Extension de la plate-forme DSM-PM2 pour le support de protocoles de cohérence relâchée multithreads

Gabriel Antoniu, Vincent Bernardi, Luc Bougé

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

Extension de la plate-forme DSM-PM2 pour le support de protocoles de cohérence relâchée multithreads

Gabriel Antoniu, Vincent Bernardi, Luc Bougé

LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
Contact : Gabriel.Antoniu@ens-lyon.fr.

Dans leur présentation traditionnelle, les bibliothèques de gestion de *mémoire distribuée virtuellement partagée* (MVP, en anglais DSM) [8, 11, 12, 4] permettent à des processus de partager un espace d'adressage commun selon un *modèle de cohérence* fixé. L'objectif du projet DSM-PM2 est de fournir au programmeur d'*application distribuée multithread* une *plate-forme d'implémentation* où il puisse développer et optimiser *conjointement* son application et le protocole de cohérence MVP qui la supporte, de manière portable. DSM-PM2 est actuellement disponible sur des grappes de PC sous Linux, avec les réseaux Ethernet, Myrinet et SCI, et les interface de communication TCP, MPI, BIP, SISCI, VIA, etc.

DSM-PM2 fournit les briques de base pour la construction d'une large classe de protocoles de cohérence utilisables dans un environnement d'exécution multithread : il généralise donc les fonctionnalités de MVP comme DSM-Threads [9] et Millipede [5]. À partir de ces briques, 6 protocoles de cohérence sont déjà construits dans la version actuelle. L'utilisateur peut facilement les modifier ou en ajouter d'autres. Dans cet article, nous décrivons la mise en place sous DSM-PM2 des deux protocoles de cohérence relâchée *multithreads* et un aperçu de leurs performances.

1. La plate-forme DSM-PM2

DSM-PM2 est une plate-forme d'implémentation pour les protocoles de cohérence MVP en environnement multithread. Elle est construite au-dessus de l'environnement d'exécution multithread distribué PM2 (*Parallel Multithreaded Machine*, [10]). PM2 fournit une interface de type POSIX pour créer, manipuler et synchroniser des threads légers en espace utilisateur entre des nœuds de calcul distribués. PM2 est disponible sur la plupart des systèmes de type Unix, en particulier Solaris et Linux (un portage sous NT est en cours). Pour assurer sa portabilité, PM2 s'appuie sur une bibliothèque de communication appelée Madeleine [3], qui a été portée sur une grande variété d'interfaces de communication : des interfaces conçues pour les hautes performances comme BIP/Myrinet, SISCI/SCI et VIA, mais aussi des interfaces plus classiques comme TCP et MPI. DSM-PM2 hérite directement de cette portabilité, puisque toutes les primitives de communication liées à la MVP ont été implémentées au-dessus de Madeleine. Une

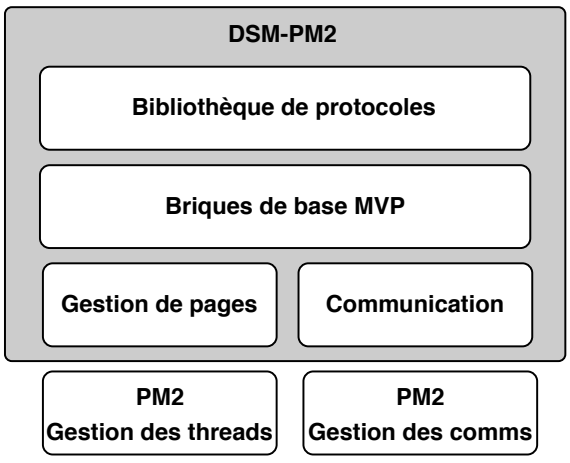

FIG. 1 : L'architecture logicielle de DSM-PM2.

Fonction du protocole	Description
read_fault_handler	Action sur défaut en lecture
write_fault_handler	Action sur défaut en écriture
read_server	Action sur réception d'une requête en lecture
write_server	Action sur réception d'une requête en écriture
invalidate_server	Action sur réception d'une requête en invalidation
receive_page_server	Action sur réception d'une page demandée
lock_acquire	Action sur acquisition d'un verrou
lock_release	Action sur relâchement d'un verrou

TAB. 1 : Les huit actions définissant un protocole DSM-PM2.

particularité intéressante de PM2 est qu'il permet la *migration de threads* de manière préemptive et transparente d'un nœud à l'autre. DSM-PM2 fournit en fait à ces threads mobiles l'illusion d'une mémoire uniformément partagée au-dessus de l'architecture distribuée de PM2.

DSM-PM2 est structuré en couches (figure 1). Au niveau supérieur, la *bibliothèque de protocoles* permet de construire des protocoles de cohérence à partir des briques de base fournies par la couche inférieure. On trouve par exemple parmi ces briques les routines nécessaires pour modifier les droits d'accès aux pages en mémoire virtuelle, pour rattraper les interruptions produites par l'accès à une page non autorisée, pour manipuler les structures de données associées aux pages partagées, pour envoyer et recevoir des messages entre les nœuds, pour gérer les accès concurrents entre threads sur un même nœud, etc.

Un nombre arbitraire de protocoles peuvent être définis dans la bibliothèque. L'application choisit l'un d'entre eux par un simple appel de fonction à l'initialisation. Par exemple :

```
pm2_dsm_set_default_protocol(li_hudak);
```

L'utilisateur peut facilement spécifier d'autres protocoles en définissant un certain nombre de fonctions caractéristiques à l'aide des briques de base fournies par les couches inférieures. Actuellement, ces fonctions sont au nombre de huit (table 1).

```
int new_proto;
new_prot = dsm_create_protocol
 (read_fault_handler, write_fault_handler,
  read_server, write_server,
  invalidate_server, receive_page_server,
  acquire_handler, release_handler);
pm2_dsm_set_default_protocol(proto);
```

En fait, les protocoles prédéfinis sont exactement spécifiés de cette manière. Il n'y a donc aucune différence entre les protocoles prédéfinis et les protocoles éventuellement définis par l'utilisateur. Dans la version actuelle, six protocoles prédéfinis sont disponibles dans la couche supérieure. Nous donnons ci-dessous une description rapide de ces protocoles (voir [2] pour une présentation plus détaillée).

Cohérence séquentielle. DSM-PM2 propose deux protocoles de ce type. 1) Le protocole li_hudak, d'après les travaux de Li et Hudak [8] adaptés par Mueller [9] au cas multithread : il utilise la réplication de pages sur un défaut en lecture, et la migration de page sur un défaut en écriture. 2) Un protocole migrate_thread original basé sur la migration de threads PM2 : sur un défaut d'accès, le thread fautif migre *spontanément* vers le nœud qui détient les données accédées, et il répète l'accès de manière transparente sur ce nouveau nœud.

Cohérence Java. DSM-PM2 est utilisé comme cible pour le compilateur Java Hyperion [1]. Il fournit deux protocoles qui implémentent le modèle de cohérence Java décrit dans la spécification du langage [6]. Ces deux protocoles diffèrent par la manière dont les défauts

d'accès sont détectés. Dans le premier, la présence d'une donnée est testée *explicitement* à chaque accès, grâce aux informations disponibles dans le *bytecode* Java. Dans le second, aucune vérification n'est faite lors de l'accès : le défaut éventuel d'accès est détecté par une interruption matérielle et traité par les routines génériques de bas niveau fournies par DSM-PM2.

Cohérence relâchée. Les sections suivantes sont entièrement consacrées à la conception des deux protocoles de cohérence relâchée disponibles, à leur implémentation en environnement multithread et à leur évaluation.

2. Mise en œuvre de protocoles de cohérence relâchée

Un protocole MRSW par invalidation

Le protocole erc_sw (*Eager Release Consistency, Single Writer* est un protocole MRSW (*Multiple Readers, Single Writer*) de cohérence relâchée immédiate est une adaptation du schéma de Li-Hudak. Aucune action particulière n'est effectuée au moment de l'entrée en section critique. La cohérence est assurée par les actions effectuées lors de la sortie. Le principe de ce protocole est simple : la gestion de chaque page partagée est confiée à un nœud *propriétaire* qui seul peut y écrire. Les autres nœud ne peuvent que lui demander une copie de la page en lecture. Lorsque le nœud propriétaire d'une page y accède en écriture, il positionne un drapeau. Lors de la sortie de la section critique, le nœud propriétaire invalide toutes les copies en lecture existantes de la page si le drapeau est validé. Ainsi, un autre nœud souhaitant plus tard lire une variable située sur la page devra demander une nouvelle copie en lecture : il lira donc la nouvelle valeur de la variable en question. Si un nœud veut écrire sur la page, il doit demander à son propriétaire de lui céder son droit : il en devient alors le nouveau propriétaire.

Le schéma général de ce protocole est classique. Notre contribution est d'en avoir fournie une version adaptée à un environnement d'exécution multithread. En effet, dans les descriptions classiques, il est implicitement supposé que les demandes d'accès se succèdent séquentiellement : aucune demande n'est acceptée par le nœud propriétaire tant que la demande courante n'est pas traitée. Cette hypothèse n'est plus valide dans contexte multithread : deux threads peuvent demander *en même temps* l'accès en écriture à une page au propriétaire (*ré-entrance*). Il faut donc que l'implémentation garantisse que l'accès concurrent au structures de données internes respecte les contraintes de cohérence, tout en permettant le degré de concurrence maximal. La prise en compte de ces aspects est très délicat dans la conception des protocoles, car la combinatoire des effets concurrents est difficile à maîtriser. DSM-PM2 fournit des briques de base dont la ré-entrance est garantie. Cette caractéristique est d'une grande aide pour le concepteur de protocoles.

Un protocole à écrivains multiples

Le protocole hbrc_mw (*Home-Based Release Consistency, Multiple Writers*) est un protocole MRMW (*Multiple Readers, Multiple Writers*) basé sur une technique classique de *twinning* [7], une méthode de calcul de différences pour déterminer les parties d'une page qui doivent être mises à jour. Chaque page est *statiquement* attachée à un nœud *propriétaire* tout au long de l'exécution. Comme dans le protocole précédent, aucune action particulière n'est effectuée lors de l'entrée d'un thread en section critique. Lorsqu'un thread demande à accéder en lecture à une page partagée, une copie en lecture en est ramenée depuis le nœud propriétaire de la page. Lorsqu'un thread demande un accès en écriture à une page partagée, une copie en lecture est

ramenée si nécessaire, puis une sauvegarde *jumelle* (*twin*) en est faite. Le thread peut alors modifier la page. Lorsqu'il quitte la section critique, chaque page qui a été modifiée est comparée mot à mot avec sa jumelle. L'ensemble des différences détectées est alors envoyées au nœud propriétaire de la page qui les applique immédiatement sur sa copie de la page, et invalide toutes les autres copies en lecture éventuellement existantes.

Ce protocole autorise la plusieurs nœuds à écrire sur la même page en même temps, évitant ainsi les effets de *va-et-vient* induits par le protocole précédent dans ce cas (*false sharing*). La technique de *twinning* évite de transférer systématiquement des pages entières sur le réseau, mais seulement les différences : ceci représente une économie importante de bande passante réseau. Par contre, elle consomme plus de ressources processeur et mémoire que le précédent : création de la jumelle et comparaison mot à mot de l'original avec la copie. Le compromis entre les deux aspects dépend donc crucialement des schémas d'accès de l'application et des performances relatives des différents mécanismes en jeu.

Comme dans le cas précédent, notre contribution a consisté ici à concevoir une version du protocole adaptée à un environnement d'exécution multithread. L'utilisation des briques de base DSM-PM2 dont la ré-entrance est garantie simplifie considérablement le travail de conception et diminue significativement le temps de mise au point. Il faut cependant noter que ce cadre de travail impose certaines contraintes minimales. Par exemple, DSM-PM2 ne fournit pas actuellement de routine pour calculer les différences entre pages avec une granularité plus fine que le mot. Si la mémoire partagée est utilisée par le programme avec une granularité plus fine, les changements appliqués à un objet pourraient être pris en compte, puis effacés par des changements appliqués à un objet voisin.

Mise en œuvre dans DSM-PM2

L'implantation de ces deux protocoles s'est faite dans le langage C. L'ensemble représente environ 1100 lignes de code en langage C et 6 semaines de travail (incluant la découverte de DSM-PM2).

La figure 2 montre un exemple (légèrement simplifié pour la lisibilité) de la fonction release() du protocole à écrivains multiples (hbrc_mw). Cette fonction est appelée par DSM-PM2 lorsqu'un thread sort de section critique. Cet exemple donne une idée assez précise du niveau d'abstraction permis par DSM-PM2 dans la conception d'un protocole. Il faut noter que tous les problèmes de concurrence sont traités au niveau des routines de DSM-PM2 : le programmeur est (au moins dans ce cas simple) totalement déchargé du souci de garantir la ré-entrance.

```
void dsmlib_hbrc_release ()
{
  page = remove_first_from_list (&list);
  while (page != NULL) {
    if (get_prob_owner (page) == dsm_self ())
      invalidate_copyset (page);
    else {
      send_diffs (page, get_prob_owner (page));
      free_twin (page);
      send_invalidate_req
                (get_prob_owner (page), page);
      wait_ack (page, 1);
      set_access (page, NO_ACCESS);
    }
    page = remove_first_from_list (&list);
  }
}
```

FIG. 2 : La fonction lock_release() du protocole hbrc_mw.

Dans cette fontion, la boucle principale applique le même traitement à toutes les pages enregistrées dans la liste, c'est-à-dire toutes celles qui ont été modifiées par le thread depuis son entrée en section critique. Pour chacune des pages, si le nœud local est le propriétaire de la

page, on se contente de demander l'invalidation de toutes les copies en lecture existantes ; dans le cas contraire, on calcule les différences entre la page modifiée et la page jumelle sauvegardée à l'entrée en section critique. Ces différences sont envoyées au propriétaire de la page, puis il est demandé au propriétaire d'invalider toutes les copies en lecture existantes. Une fois que le propriétaire a confirmé l'invalidation, on passe au traitement de la page suivante.

La mise en œuvre de ces deux protocoles dans DSM-PM2 a exigé d'étendre le module de gestion de pages (figure 1) pour y inclure la gestion des *twins*, et de restructurer les routines internes pour associer l'activation des fonctions du protocole (table 1) aux opérations sur les objets de synchronisation, en l'occurrence les verrous distribués contrôlant les section critiques. Il faut noter que ces améliorations sont réalisées au niveau du support générique de DSM-PM2, et non au niveau des deux protocoles présentés. Ils restent donc disponibles pour toute amélioration ultérieure de ces protocoles, ou pour l'implémentation d'autres protocoles de la même famille. La pérennité de l'investissement logiciel complexe ainsi réalisé est sûrement un atout majeur en faveur de l'approche présentée ici.

3. Résultats expérimentaux

Afin de comparer l'ensemble des protocoles, nous avons avons implémenté une résolution branch-and-bound du problème du voyageur de commerce (*Traveling Salesperson Problem*, TSP). Les tests ont été exécutés sans aucune modification du programme avec 4 protocoles de cohérence différents, sur 2 plates-formes différentes : 1) une grappe de PC Pentium Pro 200 MHz sous Linux 2.2.13 interconnectés par un réseau Myrinet géré par l'interface BIP ; 2) une grappe de PC Pentium II 450 MHz interconnectés par un réseau SCI géré par l'interface SISCI. Le problème est résolu pour 17 villes placées aléatoirement et nous utilisons un thread application par nœud.

```
DSM_BEGIN_DATA
int a;
int b;
END_DSM_DATA

// Thread 1 sur un noeud

dsm_lock(L1);
for (i = 1; i < N; i++)
  a = a + 1;
dsm_unlock(L1)

// Thread 2 sur un autre noeud

dsm_lock(L2);
for (i = 1; i < N; i++)
  b = b + 1;
dsm_unlock(L2)
```

FIG. 3 : Un cas pathologique de faux partage

Le comportement des différents protocoles est illustré sur les figures 4 et 5. Étant donné que la seule variable partagée accédée est la longueur du chemin critique et que les accès à cette variable sont toujours protégés par des sections critiques, les bénéfices de la cohérence relâchée ne sont pas illustrés ici. Les 3 protocoles basés sur la migration de page sont pratiquement équivalents en terme de performance à cause de la rapidité des communications inter-nœuds par rapport aux accès mémoire intra-nœuds. Par contre, le protocole basé sur la migration de thread est significativement moins efficace. Ceci est dû essentiellement au fait que tous les threads qui accèdent l'unique variable partagée migrent sur le nœud qui détient la variable partagée, qui devient ainsi surchargé.

On peut cependant mettre en évidence l'intérêt des protocoles hbrc_mw et erc_sw en déséquilibrant volontairement l'architecture. Nous avons effectué des mesures entre 2 nœuds avec des processeurs rapides (Pentium III) mais une interconnexion très lente (Ethernet 10 Mb/s sous TCP/IP) sur un programme synthétique de la forme présentée en figure 3. Les variables a et b se trouvent sur la même page et provoquent ainsi un faux partage de type lecture/écriture.

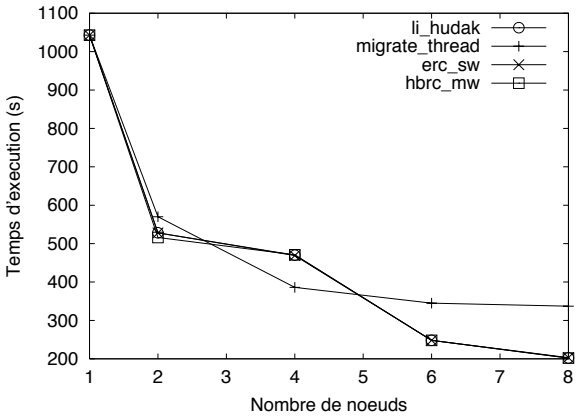

FIG. 4 : TSP sur BIP/Myrinet (17 villes).

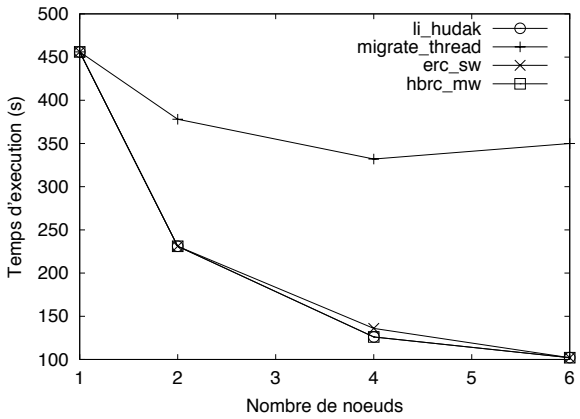

FIG. 5 : TSP sur SISCI/SCI (17 villes).

Ce fait interdit la concurrence dans le protocole li_hudak car un seul nœud y est autorisé à accéder en écriture à la page, à la différence du protocole hbrc_mw. De plus, il force le protocole li_hudak à effectuer de nombreux transferts de pages, alors que les deux autres protocoles ne font que très peu de communications. Nous avons alors observé une amélioration d'environ 30 % avec le protocole erc_sw et d'environ 50 % avec le protocole hbrc_mw. Une étude beaucoup plus fine des comportements observés est actuellement en cours dans le cadre de la rédaction de la thèse de Gabriel Antoniu.

DSM-PM2 permet ainsi de comparer dans un cadre unifié, contrôlé et portable différents protocoles de cohérence. Cette plate-forme apparaît donc comme un cadre de travail particulièrement intéressant pour les développeurs d'applications distribuées multithreads sur MVP.

Bibliographie

1. G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. In *Euro-Par 2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1039–1052, Munchen, Germany, August 2000. Springer-Verlag.
2. Gabriel Antoniu and Luc Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, San Francisco, April 2001. To appear.
3. L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–482, San Juan, Puerto Rico, April 1999. Springer-Verlag.
4. L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proceedings of the IEEE*, 87(3), March 1999.
5. A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its application in distributed shared memory systems. *J. Systems and Software*, 42(1):71–87, July 1998.
6. B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java language specification*. Addison Wesley, Second edition, 2000.
7. P. Keleher, A.L.Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software based release consistent protocols. *J. Parallel and Distrib. Comp.*, 26(2):126–141, September 1995.
8. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
9. F. Mueller. Distributed shared-memory threads: DSM-Threads. In *Proc. Workshop on Run-Time Systems for Parallel Programming (RTSPP)*, pages 31–40, Geneva, Switzerland, April 1997.
10. R. Namyst. *PM2: an environment for a portable design and an efficient execution of irregular parallel applications*. PhD thesis, Univ. Lille 1, France, January 1997. In French.
11. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE computer*, 24(8):52–60, September 1991.
12. J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Paralel and Distributed Technology*, pages 63–79, 1996.