INRIA Rhône-Alpes
Projet SIRAC

Université Joseph Fourier
U.F.R. IMA

Alexis CHEMIN

Magistère Informatique
Année scolaire 1997-1998

Rapport de stage

Outils de synchronisation sur un cluster SCI

Tuteurs :
X. Rousset de Pina
P. Koch

Sommaire

1. INTRODUCTION 4

2. LE STANDARD SCI 5

2.1 La norme SCI 5
2.2 Le pont PCI-SCI 6
2.3 Spécifications du cluster SCI utilisé 7

3. SCIOS 9

3.1 Objectifs 9
3.2 Principe de fonctionnement 9

4. ALGORITHMES DE SYNCHRONISATION 10

4.1 Les verrous 11
4.1.1 Algorithme du "Test&set" 11
4.1.2 Verrou à tickets 12
4.1.3 Verrou par messages 13
4.2 Les barrières de rendez-vous 15
4.2.1 Barrières centralisées 16
4.2.2 Barrières distribuées 18

5. IMPLANTATION SUR CLUSTER SCI 21

5.1 Partage des informations 21
5.2 Opérations atomiques sur SCI 22
5.3 Algorithmes implementés 23

6. EXPÉRIMENTATION ET RÉSULTATS 25

6.1 Tests des algorithmes de verrous 25
6.2 Tests des algorithmes de barrières. 27

7. CONCLUSION 28

8. BIBLIOGRAPHIE 29

Remerciements

Je remercie Povl Koch et Xavier Rousset de Pina qui m'ont accueilli et encadré durant ce stage, et qui ont guidé mes travaux tout au long de mon stage. Je tiens à remercier également, Emmanuel Cecchet pour ses nombreux conseils techniques, et pour l’aide qu’il m’a apportée pour la rédaction de ce rapport.

Je remercie aussi toutes les personnes du projet SIRAC, qui ont contribué à la bonne ambiance qui règne à l'INRIA Rhône-Alpes, et qui ont permis à ce stage de se dérouler dans les meilleures conditions.

1. Introduction

Le présent document détaille les activités que j'ai effectuées, dans le cadre du projet de Magistère. Ce projet s'est déroulé à l'INRIA Rhône-Alpes au sein du projet SIRAC (Systèmes Informatiques Répartis pour Application Coopératives). L'objectif du projet Sirac est de concevoir et de réaliser des services et outils pour le développement et l'exécution d'applications réparties. Son champ d'activité couvre les environnements de développement d'applications, le support système pour la construction de serveurs d'informations, et les protocoles et services pour communications mobiles. Les personnes chargées de mon encadrement pour ce stage étaient M. Povl Koch, chercheur post-doctoral et M. Xavier Rousset de Pina, professeur à l'INPG.

Une des voies explorées par le projet Sirac, est l'étude de systèmes à hautes performances réalisés en interconnectant un ensemble de machines (grappe ou cluster) par un réseau à très haut débit. Le projet utilise une plate-forme expérimentale composée de PC reliés par un réseau SCI (*Scalable Coherent Interface*, norme définie par l'IEEE). Une connexion SCI entre les bus de deux machines permet à chacune d'elle de lire ou d'écrire directement dans la mémoire d'une machine distante. Le débit atteint 70 Moctets/s, avec une latence de l'ordre de 2,5 microsecondes. Un avantage de l'écriture à distance est que le processeur de la machine à laquelle on accède n'est pas interrompu par le transfert, puisque la connexion se fait de bus à bus.

Les applications prévues sur cette grappe de machines font appel au partage de données entre processus s'exécutant sur différents processeurs. Il faut donc synchroniser les accès aux données partagées pour garantir que ces données restent cohérentes. Le réseau SCI comporte des mécanismes élementaires pour assurer la synchronisation : interruptions et opérations atomiques (*fetch&add*). La tâche qui m'a été confiée, est d'utiliser ces mécanismes élémentaires pour construire des primitives de synchronisation de plus haut niveau directement utilisables par les applications. Les opérations classiques de synchronisation sont les verrous et les barrières de rendez-vous.

Ce projet s'insère dans un projet de plus grande envergure : SciOS, qui fournit aux utilisateurs du cluster SCI, un ensemble de mécanismes de gestion de la mémoire partagée. Mon travail consiste donc à étudier comment implanter ces mécanismes classiques de synchronisation en m'appuyant sur les mécanismes fournis par la couche SCI. Ces primitives de synchronisation doivent être archivées dans une bibliothèque qui sera fournie aux programmes utilisateurs.

Il est important de présenter brièvement le standard SCI, ainsi que le principe de fonctionnement du module SciOS qui fourniront les mécanismes de bases que nous utilisons. Ceci fait l'objet des deux sections suivantes de ce document. Après avoir défini précisément les outils de synchronisation que l'on souhaite développer, quelques algorithmes de verrous et de barrières de rendez-vous, qui m'ont paru intéressants seront expliqués. Nous aborderons enfin des détails de l'implantation des algorithmes, des résultats des expérimentations effectuées pour tester ces algorithmes, et des difficultés rencontrées pour y parvenir.

2. Le standard SCI

Ce chapitre présente l'interface SCI en donnant ses caractéristiques principales et en décrivant brièvement son fonctionnement. Nous nous appuyons sur la solution consistant à interconnecter les machine via leur bus PCI à l'aide de ponts PCI-SCI car c'est ce type de matériel que nous utilisons sur notre plate-forme d'expérimentation.

2.1 La norme SCI

Depuis 1992, SCI est un standard d'interconnexion de l'IEEE ([IEEE92]). Les trois principaux objectifs sont les suivants :

• Adaptabilité *(Scalability)* : Le réseau doit fonctionner aussi bien sur des systèmes comportant peu de processeurs que ceux en comprenant un grand nombre. Il ne doit pas être optimisé pour une architecture de processeur particulière mais être un réseau d'interconnexion général.

• Cohérence : La cohérence des caches dans les systèmes à mémoire partagée distribuée doit être maintenue de façon transparente pour les processeurs par le réseau.

• Ouverture : L'interface interconnectant les nœuds ou les sous-réseaux ne doivent pas se restreindre à un principe ou une technologie particulier. Les constructeurs doivent pouvoir implanter librement leur propre interface adaptée à leurs besoins.

SCI utilise des connexions point-à-point entre les différents nœuds du réseau ce qui autorise plusieurs topologies comme les anneaux, les réseaux maillés, les réseaux à étages multiples, etc. Toutefois, la topologie en anneau est souvent retenue car elle est simple et peu coûteuse à réaliser.

Dans la pratique, la cohérence des caches est optionnelle et n'est presque jamais implantée sur les versions de SCI basées sur les bus d'entrées/sorties.

2.2 Le pont PCI-SCI

Pour profiter du standard industriel PCI[1], un pont PCI-SCI a été défini pour permettre d'interconnecter le vaste parc des machines disposant d'un bus PCI. La Figure 1 illustre l'interconnexion de deux machines via un pont PCI-SCI.

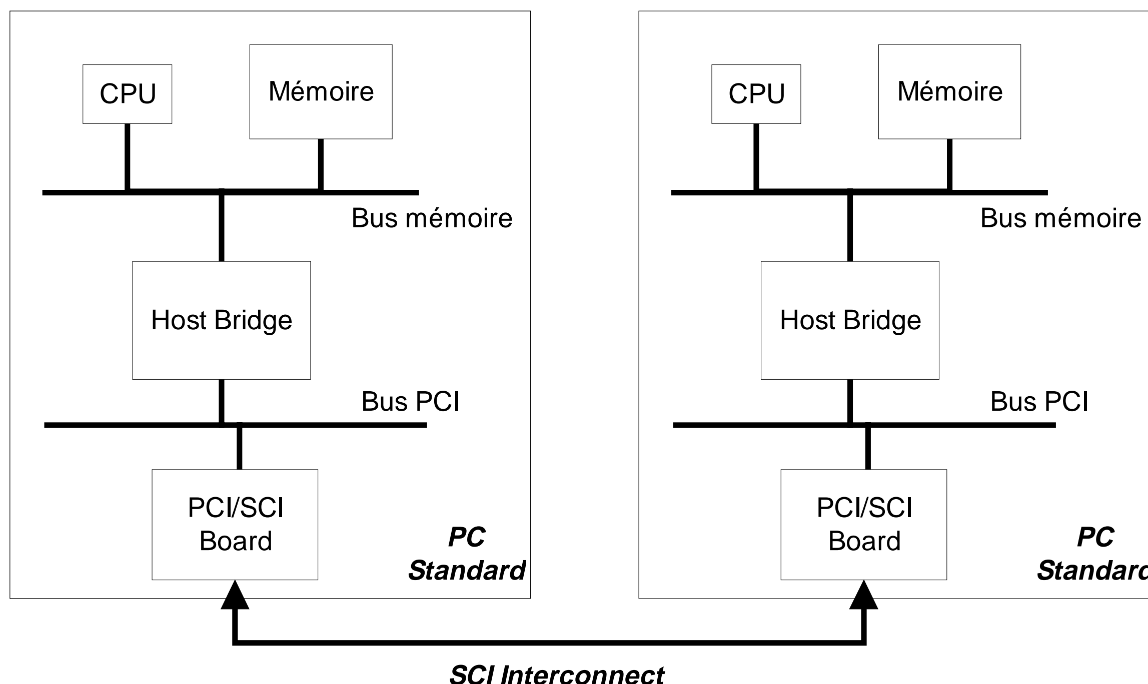

Figure 1. Interconnexion SCI via des ponts PCI-SCI

Le pont PCI-SCI permet de passer des messages avec de très faibles latences à travers de la mémoire partagée non cachée. Il est également possible d'effectuer des lectures et des écritures directement dans les mémoires distantes. Les interruptions, et transferts DMA[2] sont également disponibles.

SCI offre un grand espace d'adressage partagé de 64 bits. Les 16 premiers bits permettent de désigner le nœud (désignation d'une machine par un identificateur unique), et les 48 derniers bits adressent la mémoire physique dans le nœud.

L'adaptateur PCI-SCI de Dolphin Interconnect Solutions ([DOLPHIN96]) autorise un processeur d'un nœud d'accéder directement à la mémoire d'un nœud distant. Le processeur effectue chacune de ces opérations (lecture, écriture) en adressant la partie de son espace d'adressage physique allouée à l'interface SCI sur le bus PCI. Plusieurs portions de l'espace d'adressage physique local peuvent être projetées dans la mémoire distante d'autres nœuds.

Pour partager de la mémoire physique entre deux nœuds, un processus alloue de la mémoire physique locale. Un processus accède localement à cette mémoire en accédant aux pages virtuelles de son espace d'adressage auxquelles il est couplé. Pour qu'un processus distant accède à la mémoire partagée, il doit mettre en place deux projections en utilisant le

[1] Peripheral Component Interconnect
[2] Direct Memory Access

système d'exploitation. Tout d'abord, la mémoire distante est projetée dans l'espace d'adressage du bus PCI local en mettant à jour la table de translations d'adresses (ATT[3]) de l'adaptateur PCI-SCI avec l'identificateur du nœud distant et l'adresse physique dans la mémoire distante. Le système d'exploitation projette alors une plage d'adresses spécifique du bus PCI dans son espace d'adressage virtuel par manipulation des tables des pages.

La Figure 2 illustre le cas où le processus A projette une page de mémoire physique allouée localement par le processus B.

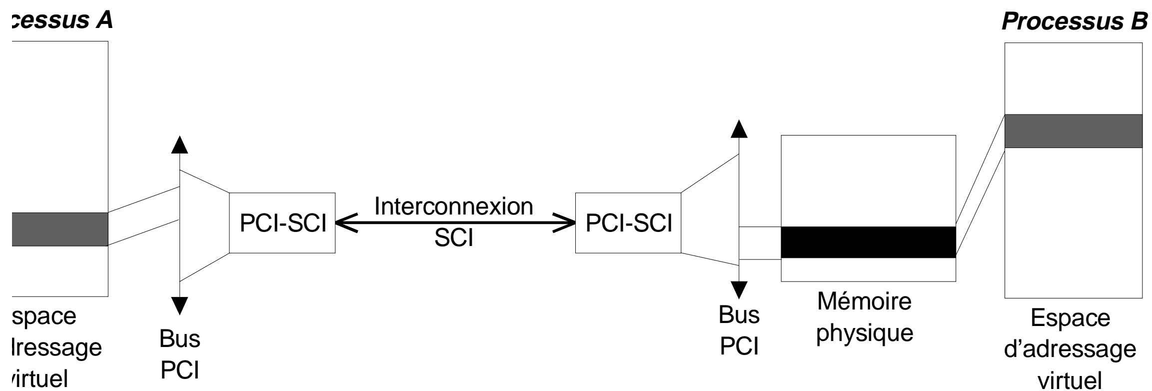

Figure 2. Projection mémoire via un pont PCI-SCI

2.3 Spécifications du cluster SCI utilisé

Un cluster SCI est un ensemble de machines interconnectées par des interfaces SCI. SCI autorise de nombreuses topologies, et pendant ce stage, notre cluster était composé de quatre Pentium II à 266MHz reliés en anneau et de deux Pentium à 200 MHz directement connectés (*back to back*). A l'heure actuelle, il est prévu d'ajouter deux bi-Pentium II à 266MHz au cluster, pour former un anneau de 8 machines.

[3] Address Translation Table

Il est également prévu de munir toutes les machines de carte PCI/SCI ayant un débit de 500 Mo/s. La plateforme SCI de l'INRIA Rhônes-Alpes est représentée par la Figure 3.

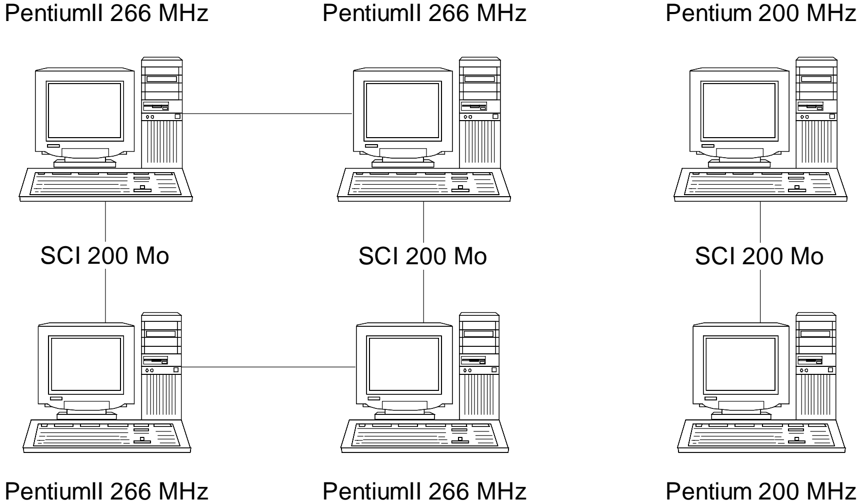

Figure 3. Cluster SCI de l'INRIA Rhônes-Alpes

3. SciOS

3.1 Objectifs

Le but de SciOS est de fournir à l'utilisateur l'abstraction d'une mémoire partagée cohérente sur une grappe de stations de travail connectées par un réseau à capacité d'adressage. L'objectif est de limiter le coût de toutes les références à la mémoire quelque soit le schéma de partage des données. Comparée aux interfaces qui favorisent l'envoi de messages, une interface de type mémoire partagée est considérée comme un moyen plus simple et plus pratique pour programmer des applications parallèles et distribuées.

SciOS s'occupe donc de la gestion des pages de mémoire partagée, comme l'allocation, la libération, la duplication, la migration des pages, le « swap » des pages vers des sites distants, ainsi que des outils de synchronisation pour l'utilisateur, dont il est question dans ce rapport.

3.2 Principe de fonctionnement

Le prototype SciOS a été conçu pour fonctionner sous Linux. Nous disposons d'adaptateurs PCI-SCI de Dolphin Interconnect Solutions, ainsi que d'un pilote disponible sous forme d'un module Linux.

Les modules Linux sont des programmes chargeables et déchargeables dynamiquement, qui constituent des extensions du noyau. Leur exécution se fait en mode noyau ce qui implique une certaine rigueur de programmation.

SciOS est également conçu sous forme d'un module Linux qui s'appuie sur le système de fichier virtuel (VFS[4]) de Linux. Ceci permet à l'utilisateur d'accéder aux fonctions de SciOS au moyen de l'interface standard de VFS, c'est-à-dire par des appels comme open(), ioctl(), … Typiquement, un programmeur d'application pourra accéder à une zone de mémoire partagée gérée par SciOS, en effectuant une création ou une ouverture de fichier, puis en couplant la mémoire partagée à ce fichier. Un exemple d'utilisation des ces fonctions de SciOS est donné en annexe A.

Une partie de SciOS s'occupe de la gestion des messages actifs qui permettent d'échanger rapidement des messages au sein du cluster et d'établir de petits protocoles de communication pour réaliser certaines opérations complexes.

[4] Virtual File System

4. Algorithmes de synchronisation

Les stratégies de mise en œuvre de synchronisation peuvent être divisées en deux catégories : *bloquantes* qui placent le processus dans un état d'attente d'événement et *attente active* dans laquelle le processus teste continuellement l'état d'une variable partagée. La synchronisation par attente active est fondamentale pour la programmation parallèle sur des système à mémoire partagée et est préférable à la stratégie par blocage, qui induit un surcoût par rapport au temps d'attente prévu dû au scheduling, lorsque le processeur n'est pas requis pour d'autres processus.

Les outils de synchronisation basés sur l'attente active les plus répandus sont les verrous et les barrières. Les verrous fournissent un moyen de réaliser une exclusion mutuelle (donc assure qu'un seul processus à la fois ne peut accéder à une zone de mémoire partagée). Ils constituent le fondement de la réalisation d'autres outils de synchronisation ayant une sémantique plus évoluée tels que les sémaphores ou les moniteurs. Les barrières permettent de synchroniser des processus s'exécutant sur des sites distants à un moment précis de leur exécution. Dans cette section sont donc décrits des algorithmes concernant les verrous et les barrières de rendez-vous qui m'ont parus intéressants. Les verrous sont généralement employés pour de petites sections critiques, et peuvent être exécutés un nombre de fois important. Nous attachons donc beaucoup d'importance aux performances des algorithmes implantés. Les implantations typiques d'une synchronisation basée sur l'attente active tendent à produire un nombre important de transactions sur le réseau, ce qui cause une dégradation de performances notamment lorsque le nombre de machines est important. Cette section présente certains algorithmes permettant de résoudre en partie ce problème.

Les méthodes de synchronisation sont, d'une manière générale, basées sur l'accès en lecture et/ou écriture à une zone de mémoire partagée qui représente l'état global de l'objet de synchronisation (verrou ou barrière). Les processus sont donc en concurrence pour la modification de cette partie de mémoire partagée et, pour des raisons de cohérence, ces accès doivent être atomiques. On appelle *opération atomique*, une opération d'accès et de modification d'une variable en mémoire pendant laquelle aucun processus concurrent ne peut altérer l'état de cette variable. Les premières méthodes de synchronisation utilisaient uniquement les opérations atomiques de lecture et d'écriture qui s'avèrent être relativement coûteuse en temps et en espace. L'importance de la synchronisation pour les applications réparties et les programmes parallèles ont permis l'évolution au cours du temps de ces opérations atomiques. En effet, les processeurs actuels incluent tous, parmi leur jeu d'instructions, des opérations atomiques plus sophistiquées permettant des stratégies de coordination de processus plus simples et plus rapides. Ces opérations permettent généralement d'effectuer un ensemble d'instructions de la forme « lecture - modification - écriture » d'une variable, et ceci de manière atomique. Cette section présente donc des algorithmes basés sur l'utilisation de telles opérations. Dans une autre section, nous discutons de l'implantation de tels algorithmes sur le cluster SCI.

Concernant la description des algorithmes, il faut préciser que la synchronisation se fait au niveau des machines et non pas au niveau des processus, c'est à dire qu'il n'y a qu'un seul processus par site qui peut utiliser les verrous ou les barrières. Ainsi, nous employons aussi bien le terme site que machine ou processus pour désigner un site participant à la synchronisation.

4.1 Les verrous

Dans cette section sont décrits deux algorithmes détaillés dans [MCS91], qui réalisent l'acquisition et la libération d'un verrou par mécanisme d'attente active. Un algorithme basé sur l'échange de messages entre les sites est également présenté. Tous supposent l'existence d'un environnement à mémoire partagée supportant des opérations atomiques évoluées. Certains algorithmes permettant d'obtenir des verrous, ne nécessitent que l'atomicité des opérations *read* et *write* [LAMPORT87], mais la relative complexité de ces algorithmes nous a poussé à explorer les façons de réaliser des opérations atomiques évoluées sur le cluster SCI. [MCS91] présente également le MCS lock, un algorithme distribué qui permet à chaque processus participant d'effectuer des attentes actives sur des variables locales uniquement. Cet algorithme n'est pas détaillé ici, car il ne pouvait être implanté sur le cluster SCI (cf. section 5, « Implantation sur cluster SCI »). Les verrous sont des objets manipulés par deux procédures :

lock_acquire(lock l) : acquisition du verrou
lock_release(lock l) : libération du verrou

On assure qu'un processus s'exécutant entre ces deux procédures est en exclusion mutuelle. Pour évaluer les différents algorithmes de verrou, les critères suivants sont pris en compte :

- Nombre d'accès distants
- Possibilité de famine pour les processus participants

4.1.1 Algorithme du "Test&set"

Le plus simple des algorithmes de verrous, très employé dans la pratique, utilise une boucle d'attente active sur une variable booléenne indiquant si le verrou est libre ou non. Chaque processus exécute une opération atomique de *test_and_set* (test d'une variable et modification de sa valeur selon le résultat du test), dans une tentative de modification de la variable de faux à vrai, afin d'acquérir le verrou. Le processus possédant le verrou peut le relâcher en positionnant cette variable à la valeur vrai.

Un des principaux défauts de cette méthode est que les processus en compétition pour l'acquisition du verrou passent leur temps à accéder à la mémoire partagée par une opération relativement coûteuse (lecture, modification, écriture). Pour éviter une dégradation trop importante du réseau, un délai est introduit entre les tentatives d'acquisition du verrou par un processus. Une bonne approche consiste à utiliser un délai croissant de manière exponentielle entre les différentes tentatives.

Un autre inconvénient de cet algorithme est qu'il n'y a pas d'ordre précis quant à l'obtention du verrou par les processus, et ceci implique un risque de famine. L'algorithme du verrou *test&set* est donné ci-dessous :

```
enum {unlocked, locked} lock;

void lock_acquire ( lock* l)
{
      int delay = 1;
      while ( test_and_set(l,locked) == locked )
            {
            pause(delay);
            delay = delay*2;
            }
}

void lock_release ( lock* l)
{
      *l = unlocked;
}
```

4.1.2 Verrou à tickets

L'algorithme suivant permet en théorie de diminuer le nombre d'accès distants par rapport à l'algorithme précédent car chaque processus en compétition pour l'acquisition du verrou peut évaluer son temps d'attente avant l'obtention du verrou. Cet algorithme assure un ordre FIFO sur l'acquisition du verrou, ce qui élimine les possibilités de famine.

Le verrou à ticket est constitué de deux compteurs, l'un contenant le nombre de requêtes pour l'acquisition du verrou, et l'autre le nombre de fois que le verrou a été libéré. Un processus acquiert le verrou en effectuant une opération atomique *fetch_and_inc* (lecture puis incrémentation d'une variable entière), sur le compteur des requêtes et en attendant que le résultat (son ticket) soit égal à la valeur du compteur des libérations. Afin de réduire le nombre d'accès distants inutiles, un délai est introduit entre les tests d'égalité des compteurs, comme dans l'algorithme précédent. Cependant, les processus obtiennent le verrou selon un ordre FIFO, et la différence entre le deux compteurs indique le nombre de processus en attente pour le verrou, donc le délai d'attente peut être calculé en fonction de cette différence et d'une estimation du temps nécessaire à un processus pour exécuter sa section critique. Malheureusement, cette durée est rarement connue dans la pratique. Nous pouvons cependant estimer que la durée de la section critique d'un processus est supérieure ou égale au temps nécessaire pour acquérir et libérer le verrou. Une bonne approche consiste donc à calculer le délai en fonction de ce temps minimal de section critique et de la différence entre les deux compteurs. l'algorithme du "ticket lock" est donné page suivante.

```
struct {
      int next_ticket;
      int now_serving;
      } lock = {0,0};

void lock_acquire (lock* l)
{
      int my_ticket;
      /* acquisition d'un ticket */
      my_ticket = fetch_and_inc(&l->next_ticket);
      do    {
            /* pause proportionnelle a la difference
            entre les deux valeur : */
            pause(my_ticket - l->now_serving);
            }
      while (my_ticket != l->now_serving);
}

void lock_release (lock* l)
{
      l->now_serving ++;
}
```

4.1.3 Verrou par messages

Il existe deux approches pour réaliser les procédures de manipulation de verrous basées sur les messages. La première consiste à envoyer un message d'acquisition de verrou à tous les sites concernés, et d'attendre une réponse positive de ceux-ci. La résolution des conflits entre sites peut se faire soit en imposant un ordre sur les sites (anneau virtuel) ou en mettant en place une horloge logique pour la datation des requêtes. Dans ces deux cas, il est nécessaire de connaître tous les sites susceptibles d'être en compétition. J'ai voulu expérimenter une autre approche, qui consiste à utiliser un serveur de verrou. Ainsi, chaque site désirant acquérir un verrou émet un message vers un site serveur qui gère les requêtes suivant un ordre FIFO, selon leur arrivée; le site candidat attend ensuite un message de la part du serveur autorisant d'entrer en exclusion mutuelle. Pour relâcher le verrou, le site qui le détient émet un message de libération vers le site serveur. Le site serveur doit être déterminé avant que toute requête puisse avoir lieu.

Un système de messages actifs à été implanté dans SciOS. Il s'agit en fait, de la possibilité pour un site de déclencher une interruption sur un autre site. Le traitement de l'interruption consiste à aller lire dans une zone de mémoire partagée les données du message. Le problème est que le traitement du message est en fait un traitement d'interruption, ce qui interdit de forcer l'ordonnancement pendant le traitement, par ailleurs, il est impossible de déclencher une interruption sur son propre site. Nous sommes cependant assuré que tous les messages émis sont traités, et que le traitement d'un message n'est pas interrompu.

Lorsque le serveur de verrous reçoit un message, il renvoie donc immédiatement un message d'acceptation si le verrou est libre, sinon, il mémorise la requête qu'il pourra traiter ultérieurement lorsqu'il recevra un message de libération de verrou. Mais si l'on désire que le site serveur de verrous puisse être candidat, il faut que la procédure locale d'acquisition du verrou soit synchronisée avec la réception de requêtes d'acquisition. Une méthode possible est de réaliser un système de producteurs - consommateurs à l'aide d'un tableau dont les indices sont incrémentés de manière atomique par une opération *fetch&inc*. Ainsi, à la réception d'une requête d'acquisition, l'indice des requêtes reçues est incrémenté et on mémorise le numéro du site ayant émit la requête. Lorsque le serveur désire acquérir le verrou, il fait de même puis effectue une attente active sur une variable booléenne. Le consommateur de requêtes doit être

un processus léger (*thread*) qui effectue une attente active sur la différence entre les deux indices de gestion du tableau des requêtes (car ils indiquent si des requêtes ont été reçues). On présente ci-dessous cet algorithme, pour un seul verrou :

```
/* sur le site serveur de verrous : */

/* tableau des requetes recues : */
int requetes[MAX_SITES] = {-1,...,-1};
/* indices de gestion du tableau : */
int req_in = 0, req_out = 0;
/* booleen pour acquisition locale : */
int lock_ready = FALSE;
/* booleen indiquant que le verrou est libre */
int lock_released = TRUE;

/* reception d'une requete d'acquisition,
   (traitement qui ne peut etre interrompu) : */
void lock_acquire_requete(int num_candidat)
{
      int index = fetch_and_inc(&req_in);
      requetes[index%MAX_SITES] = num_candidat;
}

/* reception d'une requete de liberation,
   (traitement qui ne peut etre interrompu) : */
void lock_release_requete()
{
      lock_released = TRUE;
}

/* procedure locale d'acquisition du verrou */
void lock_acquire()
{
      int index;
      lock_ready = FALSE;
      index = fetch_and_inc(&req_in);
      while (lock_ready == FALSE) schedule();
}
```

```
/* procedure locale de libération de verrou */
void lock_relase()
{
      lock_released = TRUE;
}

/* thread consommateur de requetes : */
void thread_consommateur()
{
while(1)
      if ((req_in > req_out) && lock_released)
            {
            int site = requetes[req_out%MAX_SITE];
            req_out++;
            lock_released = FALSE;
            if (site == -1) /* acquisition locale */
                  lock_ready = TRUE;
            else /* requete distante */
                  send_msg(site, LOCK_ACQ_OK);
            }
      else schedule();
}
```

4.2 Les barrières de rendez-vous

Les barrières de rendez-vous sont très utilisées dans les applications parallèles. Elles permettent de stopper un processus à un point précis de son exécution et d'attendre que d'autres processus aient également atteint un point spécifique de leur exécution, avant de reprendre le cours de son exécution. Elles assurent que certaines phases d'un algorithme distribué ne seront pas atteintes par un processus avant que celui-ci n'y soit autorisé. Les procédures de manipulation de l'objet barrière pour l'utilisateur sont :

`barrier_init(&barrier, int num_proc)`
: L'initialisation de la barrière. C'est là qu'est spécifié le nombre de processus devant participer la barrière

`barrier_rdv(&barrier)`
: Le rendez-vous proprement dit qui permet aux processus de se synchroniser

Dans les systèmes à mémoire partagée, les barrières sont généralement réalisées en deux phases. Dans la phase de *réduction* (collecte des arrivées des processus), chaque processus signale aux autres processus son arrivée à la barrière. Vient ensuite la phase de *distribution*, ayant lieu lorsque tous les processus concernés ont atteint la barrière. Cette phase autorise chaque processus à reprendre le cours normal de son exécution. Comme pour les verrous, les objets barrières sont constitués de variables partagées, qui représentent l'état global de la barrière, et de variables locales. Nous distinguons les barrières centralisées, pour lesquelles les variables partagées sont rassemblées sur une seule machine, et les barrières réparties, où l'état global de la barrière est constitué de plusieurs zones de mémoire partagée, réparties sur les différents sites.

Pour l'évaluation des algorithmes de barrières, nous tenons compte des critère suivants :

- Le nombre de transactions entre les machines
- La quantité de mémoire partagée nécessaire
- Le type d'opérations atomiques utilisées

4.2.1 Barrières centralisées

Un principe de réalisation triviale d'une barrière de rendez-vous est présenté ici. Chaque processus se rendant à la barrière accède à un compteur partagé (qui indique le nombre de processus présents à la barrière) de manière exclusive et l'incrémente grâce à l'opération atomique *fetch_and_inc*. Chaque processus effectue ensuite une attente active sur la valeur du compteur jusqu'à ce que tous les processus soient arrivées.

```
/* Initialisation */
const int P;      /* nombre total de noeuds */
int count = 0;    /* nombre de nodes a la barriere,
                           (variable partagee) */

void barrier()
{
      int n;
      n = fetch_and_inc(&count);  /* atomique */
      if (n == P - 1) count = 0;  /* reinitialisation */
      else while (count < P){ pause(); }
}
```

Bien que permettant de comprendre le principe d'une barrière centralisée, cet algorithme est faux car il pose un problème lors de la réinitialisation du compteur. En effet, le dernier processus se présentant à la barrière peut incrémenter le compteur et lui donner ainsi la valeur de P, puis le refaire basculer à 0 alors que d'autres processus attendent que le compteur ait la valeur de P. Ceci met en évidence une propriété nécessaire pour le bon fonctionnement d'une barrière : tout processus en attente dans la barrière doit être débloqué avant la réinitialisation de la barrière. En modifiant l'algorithme, il est possible d'éviter la réinitialisation du compteur :

```
void barrier()
{
      fetch_and_inc(&count);/* atomique */
      while (count % P) { pause(); }
}
```

Mais le problème demeure lorsque les processus doivent effectuer deux passages successifs à la barrière, car le compteur peut être incrémenté deux fois de suite alors que des processus se trouvent dans la boucle d'attente.

4.2.1.1 Barrière à double compteurs

Une solution simple pour résoudre le problème posé, est d'utiliser une barrière à deux compteurs. Ceci est équivalent à utiliser deux barrières identiques à celles décrites précédemment. Ainsi, lorsqu'un processus sort de la première boucle d'attente, il va se bloquer dans la seconde, ce qui lui interdit de modifier le premier compteur tant que tous les processus ne sont pas sortis de la première boucle.

```
int count1 = 0;  /* deux compteurs partages */
int count2 = 0;

void barrier()
{
      fetch_and_inc(&count1);      /* première barrière */
      while (count1 % P)     { pause(); }

      fetch_and_inc(&count2);      /* seconde barrière */
      while (count2 % P)     { pause(); }
}
```

Cet algorithme fut le premier à être implanté dans SciOS pour les besoins urgents des programmes de tests. Il sert donc d'algorithme de référence pour l'évaluation des autres algorithmes.

4.2.1.2 Inversion de sens

La barrière à inversion de sens décrit dans [MCS91] est un algorithme plus élégant que l'algorithme vu précédemment. L'idée est de « traverser » la barrière dans un sens opposé au sens du dernier passage dans la barrière. Un processus arrivant à la barrière, incrémente le compteur et attend jusqu'à ce que la variable *sens* ait une valeur différente que lors de son dernier passage dans la barrière. Le dernier processus réinitialise le compteur et inverse le sens. Les passages successifs à la barrière n'interfèrent pas entre eux car toutes les opérations sur le compteur s'effectuent avant l'inversion de la variable *sens* qui libère les processus en attente. Ce principe d'inversion de sens est par ailleurs utilisé dans d'autres algorithmes.

```
int count = 0;          /* variable partagee */
int sense = 1;          /* booleen partage par tous les sites :
                           sens de la barriere */
int local_sense = 1;    /* booleen local au site */

void barrier()
{
      /* inversion du sens local de la barriere */
      local_sense = !local_sense;
      if ( fetch_and_inc(&count) == P - 1 )  {
            count = 0;
            sense = local_sense;
            }
      else while(sense != local_sense) { pause(); }
}
```

Le problème majeur posé par les algorithmes centralisés est que l'attente active se fait sur une unique variable partagée située sur un seul site. Ceci provoque un nombre important d'accès distants difficiles à estimer car, en pratique, les processus n'arrivent pas simultanément à la barrière. Les barrières distribuées permettent de minimiser ce phénomène. L'avantage des barrières centralisés est qu'elles utilisent une quantité constante de mémoire partagée, mais on doit accéder à celle-ci via une opération atomique de type *fetch_and_inc*.

4.2.2 Barrières distribuées

Le principe des barrières distribuées est de partager l'ensemble des variables qui constituent la barrière sur différents sites participant à la barrière. Ainsi les accès distants ne sont pas tous dirigés vers un seul site, et les attentes actives des processus en attente dans la barrière deviennent locales, ce qui réduit le trafic sur le cluster et permet de meilleurs performances.

4.2.2.1 Barrière arborescente

L'état global de la barrière est représenté par un arbre. Le rendez-vous se fait par groupe : à son arrivé, chaque site modifie l'état du noeud auquel il est attaché. Lorsque tous les membres d'un groupe sont arrivés, l'information est propagée vers le niveau supérieur de l'arbre. Lorsque l'information atteint la racine de l'arbre, cela signifie que tous les processus ont rejoint la barrière. Chaque noeud « réveille » ses nœuds fils pour sortir de la barrière. [MCS91] décrit deux algorithmes de barrière arborescente, mais nous ne les détaillerons pas ici, puisqu'ils n'ont pas été implantés.

Cependant, l'avantage de ce type de barrière en arbre est que certaines transactions peuvent se faire en parallèle, mais ceci dépend aussi de la topologie du réseau utilisé. [YCTK98] explique par exemple, comment construire une barrière arborescente de synchronisation sur un réseau maillé à deux dimensions.

.

4.2.2.2 "Dissemination barrier"

Cet algorithme tire son nom de la façon de disséminer de l'information parmi un ensemble de processus. Chaque site participant effectue une séquence de $\log_2$ P synchronisations (où P est le nombre de sites). La synchronisation se fait par tours au cours desquels un site (appelons le p1), va émettre un message vers un site précis (p2), signalant ainsi son arrivée à la barrière, puis il va attendre un message d'un autre site (p3). Au tour suivant, ce processus p1 va émettre un nouveau message vers un autre site (p4), signalant à nouveau son arrivé à la barrière, mais également le fait que le site p2 est lui aussi arrivé à la barrière (puisque p4 sait que p1 ne peut lui envoyer de message avant qu'il n'en ait reçu un de p2). Cette méthode permet de réduire le nombre de transaction entre les sites car l'information est propagée entre les sites. Il faut néanmoins déterminer quels sont les sites devant se synchroniser à chaque tour, et ceci de manière à minimiser le nombre de tours de synchronisation.

Dans l'algorithme présenté dans [HFM88], à chaque tour *k*, le processus *i* se synchronise avec le processus $(i+2^k)$ mod P (il lui envoie un message signalant son arrivé à la barrière), et attend donc l'arrivé du processus $(i-2^k+P)$ mod P. [HFM88] prouve que chaque processus doit émettre et recevoir exactement $\log_2$ P messages pour que toute l'information concernant les arrivées soit disséminée parmi tous les processus.

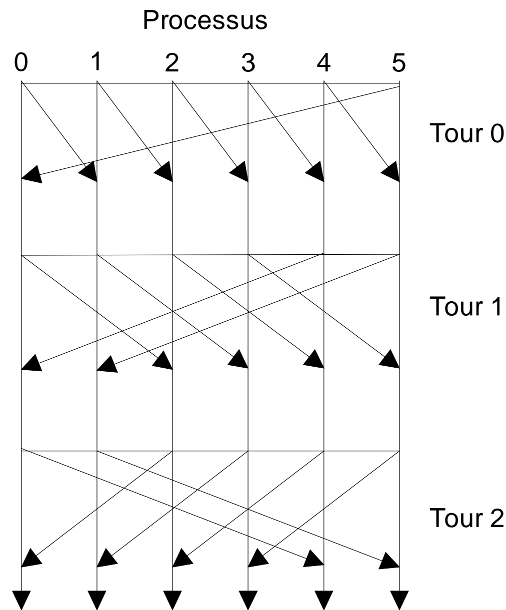

Figure 4. Communications entre 6 processus participant à une barrière disséminée

La Figure 4 donne un exemple des signaux émis pour une barrière à 6 processus. Si l'on considère le processus p0, à la fin du premier tour il sait que le processus p5 est arrivé. Au deuxième tour, il reçoit un message de p4, lui indiquant que p3 et p4 sont arrivés. Enfin au troisième tour, le message émis par p2 lui permet de savoir que p1 et p2 sont arrivés, et avec les informations collectées précédemment, il sait que tous sont arrivés à la barrière. L'algorithme est détaillé ci-dessous :

```
typedef struct _flags_st {
      int myflags[LOG_P][2];      /* tableau de booleens */
      int* partnerflags[LOG_P][2];
      } flags;

int parity = 0;         /* local */
int sense = 1;          /* booleen local */
flags* localflags;      /* pointeur sur flags locaux */
flags allnodes[P];      /* tableau de flags partage, mais
                             allnodes[i] est accessible
                             localement par le site i */
```

```
/*
initialisation :
      sur le site i :
            localflags = &allnode[i];

      pour tout i, r, k :
            allnodes[i].myflags[k][r] = 0;
            calcul de j, site avec lequel on se synchronise
            au tour k :
                  j = (i+(1<<k))%P;
            allnodes[i].partnerflags[k][r] =
                                    &allnode[j].myflags[k][r];
*/

void dissemination_barrier
{
      int tour;
      for (tour=0; tour < LOG_P; tour++)      {
            /* emission d'un message d'arrivee */
            *(localflags->partnerflags[tour][parity]) = sense;
            /* attente de reception d'arrivee (local spin) */
            while (localflags->myflags[tour][parity] != sense)
                  pause();
            }
      if (parity) sense = !sense;
      parity = 1 - parity;
}
```

Afin d'éviter les interférences entre deux passages successifs, deux ensembles de variables pour la mémorisation des signaux reçus sont utilisés. La variable *parity* permet l'alternance d'un ensemble à l'autre. L'inversion du sens de la barrière grâce à la variable *sense* permet ici d'éviter la réinitialisation des variables après chaque passage à la barrière. Comme pour l'algorithme en arbre, toutes les attentes actives se font sur des variables partagée mais locales. La quantité de mémoire nécessaire pour cet algorithme est en $O(P \log_2 P)$.

5. Implantation sur cluster SCI

5.1 Partage des informations

Lorsqu'un utilisateur désire utiliser les outils de synchronisation, il doit créer, ou ouvrir un même fichier sur chaque site. Ce fichier est ensuite couplé à une zone de mémoire virtuelle du processus sur une machine donnée. Le module SciOS gère le couplage entre cette zone de mémoire virtuelle et les pages de mémoire physiques partagées par tous les sites. L'utilisateur peut ensuite effectuer des opérations de synchronisation en utilisant des opérations d'IOCTL standards (procédures de contrôle des entrées - sorties, permettant dans notre cas, de réaliser diverses opérations sur les fichiers). A la création de l'inode du fichier une page de mémoire physique est allouée sur le site qui créé le fichier, pour contenir les informations concernants les verrous et les barrières. C'est sur cette page que sont placées les informations qui doivent être partagées par tous les sites. Cette page permet à l'utilisateur de disposer de 128 verrous et 128 barrières pour chaque fichier. Lorsqu'un utilisateur désire utiliser un verrou ou une barrière, il doit appeler une procédure d'ioctl, en passant en paramètre l'index du descripteur de fichier, une constante indiquant l'action qu'il désire effectuer sur l'outil de synchronisation, ainsi qu'un numéro identifiant le verrou ou la barrière. Un exemple d'utilisation d'un verrou associé à un fichier est donné ci-dessous :.

```
int lock_id = 5; /* identificateur du verrou */
fd = open(...);  /* creation d'un fichier partage */

...                    /* couplage memoire / fichier, etc. */

/* acquisition d'un verrou associe au fichier */
ioctl(fd, SCIOS_LOCK_ACQUIRE, &lock_id);

/* liberation du verrou */
ioctl(fd, SCIOS_LOCK_RELEASE, &lock_id);
```

Certains algorithmes de barrière nécessitent l'emploi de variables à la fois partagées mais locales à un site précis. Pour ces variables, une page physique est allouée sur chaque site, mais il faut également que chaque site puisse atteindre n'importe laquelle de ces pages. Pour cela, chaque structure constituant une barrière possède une page représentant un tableau d'adresses de pages physiques allouées sur divers sites : la page d'index i dans ce tableau est allouée sur le site i. Ainsi ces pages permettent de représenter des variables partagées et locales à un site spécifique.

L'affectation d'un numéro d'identification d'un site se fait lors de l'initialisation de la barrière, en exclusion mutuelle car ce numéro doit être unique. Le premier site qui exécute la procédure d'initialisation de la barrière obtient l'identificateur 0, et il est chargé d'allouer des pages sur chacun des sites, ainsi que de créer le tableau contenant des pointeurs ces pages locales. Les autres sites qui exécutent ensuite la procédure d'initialisation font une copie de ce tableau, ce qui leur permet de connaître l'emplacement des information locales de chaque site. La Figure 5 donne un exemple de l'allocation des pages physiques.

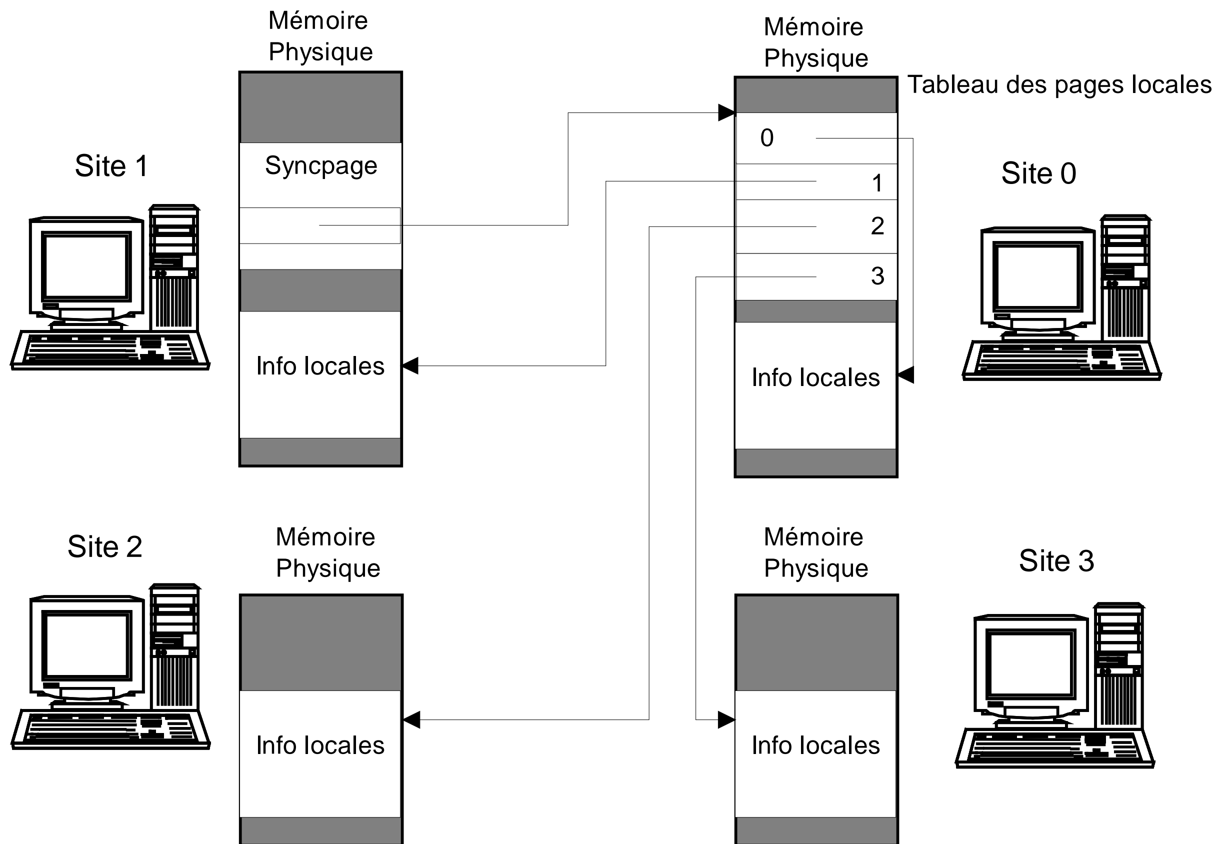

Figure 5. Exemple d'allocation des pages pour une barrière

5.2 Opérations atomiques sur SCI

Le standard SCI prévoit l'utilisation d'un certain nombre d'opérations atomiques évoluées *fetch&add*, *compare&swap*, *bounded_add*, etc. Malheureusement ces opérations n'ont pas été implantées sur les cartes PCI-SCI. La seule façon de réaliser une opération atomique est d'effectuer un couplage spécifique de la mémoire partagée, avec lequel tout accès à la mémoire (lecture et écriture) a la sémantique d'une opération atomique de type *fetch_and_inc*.

Voici un exemple de programme permettant d'effectuer une opération atomique *fetch_and_inc* :

```
int *adr_vir, *adr_phy;
int var;

      /* couplage de la page physique a une adresse virtuelle selon un
      couplage 'special' */
adr_vir = page_map_fetch_and_inc( adr_phy );
      /* var prend la valeur de adr_vir[0], et
      adr_vir[0] est incrementee :*/
var = adr_vir[0];
page_unmap_fetch_and_inc( adr_vir);

/* couplage normal de la page physique : */
adr_vir = page_map_kernel( adr_phy );
      /* acces normal a l'adresse adr_vir */
      while (adr_vir[0] != P) { }

page_unmap_kernel( adr_vir );
```

5.3 Algorithmes implementés

Parmi les algorithmes de verrous présenté, nous avons choisi d'implanter le « ticket lock », car d'une part, c'est celui qui présente manifestement le plus d'intérêt puisqu'il permet d'anticiper l'attente d'un site avant qu'il puisse obtenir le verrou. D'autre part, cet algorithme est basé sur l'utilisation de l'opération *fetch_and_inc* qui est, comme vu précédemment, la seule opération atomique réalisable sur SCI. Les expérimentations ont eu pour but d'appréhender la façon dont évoluait le temps d'attente en fonction du nombre de prétendants au verrou. Les résultats de ces expérimentations sont détaillés dans la section suivante. Une version simple des verrous par messages à été implementée également, dans laquelle le serveur de verrou ne participe pas à l'obtention du verrou.

L'algorithme MCS Lock, décrit dans [MCS91] nécessite de connaître avant l'acquisition du verrou, quels sont précisément les sites qui désirent participer à la synchronisation, puisqu'il faut créer des liens entre les sites participants. Ceci impose de réinitialiser le verrou entre deux utilisations si les participants sont différents, or nous ne désirons pas utiliser les verrous autrement que par les procédures *lock_acquire* et *lock_release*, sans préciser les sites. Par ailleurs, l'opération *fetch_and_store* n'est pas réalisable. Par contre, pour les barrières, l'utilisation d'une procédure d'initialisation de la barrière est autorisée. J'ai donc implanté plusieurs algorithmes : deux barrières centralisées, la barrière utilisant deux compteurs qui sert de référence pour les mesures, et la barrière à inversion de sens. Aucun algorithme basé sur la représentation de la barrière sous forme d'arbre n'a pas été implanté pour cause de manque de sites disponibles (4 seulement pour la création d'un arbre digne de ce nom). L'algorithme « dissemination barrier » à été implanté pour comparer une méthode distribuée face aux méthodes centralisées, ainsi qu'un autre algorithme très voisin, de notre cru, que je nomme barrière distribuée simple, dans lequel chaque site envoie un message à un seul site maître qui attend l'arrivée de tous les sites, puis renvoie à chacun un message leur permettant de quitter la barrière. Chaque site esclave n'envoie donc qu'un seul message par passage, et attend qu'un seul message en provenance du maître. Il est intéressant de comparer les performances d'un tel algorithme avec celles de la barrière disséminée, pour un petit nombre de processus participants. L'utilisation de barrières distribuées permet théoriquement de meilleures performances par rapport aux barrières centralisées, surtout pour un nombre élevé de machines. Mais elles nécessitent une phase

d’initialisation plus lourde, et celle-ci doit survenir dès que l’utilisateur désire changer le nombre de sites participant à la barrière. En contrepartie, pour les barrières centralisée décrites, le nombre de sites devant se synchroniser peut être spécifié pour chaque passage à la barrière, sans la réinitialiser.

L’algorithme de dissémination présente quelques problèmes quant à l’initialisation de la barrière. En effet, chaque site participant à la barrière doit posséder un numéro d’identification unique, qui ne peut être représenté par le numéro d’identification de nœud fourni par SciOS puisque tous les sites ne participent pas à la barrière. Pour y parvenir, l’initialisation se fait en exclusion mutuelle, grâce à des verrous ; ainsi chaque processus commence par tirer un numéro d’identification. Le premier processus exécute le code ayant trait à l’allocation et l’initialisation des variables partagées, puis tous les processus initialisent leurs variables locales non partagées respectives. Le dernier processus qui participe à l’initialisation de la barrière replace la variable d’identification à zéro pour la prochaine initialisation.

6. Expérimentation et résultats

Cette section donne et explique les résultats des expériences réalisées pour évaluer les performances des algorithmes implantés. Le listing des implantations est donné en annexe ainsi que les programmes de tests et les résultats des mesures effectuées.

6.1 Tests des algorithmes de verrous

Avant d'implanter les algorithmes basés sur l'opération atomique *fetch_and_inc*, il a fallu d'abord s'assurer qu'elle possédait bien son caractère d'atomicité. Ceci a été testé par un simple programme effectuant un très grand nombre de fois l'incrémentation d'une variable en utilisant l'opération *fetch_and_inc*. Le programme étant exécuté par deux sites distants, nous vérifions que la valeur finale de la variable partagée correspond bien à deux fois le nombre d'itérations effectuées par un processus.

Pour les algorithme de verrous, j'ai réalisé un programme de test, qui exécute en boucle une exclusion mutuelle à l'aide d'une acquisition et d'une libération de verrou. J'ai mesuré pour les verrous, le temps moyen pour l'acquisition, le temps moyen passé en attente de libération du verrou, ainsi que le temps moyen pour libérer le verrou. Le programme de test permet également de simuler une section critique plus longue en ajoutant un délai entre l'acquisition et la libération du verrou. Les sites participants sont synchronisés au début du programme à l'aide d'une barrière, afin de créer des conflits pour l'acquisition du verrou. L'idée était de voir comment évoluait le temps pour acquérir le verrou en fonction du nombre de sites participants.

Pour comparer rapidement les algorithmes de verrous, nous constatons que l'algorithme des messages, bien qu'implanté de façon simple, donne de plus mauvais temps que l'algorithme du verrou à tickets (66µs pour pour l'acquisition du verrou contre 35µs pour le ticket lock), de plus le temps semble croître plus rapidement que celui du ticket lock en fonction du nombre de machines. Cependant, il faut rappeler que l'on ne dispose que de quatre machines, ce qui ne permet pas d'établir clairement l'évolution du temps d'acquisition des verrous en fonction de leur nombre.

	Données locales (maître)		Données distantes (esclave)	
	Acquisition	Exc. mutuelle	Acquisition	Exc. mutuelle
1 site	22.172	33.32	35.272	57.402
2 sites	61.621	73.42	36.386	58.698
3 sites	82.605	95.036	57.199	79.444
4 sites	127.947	141.698	103.26	125.769

Acquisition : temps (µs) pour l'acquisition d'un verrou
Exc. mutuelle : temps (µs) pour l'acquisition et le relâchement d'un verrou

Figure 6. Résumé des temps d'acquisition d'un verrou (à tickets)

Pour le "ticket lock", dont la Figure 6 donne un résumé des mesures, nous remarquons que le temps mis pour acquérir le verrou est de 22µs si les données sont accessibles localement, et de 35µs si les données sont sur un site distant, et ceci pour un site (pas de conflit). En revanche, lorsque des sites sont en conflit pour l'acquisition, nous constatons que le site possédant les données localement (le maître) passe plus de temps qu'un site distant (un esclave).

J'explique ceci par le fait que le site maître n’est en compétition qu’avec que des sites esclaves, dont le temps passé en exclusion mutuelle est plus long que celui du site maître. En revanche, lorsqu’un site esclave désire acquérir le verrou, il y a toujours parmi ses concurrents le site maître dont le temps d'exclusion mutuelle est plus court. Ce qui expliquerait pourquoi la différence de temps entre les acquisitions d'un maître et d'un esclave est constante (environ 23µs).

La constante multiplicative pour le calcul du délai utilisé dans le "ticket lock" (cf. paragraphe 4.1.2, « Verrou à tickets ») a été calculée de manière empirique, en faisant croître cette constante multiplicative jusqu'à la détérioration des performances de l'algorithme. Par ailleurs, chaque site ayant le pouvoir de déterminer si la page de synchronisation est locale ou non, il y a une constante pour les sites qui ne possèdent pas la page (20µs), et une autre constante pour le site qui possède la page (30µs).

Une autre expérience consistait à réaliser l'algorithme du "ticket lock", sans tenir compte de la différence entre les deux compteurs pour le calcul du délai, mais en réalisant un délai croissant exponentiellement à la manière de l'algorithme du "test and set". Nous constatons, que les mesures relevées sont très voisines de celles du verrou à tickets avec délai linéaire, ceci permet de penser que le nombre de transactions générées sur le réseau par les quatre machines, n'est pas assez important pour perturber les acquisitions. En fait je pense que les espacements entre les accès distants pour tester l’état du verrou sont suffisamment longs pour que les autres sites aient le temps d'effectuer leur exclusion mutuelle sans que le délai exponentiel n'ait atteint un très grande valeur. Une fois encore, on ne peut prédire comment évolueraient ces temps avec un nombre plus important de machines.

6.2 Tests des algorithmes de barrières.

Le programme de test des barrières de rendez-vous est semblable à celui des verrous. Nous exécutons un grand nombre de fois un rendez-vous par barrières entre les différents sites, et nous mesurons le temps moyen passé dans la procédure de barrière. La première remarque est que ce temps est identique pour les sites distants et pour les sites possédant la page de synchronisation. Ceci est dû au fait que les processus exécutant ce programme sont complètement synchronisés puisqu'ils passe leur temps à s'attendre.

Pour la barrière à deux compteurs, nous constatons que le temps moyen varie relativement peu avec le nombre de machines participantes. En fait, le temps d'attente réelle est assez faible, car les sites passent plus de temps à réaliser l'incrémentation des deux compteurs, et constatent rapidement que tous les sites sont présents à la barrière. Concernant les barrières centralisées, nous remarquons sans surprise que l'algorithme de la barrière à inversion de sens donne de meilleurs résultat que la barrière à deux compteurs, puisqu'il effectue un nombre moins important d'accès distants.

Mais les résultats les plus prometteurs proviennent des barrières distribuées, car bien que nécessitant une phase d'initialisation relativement coûteuse (environ 500µs), les temps passés dans les barrières sont de 27µs pour la barrière disséminée et de 34µs pour la barrière distribuée simple.

Il est étonnant de constater une différence de temps entre les deux algorithmes, surtout pour deux sites participants, puisqu'ils effectuent exactement le même nombre d'accès distants. J'explique la différence par la façon dont les messages sont envoyés :

- Pour la barrière disséminée, chaque site envoie un message vers l'autre site puis effectue son attente active et, puisque les machines sont synchronisées, les messages émis vont se croiser.
- Dans l'algorithme distribuée simple, un site envoie un message vers le site maître ensuite enverra un message de sortie de la barrière ; les deux émissions de messages sont donc séquentielles dans ce cas.

Le problème des barrières distribuées par rapport aux barrières centralisées, c'est qu'il est nécessaire de connaître qui sont les participants avant l'initialisation de la barrière. On pourrait cependant fournir au programmeur d'application réparties sur SciOS, le choix entre deux types de barrières suivant les besoins de son application.

7. Conclusion

L'objet de ce projet était de réaliser des outils de synchronisation sur un cluster SCI. Après une étude bibliographique des méthodes de synchronisation dans un environnement à mémoire partagée, nous avons sélectionné des algorithmes de verrous et de barrières pouvant être implantés en utilisant les possibilités offertes par la carte SCI (opération atomique *fetch&inc*). Une méthode de verrou est déjà fonctionnelle, le « ticket lock » et la méthode de serveur de messages est encore à l'étude. Pour les barrières, les avantages des barrières distribuées en termes de performances ont clairement été montrés, bien que les barrières centralisées soient plus facilement à manipuler.

Il aurait été intéressant de tester les algorithmes implantés sur un nombre plus important de machines. Deux nouvelles machines sont en cours d'installation à l'heure actuelle, et vont permettre d'effectuer des mesures complémentaires. Cependant, il est possible que le projet SIRAC et le projet APACHE se munissent, dans un future proche, d'un parc de machines reliées par SCI beaucoup plus important (de l'ordre de la centaine de machines). Il est fort probable que certains algorithmes montreraient des performances différentes sur un tel nombre de machines, et il est certain que d'autres algorithmes pourraient être envisagés.

Pour ma part, ce projet m'a permis d'améliorer mes compétences concernant la programmation de bas niveau, puisqu'il s'agissait de programmer une extension du noyau Linux. Par ailleurs j'ai bénéficié des fortes connaissances en systèmes de mes collaborateurs et tuteurs, et j'espère que mon travail aura été à la hauteur de leurs espérances.

8. Bibliographie

[DOLPHIN96]
PCI-SCI cluster adapter specification version 1.2
Dolphin Interconnect Solutions
Mai 1996
http://www.dolphinics.no

[HFM88]
Two Algorithms for Barrier Synchronisation
Debra Hensgen, Raphael Finkel, and Udi Manber
Mars 1988 – International Journal of Parallel Programming, Vol. 7, No. 1

[IEEE92]
IEEE Standard for Scalable Coherent Interface (SCI)
Institute of Electrical and Electronics Engineers
1992 – Standard P1596

[LAMPORT87]
A Fast Mutual Exclusion Algorithm
Leslie Lamport
Février 1987 - ACM Transactions on Computer Systems, Vol. 5, No. 1,
pages 1-11

[MCS91]
Algorithms for Scalable Synchronisation on Shared-Memory Multiprocessors
John M. Mellor-Crummey, Michael L. Scott
Février 1991 - ACM Transactions on Computer Systems, Vol. 9, No. 1,
pages 21-65

[YCTK98]
Designing Tree-Based Barrier Synchronisation on 2D Mesh Networks
Jenq-Shyan Yang and Chung-Ta King
Février 1998 - IEEE Transactions on parallel and distributed systems, Vol. 9, No. 1,
pages 21-65