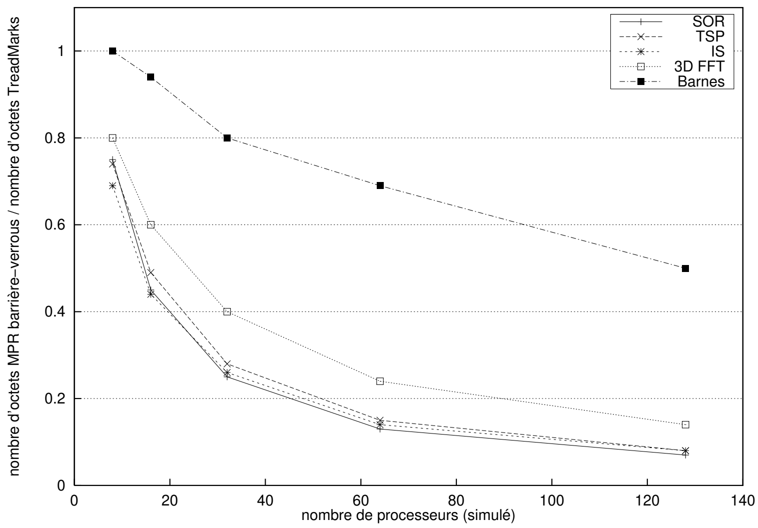

FIG. 5 – *Volume de données échangées aux opérations de synchronisation.*

4 Une approche hiérarchique pour des multi-réseaux

Nous proposons une MPR multi-réseaux [2] basée sur le principe suivant: chaque réseau (i.e., chaque MPR locale) voit un réseau distant comme une machine virtuelle unique et non pas comme un ensemble de machines. Au lieu d'être une MPR globale, la MPR multi-réseaux est composée de plusieurs MPRs locales inter-connectées. L'utilisation des horloges *barrière-verrous* est appropriée car elle permet de comparer les horloges locales des processeurs exécutant des MPRs distinctes, ce qui serait plus coûteux avec les horloges traditionnelles.

L'idée est d'appliquer le protocole de maintien de la LRC de manière hiérarchique, i.e., au niveau des processeurs et des réseaux. Toutes les mises à jour disponibles dans un réseau local sont utilisées au maximum pour éviter les communications entre réseaux. Quand un processeur envoie une requête à un processeur qui se trouve sur un réseau distant, le processeur demandeur ne reçoit de la MPR distante que les mises à jour que les autres processeurs de la même MPR ne possèdent pas. Le processeur demandeur obtient les mises à jour locales qui lui manquent en interrogeant directement les processeurs locaux.

Le protocole LRC hiérarchique multi-réseaux sera implanté en utilisant le principe des *chemins de causalité* des intervalles, décrit dans la section 3.1. Dans 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 qui lui manquent (i.e., toutes les *notifications d'écriture* liées à cette page) ainsi que les processeurs qui les ont effectuées. Toutefois, la requête pour les mises à jour est toujours adressée au dernier processeur qui a modifié la page, i.e., le responsable pour la dernière mise à jour du *chemin de causalité*. Dans le protocole LRC hiérarchique, si ce processeur appartient à un réseau distant, le processeur demandeur divise chaque *chemin de causalité* en deux parties. Il identifie la dernière mise à jour réalisée par un processeur local à son réseau. Le processeur demandeur récupère donc de ce dernier les mises à jour que celui-ci possède et n'obtient du processeur dis-

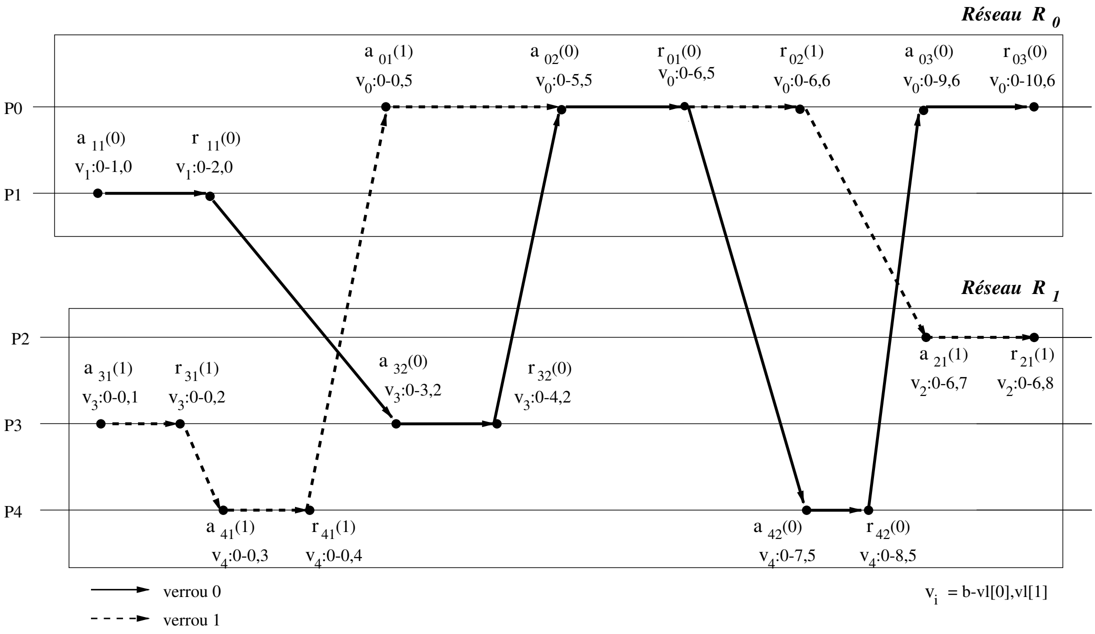

FIG. 6 – *LRC hiérarchique.*

tant 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 réseaux peut être fortement réduit. De plus, cette approche parallélise le traitement des deux 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 (verrou et barrière).

Le principe du protocole LRC hiérarchique est illustré dans la figure 6. Les synchronisations sont identiques à celles de la figure 4, mais les processeurs P_0 et P_1 appartiennent au même réseau R_0, tandis que P_2, P_3 et P_4 font partie du réseau R_1. Supposons qu'après l'opération d'acquisition $a_{42}(0)$, le processeur P_4 fait un accès à une page qui n'est pas à jour. P_4 genére, alors, un défaut de page. Toutefois, il a déjà la connaissance des mises à jour qui lui manquent (envoyées par P_0 lors de l'opération d'acquisition $a_{42}(0)$). Dans le cas du protocole LRC original, si, par exemple, P_0 avait été le dernier processeur à avoir modifié la page, P_4 lui demanderait toutes les mises à jour. Dans le cas du protocole LRC hiérarchique, P_4 émet deux requêtes en parallèle: l'une vers P_3, le dernier processeur de R_1 ayant possédé le verrou 0 et l'autre vers P_0. Ainsi, P_4 demande à P_3 les mises à jour entre la valeur de son horloge, v_4= 0-0,4 et la valeur v_3=0-4,2 et demande à P_0 le restant des mises à jour.

5 Travaux apparentés

TreadMarks [10] est la première MPR à avoir implanté le protocole de maintien de la cohérence relâché paresseuse. *Automatic Update Release Consistency* (AURC)] [9] et *Home-based Lazy Release Consistency*(HLRC) [19] de l'Université de Princeton sont des variantes du protocole LRC. Les trois implantations utilisent les horloges vectorielles traditionnelles pour assurer la causalité des opérations de synchronisation. Le protocole AURC utilise un mécanisme matériel pour propager automatiquement les mises à jour d'une page. Un site propriétaire est associé à chaque page. Chaque écriture sur une copie de page est automatiquement retransmise à son site propriétaire. Lors d'un défaut de page, le processeur fautif charge la page entièrement à partir de ce noeud propriétaire. HLRC utilise également l'idée de collecter toutes les mises à jour sur un site particulier. Cependant, au lieu d'utiliser un support matériel spécifique, les *diffs* sont transmis et appliqués sur le noeud propriétaire à la fin de chaque intervalle.

En [12], l'auteur étend le modèle de cohérence causale pour faire de la cohérence relâchée. L'odre partiel qu'il propose tient compte des opérations de synchronisation (verrous et barrières) et des créations de processus fils ("fork"). Sa mise en oeuvre utilise des horloges vectorielles dont les coordonnées sont associées aux pages.

D'autres travaux proposent d'étendre des MPRs conçues pour des réseaux de stations de travail à des réseaux de machines SMP (symmetrical multiprocessor). Ainsi Hu et al. présentent dans [6] une version de TreadMarks pour des réseaux SMPs. Leur solution exploite les mécanismes matériels de partage de mémoire qui existent dans les architectures SMP. Cashemere-2L [17] offre un protocole à écrivains multiples tandis que SMP-Shasta [14] supporte une mémoire partagée avec une granularité plus fine. HLRC-SMP [13] propose une extension du protocole HLRC aux réseaux de machines SMP.

En terme d'extensibilité, il existe des horloges logiques utilisant un nombre constant d'entrées et indépendantes du nombre de site. Par exemple, les *horloges plausibles* [18] permettent de déduire des informations sur l'ordre causal des événements. Cependant, elles peuvent ordonner quelques événements qui sont en fait concurrents (l'isomorphisme entre les horloges plausibles et la causalité n'est pas toujours respecté).

6 Conclusions

Nous avons présenté les horloges *barrière-verrous* pour contrôler l'ordre partiel des opérations de synchronisation dans le protocole LRC. Ces horloges sont extensibles car indépendantes du nombre de sites. Si l'application n'utilise pas un nombre important de verrous, une diminution conséquente de la taille des messages de synchronisation par rapport à l'implantation de LRC basée sur les horloges vectorielles traditionnelles peut être observée.

Nous sommes en train de développer un deuxième prototype pour des plates-formes multi-réseaux, basé sur le protocole LRC hiérarchique présenté dans la section 4. Ici, l'utilisation des horloges *barrière-verrous* est essentielle car elle permet de comparer les horloges locales des processeurs exécutant des MPRs différentes.

Remerciements: Nous remercions Eric Gressier et Bruno Bretelle pour leurs relectures et nombreuses suggestions.

Bibliographie

1. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

2. L. Arantes, B. Folliot, and P. Sens. An approach for a multi-LAN DSM based on Lazy Release Consistency. In *The 3rd European Research Seminar on Advances in Distributed Systems*, Funchal (Portugal), April 1999.
3. L. Arantes, B. Folliot, and P. Sens. *A node count-independent logical clock for scaling Lazy Release Consistency*, 1999. Submitted to publication.
4. D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmark. Technical Report 103863, NASA, July 1993.
5. H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *The 12th International Parallel Processing Symposium*, April 1998.
6. A. Cox, Y. Hu, H. Lu, and W. Zwaenepoel. OpenMP on networks of SMP. In *The 13th International Parallel Processing Symposium*, April 1999.
7. C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, July 1991.
8. K Gharachorloo, D. Lenoski, J. Laudon, Gupta A., and J. Henessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *The 17th International Symposium on Computer Architecture*, pages 15–26, 1990.
9. L. Iftode, C. Dubnicki, E. Felten, and K Li. Improving release consistency shared virtual memory using automatic update. In *The 2nd IEEE Symposium on High Performance Computer Architecture*, February 1996.
10. P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Texas University, Texas (EUA), 1995.
11. F. Mattern. Virtual time and global states in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, Elsevier (Holland), October 1988.
12. J. Ranjit. *Implementing and Programming Weakly Consistency Memories.* PhD thesis, Georgia Institute of Technology, March 1995.
13. R. Samanta, A. Bilas, L. Iftode, and J. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *The 4th Symposium on High Performance Computer Architecture*, February 1998.
14. D. Scales, Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. Technical Report Research Report 97/3, Western Research Laboratory, March 1997.
15. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, March 1992.
16. M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 33:47–53, August 1992.
17. 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 *The 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.
18. F. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *The 10th International Workshop on Distributed Algorithms*, Bologna (Italy), October 1996.
19. Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *The 2nd Symposium on Operating Systems Design and Implementation*, October 1996.