Conception d'un systeme supportant des modeles de coherence multiples pour les machines paralleles a memoire virtuelle partagee

Alba Cristina Balaniuk

Thèse

présentée par

Alba Cristina Magalhães de MELO BALANIUK

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(arrêté ministériel du 30 Mars 1992)

(Spécialité : **Informatique**)

Conception d'un Système Supportant des Modèles de Cohérence Multiples pour les Machines Parallèles à Mémoire Virtuelle Partagée

---

Date de soutenance : 18 septembre 1996

Composition du jury

Président :	Brigitte	PLATEAU
Rapporteurs :	Claude	TIMSIT
	Thierry	PRIOL
Examinateurs :	Claude	BOKSENBAUM
	Isabelle	DEMEURE
	Traian	MUNTEAN

Thèse préparée au sein du
Laboratoire Logiciels Systèmes et Réseaux – IMAG

A mon fils, Rafael

Remerciements

Je tiens à remercier très sincèrement Mme Brigitte PLATEAU, professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir fait l'honneur de présider mon jury de soutenance. Je la remercie également de m'avoir permis de partager les locaux avec son équipe et d'avoir rendu si agréable ma dernière année de thèse à Grenoble.

Je voudrais temoinger de ma gratitude à M. Thierry PRIOL, professeur à l'Université de Rennes I, et à M. Claude TIMSIT, professeur à l'Université de Versailles Saint-Quentin, d'avoir accepté de rapporter sur mon travail. Je les remercie pour le temps qu'ils m'ont consacré et pour leurs remarques. Je remercie également M. Claude BOKSENBAUM, professeur à l'Université de Montpellier, et Mme Isabelle DEMEURE, maître de conférences à l'Ecole Nationale Supérieure des Telecommunications de Paris, pour l'intérêt qu'ils ont manifesté pour mon travail et pour l'honneur qu'ils me font en acceptant de participer au jury.

Mes remerciements vont aussi à mon directeur de recherches, M. Traian MUNTEAN, professeur à l'Université de la Méditerranée. Son esprit encourageant et son soutien ont été extrêmement importants pour le bon déroulement de mon travail.

Merci infinimment à Léon Mugwaneza d'avoir consacré son temps à vérifier la qualité de rédaction de la thèse et pour ses conseils toujours utiles. Je remercie également Alexandre Carissimi d'avoir aussi consacré son temps à lire ma thèse.

Merci beaucoup au CNPq/Brésil pour son appui financier et pour sa confiance.

Je remercie aussi tous les membres anciens et actuels de l'Equipe SyMPa pour leur aide et leur sympathie: Leïla, Harold, Robert, Ahmed, François, Yves, Nestor, Ibrahima, Pierre et Ghazali. Un merci spécial à Martine Pernice pour sa patience et sympathie.

J'adresse également mes remerciements aux participants du projet APACHE et de l'équipe Calcul Formel du LMC/IMAG. L'ambiance amicale que j'ai trouvée au sein de ces équipes a été très importante dans la phase de rédaction de la thèse. Un merci spécial à Philippe Waille, qui m'a permis d'occuper une partie de son bureau. Merci beaucoup à Denis Trystram, Jean-Marc Vincent, Denis Naddef, Fred, Joële, Kadhija et aux brésiliens Gerson, Ricardo, Paulo, Geyer, Pasin et BenHur.

Un grand merci aux amis que j'ai rencontrés pendant mon séjour à Grenoble et qui m'ont soutenu dans les moments difficiles: Geraldo Cernicchiaro, João Paulo Kitajima, Maria Aparecida Sinohara, Marilena Bittar et Cláudia Linhares. Un merci spécial à Alvaro et Vera Guarda et à Alfredo et Ana Paula Goldman qui m'ont si gentiment accueilli dans leurs maisons. Merci aussi aux brésiliens avec qui j'ai partagé des nombreuses occasions heureuses: Dôdô, Adelina, Ana Paula Jahn, Luis, Denise, Edmar, Glaucia, Elson, Stella, Fabiano, Alessandra, Mari, Isabel, Alexandre, Joao, Jaime, Isabela, Javam, Rosa, Luiz, Carla, Marilia, Celso, Remis.

Je remercie finalement ma famille qui m'a toujours soutenu et encouragé. Mes parents Ivete et Osmar, pour m'avoir montré très tôt l'importance des études et du savoir. Mes soeurs Ana et Adriana pour leurs lettres et "emails". Un grand merci à vóvó Alba, tia Elza et vóvó Laura (in memorium), tia Amália e família, tia Hermínia et tia Leylah pour leur soutien.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Motivation

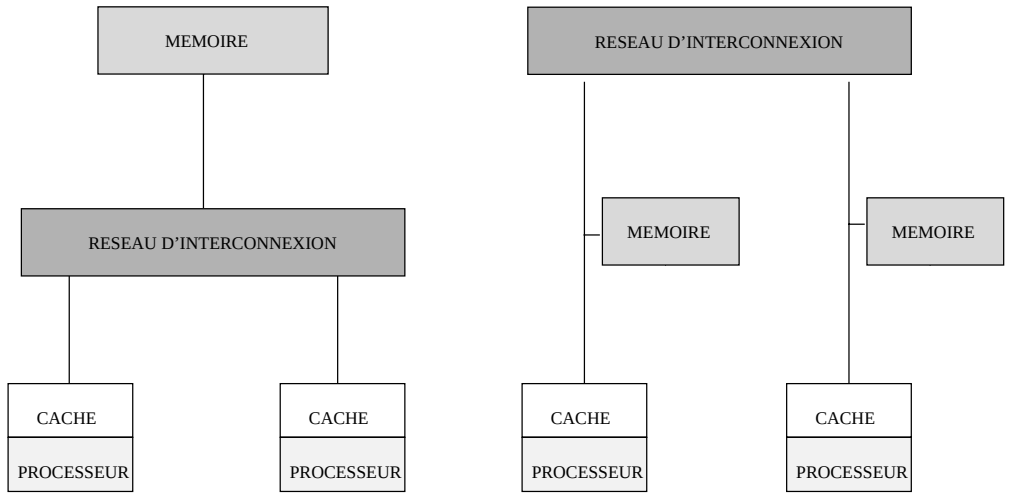
Au cours des deux dernières décennies, nous avons noté une amélioration considérable des performances des processeurs. Ceci a permis que l'utilisation des ordinateurs soit largement disséminée dans la plupart des domaines de la vie moderne: industrie, loisirs, culture, recherche scientifique. Chaque jour, des nouvelles applications apparaissent, toujours gourmandes en puissance de calcul, dans les domaines comme le multimédia, les simulations complexes, le calcul scientifique, la robotique.

Compte tenu des contraintes technologiques et économiques, il apparaît de plus en plus clairement que l'évolution prévisible de la puissance de calcul des monoprocesseurs ne suffira pas à satisfaire les performances requises pour ces applications [HB84]. Dans ce contexte, les machines parallèles ont été conçues avec un double objectif: palier cette demande croissante des performances et adapter au mieux le nombre et le type des processeurs aux besoins des applications, qui sont souvent par nature parallèles.

Les premières machines parallèles dites *fortement couplées* n'étaient qu'une simple extension des monoprocesseurs. Dans les architectures fortement couplées, tous les processeurs accèdent physiquement tous les modules mémoire du système. En ce qui concerne le temps d'accès à la mémoire commune, les machines fortement couplées sont traditionnellement divisées en deux classes: UMA[1] et NUMA[2].

Les architectures UMA ont un temps uniforme d'accès à la mémoire partagée. Les différents processeurs de la machine accèdent en général à la mémoire commune par un bus unique (figure 1.1(a)). Afin de réduire le temps d'accès,

1. UMA - Uniform Memory Access
2. NUMA - Non-Uniform Memory Access



(a) Architectures UMA *(b) Architectures NUMA*

FIG. 1.1 – *Architectures à mémoire commune*

des caches peuvent être rajoutés à l'architecture et placés entre les processeurs et la mémoire. C'est en fait le cas de la plupart des machines UMA existantes. La cohérence entre les différents caches est entièrement assurée par matériel. A cause du bus commun et de la mémoire commune, ce type de machine est peu extensible [EHH92] et le nombre de processeurs supportés arrive à peine à quelques dizaines.

Les machines NUMA ont été conçues pour réduire les problèmes d'extensibilité des architectures parallèles à mémoire commune [LE91]. Comme dans les machines UMA, les processeurs qui composent une machine NUMA peuvent accéder directement à toutes les mémoires du système. En revanche, à la place du bus commun c'est un réseau d'interconnexion qui généralement relie les mémoires aux processeurs (voir figure 1.1(b)). Ceci fait que le coût des accès aux données soit dépendant de la distance entre le processeur et la mémoire où la donnée réside.

Dans ce contexte, les performances d'un algorithme sont directement affectées par le placement des données par rapport aux processeurs qui les accèdent. Toujours dans le but de réduire le temps d'accès aux données, la plupart des architectures NUMA ont recours aux caches. Bien que quelques machines NUMA garantissent la cohérence des caches par matériel, les protocoles employés sont encore très complexes et peu performants [LEH92].

Pour bâtir des systèmes parallèles contenant un nombre important de proces-

seurs, l'approche *faiblement couplée* est généralement adoptée. Les machines résultantes ne possèdent pas de mémoire commune et sont composées d'un ensemble de nœuds connectés par un réseau d'interconnexion. Chaque nœud comprend généralement un processeur et une mémoire privée. Les nœuds communiquent uniquement par échange de messages. La figure 1.2 représente l'architecture faiblement couplée.

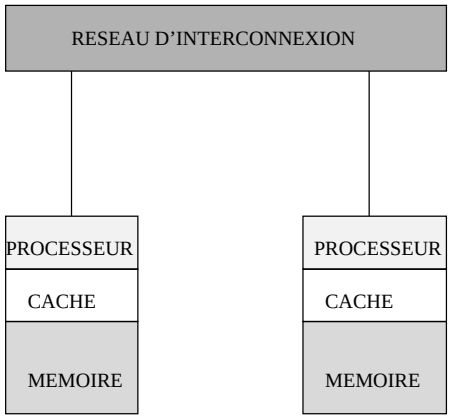


FIG. 1.2 – *Architecture sans mémoire commune*

Par définition, les systèmes massivement parallèles sont des machines faiblement couplées conçues pour être extensibles. Dans une machine extensible, le temps d'exécution des algorithmes qui s'y exécutent est toujours inversement proportionnel au nombre de processeurs [US92]. En d'autres termes, le nombre de processeurs du système ne doit jamais limiter les performances d'un algorithme.

Bien que cette notion soit très simple, la définition formelle de l'extensibilité s'avère une tâche complexe. Ceci est dû à deux aspects fondamentaux. D'abord, l'extensibilité n'est pas un concept global car elle dépend de l'algorithme parallèle en question. Ensuite, il existe plusieurs niveaux d'extensibilité. Une machine extensible doit assurer l'extensibilité au niveau de l'architecture, du système d'exploitation et du langage de programmation.

Selon Nussbaum et Agarwal [NA91], "l'extensibilité d'une architecture mesure la part du parallélisme inhérent à un algorithme qui peut être réalisé sur l'architecture".

Une architecture extensible permet d'exprimer le parallélisme inhérent à un algorithme dans sa totalité. Le temps d'exécution d'un algorithme est limité par ses propres caractéristiques et non par les caractéristiques de l'architecture cible utilisée.

De façon similaire à une architecture extensible, un système d'exploitation

extensible ne doit pas limiter le parallélisme d'une application. La manière de garantir cette propriété est pourtant très différente dans les deux niveaux. Contrairement à l'architecture, nous n'espérons pas une accélération du temps de réponse aux appels système proportionnel au nombre de processeurs. Comme l'addition de processeurs à la machine rend sa gestion plus complexe, l'objectif d'un système d'exploitation extensible est d'assurer les services d'une façon équitable dans un temps fini et borné, indépendant du nombre de processeurs.

Plutôt qu'un mécanisme, l'extensibilité est donc un principe de conception de systèmes parallèles. La conception d'un système extensible exige que les contraintes d'extensibilité soient respectées du plus bas niveau (le matériel) au plus haut niveau (outils de programmation).

En général, le modèle de programmation associé aux machines à mémoire commune est la programmation à variables partagées. Dans ce modèle, un programme est composé de plusieurs flots d'exécution, un pour chaque processus séquentiel [And91]. D'une façon générale, les processus partagent le même espace d'adressage et les interactions entre eux sont réalisées par lecture et écriture des données partagées.

Pour qu'un modèle de programmation à mémoire partagée soit complet, il lui faut des mécanismes de synchronisation qui assurent l'ordonnancement correct des opérations. Le rôle de la synchronisation est de regrouper les opérations d'accès à la mémoire et les exécuter de manière atomique, afin d'empêcher que certains entrelacements indésirables ne se produisent.

Le modèle de programmation traditionnellement associé aux machines sans mémoire commune est fondé sur l'échange de messages, une abstraction naturelle du matériel [NL91]. Dans ce modèle, le processus correspond à un processeur virtuel. Les processeurs physiques communiquent par envoi et réception de messages sur les liens de communication du réseau d'interconnexion. De même, il faut une entité logicielle pour établir la liaison entre les processus. Plusieurs entités ont été définies dans ce but: le canal, la porte, la boîte aux lettres [Les93]. Bien que la sémantique de chacune de ces entités soit différente, la donnée doit toujours être placée sur l'entité de communication et retirée de cette entité pour que sa valeur soit connue. Contrairement aux variables partagées, la communication par échange de messages est donc explicite.

La figure 1.3 illustre la différence entre la programmation par échange de messages et la programmation par mémoire partagée. L'algorithme présenté effectue la multiplication de deux matrices carrées. Dans cet exemple, seules des modifications très simples sont nécessaires pour adapter l'algorithme au modèle de programmation par variables partagées. Il suffit de créer les processus parallèles (forall) et une variable locale par processus qui sert à stocker les résultats intermédiaires. A chaque référence aux variables partagées (matrices a, b et c) le

MULTIPLICATION DE MATRICES

MEMOIRE PARTAGEE

```
int a[N][N], b[N][N], c[N][N];

produit_vecteur(int i,int j)
{
  int somme, k;
  somme = 0;
  for(k=0; k<N; k++)
    somme = somme+a[i][k]*b[k][j];
  c[i,j] = somme;
}

main()
{
  int i, j;

  remplir_matrices(&a,&b);

  for(i=0; i<N; i++)
    for(j=0; j<N; j++)
      creer_parallele(
      produit_vecteur(i,j));
}
```

ECHANGE DE MESSAGES

```
struct message{ int va[N], vb[N] };

produit_vecteur()
{
  int c, ret, i;
  struct message msg_esclave;

  recevoir_message(PERE, &msg_esclave);
  ret = 0;
  for(i=0; i<N; i++)
    ret=ret+msg.esclave.va[i]*
            msg.esclave.vb[i];
  envoyer_msg(pere,ret);
}

main()
{ int a[N][N], b[N][N], c[N][N],
      pid[N][N], i, j;
   struct message msg_maitre;

   remplir_matrices(&a, &b);
   for(i=0; i<N;i++)
    for(j=0; j<N; j++)
     pid[i][j]=creer_parallele(produit_vecteur);
   for(i=0; i<N; i++)
    for(j=0;j<N;j++)
    {
        preparer_msg(&msg_maitre,i,j);
       envoyer_msg(pid[i][j], &msg_maitre);
    }
    for(i=0; i<N; i++)
     for(j=0; j<N; j++)
       recevoir_msg(pid[i][j], &c[i][j]);
  }
```

FIG. 1.3 – *Multiplication de Matrices*

mouvement des données entre les processus s'effectue de façon transparente.

En revanche, plusieurs modifications sont nécessaires dans le cas de l'échange de messages. La communication n'est plus transparente à l'utilisateur. Celui-ci doit maintenant se servir des primitives d'envoi et réception de messages pour passer les données entre processus. Cet exemple illustre l'organisation maître/esclave où un processus parallèle (maître) crée tous les autres processus (esclaves) et leur envoie des messages avec les données nécessaires au calcul. Les valeurs obtenues par chaque processus sont renvoyées au processus maître à la fin du calcul.

Du point de vue du programmeur, le modèle de programmation par variables partagées est donc en général plus simple que le modèle de programmation par

échange de messages.

Le portage du modèle de programmation à mémoire partagée aux architectures UMA est simple. Il n'est pas nécessaire de simuler une vue globale et unique des données puisqu'elle existe au niveau le plus bas grâce à l'unicité et à l'accessibilité de la mémoire physique. Avec un protocole correct de cohérence de caches, le modèle peut être implanté comme dans les machines monoprocesseur. Néanmoins, quelques optimisations internes au processeur telles que tampons d'écriture doivent être exclues car son utilisation peut conduire à des résultats inattendus [Lam79].

Etant donnée que les mémoires de l'architecture NUMA sont accessibles par tous les processeurs directement par matériel, la programmation à mémoire partagée peut être implantée exactement comme dans les machines monoprocesseur. Cependant, une implantation de ce type est très peu performante. La plupart des implantations de la mémoire partagée adaptés à ce type de machine essayent de prendre en compte la distance entre le processeur et la mémoire.

Les utilisateurs des machines parallèles sans mémoire commune souhaitent naturellement, souvent pour des raisons de portage des applications, utiliser le modèle de programmation à mémoire partagée, bien que la mémoire de ces machines ne soit pas physiquement partagée. Afin de créer l'illusion d'une mémoire accessible par n'importe quel processeur, une couche intermédiaire entre le matériel et le programmeur se fait nécessaire. Cette couche se sert du concept de *mémoire distribuée partagée*.

La mémoire distribuée partagée a été conçue pour permettre aux utilisateurs d'une machine faiblement couplée de profiter du modèle de programmation à données partagées. Au plus bas niveau, les mécanismes qui simulent la mémoire partagée communiquent par échange de messages. L'utilisateur, en revanche, a l'illusion de manipuler une mémoire globale accessible directement par tous les processeurs du système. En plus d'offrir aux utilisateurs d'une machine faiblement couplée un modèle de programmation plus simple, la mémoire distribuée partagée est le premier pas vers un standard de programmation parallèle, où les particularités architecturales de chaque machine ne sont pas prises en compte. En effet, l'utilisation de la mémoire distribuée partagée fait que le portage des programmes parallèles entre architectures distinctes devient une tâche simple.

Il existe deux approches pour implanter la mémoire distribuée partagée [LKBT92]: les objets partagés et la mémoire virtuelle partagée (MVP).

Dans la première approche, les données partagées sont projetées sur un ensemble d'objets. Ces objets sont accédés par des opérations de haut niveau définies par l'utilisateur [BKT92]. Son implantation est en général assurée par les langages de programmation. Bien que cette approche rende la programmation très structurée et fiable, les performances atteintes ne sont pas les meilleures. De

plus, le modèle de programmation devient plus complexe à cause du rajout des primitives d'accès aux données.

La mémoire virtuelle partagée a été initialement définie par K. Li dans [Li86]. Dans cette approche, les données partagées constituent un espace d'adressage global et unique. De façon similaire à la mémoire virtuelle, cet espace d'adressage est organisé en pages et accédé par des primitives de bas niveau de lecture et écriture en mémoire (LOAD et STORE). Ceci fait que la mémoire virtuelle partagée est généralement implantée par le matériel ou par le système d'exploitation. Ce type d'implantation permet une amélioration sensible des performances par rapport aux objets partagés. De plus, le modèle de programmation résultant est très proche de celui utilisé dans les machines monoprocesseur.

La mémoire virtuelle partagée est une façon transparente et élégante d'implanter la mémoire distribuée partagée. L'activité de recherche menée dans le domaine de la mémoire virtuelle partagée est très importante et suit plusieurs axes.

Une grande partie des efforts ont été faits dans le but de concilier le modèle de programmation à variables partagées et les performances des machines parallèles sans mémoire commune. Nous pouvons distinguer deux phases distinctes dans la recherche dans ce domaine.

La première phase est "la découverte de la mémoire virtuelle partagée". Cette phase a pour objectif de montrer que la programmation dans un espace d'adressage partagé peut être accomplie même dans un environnement où la mémoire physiquement partagée n'existe pas. Plusieurs systèmes à mémoire virtuelle partagée, tels que Ivy [Li88], Mirage [FP89], KOAN [LP92] et Platinum [CF89], ont été conçus dans cette étape. Une étude plus approfondie de ces systèmes a pourtant amené à une constatation frappante: le maintien de l'abstraction parfaite d'une mémoire commune était très coûteux en temps d'exécution. Les pertes très importantes en performances ont conduit à la remise en cause du modèle de cohérence de la mémoire utilisé jusqu'alors. Ce modèle, appelé cohérence forte, garantit que tous les processeurs perçoivent toujours le même ordre des accès aux données partagées.

Dans ce rapport, nous utilisons "cohérence de la mémoire" dans le sens des termes anglais "memory consistency". Le terme cohérence est aussi couramment utilisé dans la littérature quand il faut garantir que toutes les copies d'une même donnée ont la même valeur. Dans ce cas, nous utilisons "cohérence du cache". Dans le texte qui suit, les termes "cohérence de la mémoire" et "cohérence du cache" traitent donc de problèmes différents et sont utilisés de façon non ambiguë.

L'abandon du concept abstrait d'une mémoire commune similaire à la mémoire physique des machines monoprocesseur a marqué le début de la seconde phase. Afin d'approcher des performances acceptables, plusieurs systèmes à mé-

moire virtuelle partagée ont relâché certaines conditions de cohérence de la mémoire partagée. Les modèles de la mémoire résultants sont dits de cohérence relâchée et tous laissent entrevoir à l'utilisateur la nature distribuée de la mémoire virtuelle partagée. En permettant une plus grande concurrence dans les opérations d'accès à la mémoire, ces modèles offrent la possibilité d'atteindre des performances plus élévées que celles des modèles à cohérence forte. Le prix à payer est l'augmentation de la complexité du modèle de programmation.

L'absence d'une mémoire commune n'est plus cachée à l'utilisateur. Celui-ci, devant raisonner avec la nature distribuée de la mémoire, manipule des modèles de cohérence beaucoup plus complexes que ceux offerts à l'utilisateur d'une machine monoprocesseur. Plusieurs systèmes à mémoire virtuelle partagée, tels que Munin [Car93], Midway [BZS93a] et ThreadMarks [KDCZ93], ont été conçus dans cette phase.

La multitude de modèles de cohérence de la mémoire et les analyses des performances correspondantes semblent montrer qu'il n'existe pas de modèle de cohérence de la mémoire qui présente un bon compromis entre les performances et la simplicité de programmation pour une large gamme d'applications. Ainsi, la tendance des recherches d'aujourd'hui est de lier le choix du modèle de cohérence de la mémoire aux caractéristiques d'accès aux données inhérentes aux applications. Les systèmes qui permettent ce choix sont dits systèmes à modèles de cohérence multiples. Les recherches dans ce sens n'en sont qu'à leur début.

Un modèle de programmation à variables partagées doit offrir des mécanismes de synchronisation. Autre que la définition de nouvelles primitives de synchronisation, la recherche dans ce domaine consiste en grande partie à trouver des manières d'implanter les mécanismes de synchronisation existants de façon efficace. Ces mécanismes présentent en général une très grande contention, qui s'aggrave sur les machines parallèles. Une implantation efficace de la synchronisation entre processus reste donc aussi une question ouverte.

Un autre axe de recherche important est celui de la gestion de la mémoire virtuelle dans un environnement parallèle. Il comprend la localisation et le chargement des pages aussi bien que leur remplacement et leur placement sur les nœuds. Une étude très complète à propos de la localisation des pages a été menée par Li et Hudak dans [LH89]. Presque tous les systèmes à mémoire virtuelle partagée postérieurs à cette étude implantent des techniques qui y sont proposées. Le problème du remplacement des pages a été étudié par Lahjomri et Priol dans [LP92]. Une solution a été proposée qui place la page a supprimer dans les nœuds distants.

L'étude du chargement des pages remet en discussion la technique du préchargement, qui a été extensivement étudiée pour le cas des monoprocesseurs, sans pour autant présenter de bons résultats. Son adéquation aux environnements

parallèles est encore une question ouverte et quelques travaux ont été ménés dans ce sens [LE91].

L'étude du remplacement de pages sur une machine monoprocesseur a été menée d'une manière extensive. Les résultats indiquent qu'une politique LRU globale présente les meilleurs résultats [Tan92]. Pour ces machines, résoudre le problème du remplacement des pages se résume à choisir la meilleure page à remplacer. Sur les machines parallèles, outre le problème du choix de la page à remplacer, nous devons aussi décider du site vers lequel la page choisie doit migrer. La nécessité de ce choix rajoute une nouvelle dimension au problème du remplacement des pages. Le choix du site de migration est un problème très complexe pour lequel il n'existe pas de solution optimale.

D'autres questions telles que la taille de la page [Hol89], l'analyse des références aux pages [LEH92], le faux partage [KJe91], l'annotation des données [HSMB91] sont aussi étudiées dans le domaine de la mémoire virtuelle partagée. D'autres domaines importants de recherche sur la mémoire virtuelle partagée, tels que l'allocation dynamique de la mémoire [JJ92], l'hétérogénéité [ZSLW92], la tolérance aux pannes [CMP95] et la protection de la mémoire [HERV93], feront eux-mêmes des sujets de thèse à part.

1.2 Contributions

Cette thèse porte sur la conception des mécanismes permettant à offrir le support nécéssaire à la programmation à mémoire partagée avec des modèles de cohérence multiples. La conception de ces mécanismes a été faite en observant des critères d'extensibilité pour les adapter aux architectures massivement parallèles.

Dans notre approche, nous considérons que l'application doit avoir le choix des mécanismes les plus adaptés à ses besoins. Ce choix doit être offert à plusieurs niveaux, tels que la communication, la gestion des processus et, notamment, la gestion de la mémoire.

Nous nous plaçons alors dans le cadre des systèmes réconfigurables, qui offrent aux applications le choix entre différents supports d'exécution à partir de la construction générique de mécanismes. Nous suivons la méthodologie de conception adoptée dans le micro-noyau ParX [CMW93] [Lan91] [M+89], dévéloppé au sein de notre équipe.

L'objectif principal de notre système, appelé $\mathcal{DIVA}$[3] , est d'offrir un modèle de programmation simple capable d'atteindre des hautes performances. Pour y arriver, nous traitons les deux grandes questions associées à la conception d'un

3. $\mathcal{DI}$stributed $\mathcal{V}$irtual memory $\mathcal{A}$pproach

système à mémoire virtuelle partagée: la gestion de la mémoire partagée et la gestion de la mémoire virtuelle.

Dans le domaine de la gestion du partage de la mémoire, nous nous concentrons sur la conception des mécanismes de base sur lesquels plusieurs modèles de cohérence de la mémoire peuvent être bâtis. Nous croyons que le bon compromis entre la simplicité de programmation et les hautes performances est trouvé lorsqu'il est possible pour une application de manifester ses propres besoins de cohérence.

Dans le domaine de la gestion de la mémoire virtuelle, nous proposons des mécanismes pour réduire le surcoût apporté par la gestion distribuée des pages sur le temps d'exécution de l'application. Les mécanismes proposés prennent toujours en compte la coexistence entre différents modèles de cohérence.

Les contributions apportées par cette étude sont:

- *Modèles multiples de cohérence de la mémoire.* En offrant plusieurs modèles de cohérence de la mémoire, $\mathcal{DIVA}$ laisse à l'utilisateur le choix du support de cohérence le plus adapté aux besoins de son application parallèle. En plus, $\mathcal{DIVA}$ permet la définition par l'utilisateur de ses propres modèles de cohérence.

  Pour qu'un modèle de programmation à mémoire partagée soit complet, il doit intégrer des primitives de synchronisation. La sémantique de ces primitives dépend du modèle de cohérence utilisé. Bien que plusieurs modèles de cohérence de la mémoire soient admis par $\mathcal{DIVA}$, la syntaxe des primitives de synchronisation est unique. C'est à $\mathcal{DIVA}$ d'appliquer la sémantique correspondant au modèle courant et ceci est fait de façon transparente.

- *Optimisation des mécanismes de gestion de pages.* Nous proposons deux mécanismes pour augmenter les performances de la gestion des pages dans un système à mémoire virtuelle partagée. Ces mécanismes traitent du remplacement et du préchargement des pages, respectivement.

  Contrairement aux systèmes à mémoire virtuelle partagée traditionnels, où les données à remplacer sont toujours envoyées au disque, $\mathcal{DIVA}$ essaye de les envoyer aux mémoires voisines. Cette politique permet de réduire le temps nécessaire pour stocker les données ainsi que le temps de leur chargement en mémoire, si jamais elles y sont référencées de nouveau.

  Aussi, $\mathcal{DIVA}$ offre à l'utilisateur des mécanismes qui permettent le chargement d'une donnée en mémoire avant que l'accès à celle-ci ne soit généré. Les données préchargées sont stockées dans des tampons système. Les opérations de cohérence y sont aussi appliquées.

1.3 Organisation du rapport

L'organisation de la suite de ce rapport est la suivante. Dans le chapitre 2, nous présentons les principaux problèmes posés dans la conception d'un système à mémoire virtuelle partagée. Nous présentons d'abord la problématique associée à la gestion du partage. Dans ce domaine, nous nous concentrons sur la définition des modèles de cohérence de la mémoire. Nous y présentons aussi quelques modèles de cohérence existants dans la littérature. Ensuite, nous décrivons les problèmes associés à la gestion de la mémoire virtuelle dans un environnement parallèle. A la fin de ce chapitre, nous présentons quelques exemples de systèmes à mémoire virtuelle partagée.

Le chapitre 3 présente les mécanismes proposés dans $\mathcal{DIVA}$ pour le support aux modèles de cohérence multiples. Nous y décrivons la conception du module qui permet le traitement d'un modèle de cohérence générique et nous présentons l'interface qui permet l'ajout de nouveaux modèles de cohérence à $\mathcal{DIVA}$. A la fin du chapitre, nous décrivons l'approche que nous avons adopté pour gérér la synchronisation.

Les mécanismes proposés dans $\mathcal{DIVA}$ pour traiter la gestion de la dynamique des pages dans un environnement multi-modèles sont présentés dans le chapitre 4. Nous y traitons notamment le problème du remplacement et du préchargement de pages.

La mise en œuvre d'un prototype de $\mathcal{DIVA}$ qui implante les mécanismes proposés dans les deux chapitres précédents est présentée dans le chapitre 5.

Le chapitre 6 présente quelques mesures de performances et fait l'analyse des fonctionnalités offertes par notre système. Enfin, le chapitre 7 fait le bilan et trace quelques perspectives futures de ce travail.

Chapitre 2

La mémoire virtuelle partagée

La mémoire virtuelle partagée étant un domaine de recherche très vaste et diversifié, il nous est impossible de citer ici tous les travaux de ce domaine. Dans ce chapitre, nous nous limitons aux travaux concernant les problèmes traités dans la suite de cette thèse.

En général, deux grandes questions se posent lors de la conception d'un système à mémoire virtuelle partagée: la gestion du partage de la mémoire et la gestion des pages.

La gestion du partage traite de la définition du comportement des opérations d'accès à la mémoire partagée dans un environnement parallèle. Dans ce contexte, nous présentons le support nécéssaire au partage de la mémoire, les différents modèles de cohérence de la mémoire et les mécanismes couramment utilisés pour la synchronisation entre les processus parallèles.

La gestion des pages concerne les problèmes posés par la gestion de la mémoire virtuelle dans les machines parallèles, tels que la localisation des pages, aussi bien que leur placement, chargement et remplacement. Nous y présentons aussi le problème du faux partage.

A la fin du chapitre, nous présentons quelques systèmes à mémoire virtuelle partagée qui ont été proposés dans la littérature.

2.1 Support au partage de la mémoire

Dans ce paragraphe, nous présentons les mécanismes de base qui sont utilisés pour assurer le partage de la mémoire dans les systèmes à mémoire virtuelle partagée.

2.1.1 Niveau de partage des données

La première décision à prendre lorsqu'on envisage le partage des données concerne la définition des données à partager. D'une façon générale, le partage peut être réalisé au niveau du segment, du bloc ou des variables.

Partage de segments

Au niveau le plus général, nous pouvons décider que toutes les données d'un processus sont accessibles à tous les autres processus parallèles (voir figure 2.1). Dans ce cas, le segment de données de chaque processus est projeté sur un même segment de mémoire global et aucune restriction d'accès n'est appliquée.

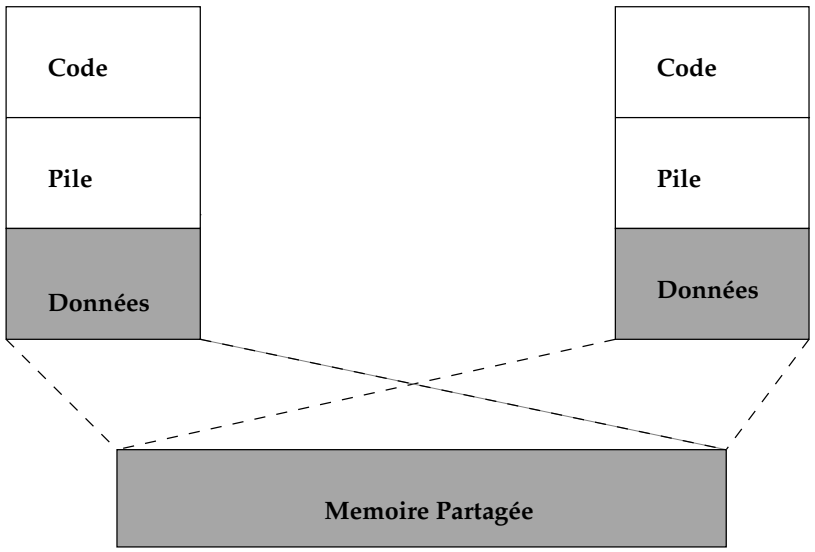


FIG. 2.1 – *Partage de segments*

Bien que cette approche rende la tâche de définition des données à partager transparente à l'utilisateur, elle entraîne plusieurs inconvénients. D'abord, les mécanismes de gestion de la mémoire partagée sont utilisés pour toutes les données du programme, même si la plupart des accès ne sont effectués que localement. De plus, le type de partage ne peut pas être spécifié. Ainsi, d'une façon conservative, toutes les données sont considérées comme partagées en écriture. Ivy [LH89] est un exemple de système à mémoire virtuelle partagée qui adopte le partage de segments.

Partage de Blocs

Dans cette approche, l'utilisateur précise un ensemble de données à partager (bloc). Le type de partage, la taille et les restrictions d'accès au bloc sont spécifiés. Par rapport à l'approche précédente, l'utilisateur dispose de deux primitives additionnelles, une pour demander la projection du bloc sur un segment partagé (*mapping*) et l'autre pour retirer la projection (*unmapping*).

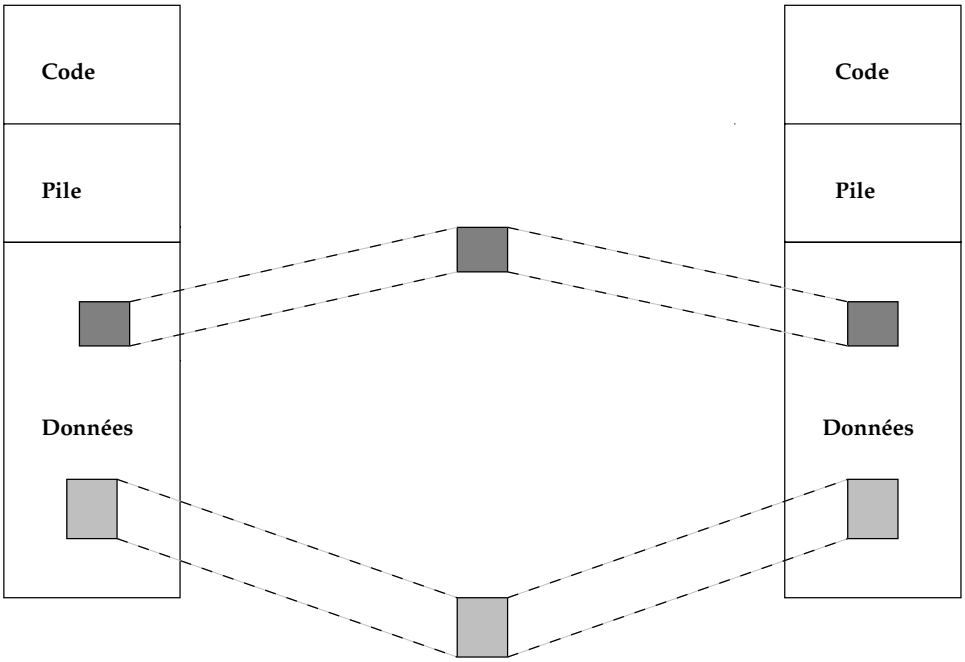


FIG. 2.2 – *Partage de blocs*

Il existe maintenant plusieurs espaces de partage (voir figure 2.2), un pour chaque bloc global. Ceci rend la tâche de protection des données plus simple puisque les seuls processus qui ont accès aux blocs sont ceux qui les ont projetés sur leur espace d'adressage. Le seul inconvénient du partage de blocs est une relative perte de transparence causée par l'addition des deux primitives `map` et `unmap`. L'accès aux données est toujours réalisé de manière transparente. Les systèmes Mirage [FP89] et Koan [LP92] adoptent cette approche.

Partage de variables

Avec cette approche, le programmeur annote chaque variable du programme comme "partagée" ou "locale". De façon générale, le type "partagé" est rajouté

à l'ensemble de types de données admis par le langage. Dans certains cas, le type de partage (lecture, écriture) peut être précisé.

Le partage au niveau des variables peut générer des systèmes à mémoire virtuelle partagée très performants puisque différents protocoles peuvent être mis en œuvre au niveau de la variable et selon le type de partage. Pourtant, les inconvénients sont nombreux. D'abord, l'addition d'un nouveau type de données au langage parallèle apporte des modifications au compilateur. De plus, la transparence est perdue dans sa totalité. Le modèle de programmation se complique car la tâche d'annotation des données y est rajoutée. Les systèmes Munin [Car93] et Midway [BZ91a] ont adopté le partage des variables.

2.1.2 Politique de placement des données partagées

Dans un système à mémoire virtuelle partagée, les données partagées sont accédées par plusieurs nœuds au cours de l'exécution d'un programme parallèle. Puisqu'il n'existe pas de mémoire commune, la stratégie de placement détermine où placer la donnée lorsqu'un accès à celle-ci est généré. Trois solutions peuvent être envisagées pour ce problème: le placement fixe, la migration ou la duplication sur les nœuds.

Avec la première approche, la donnée à partager est placée sur un site gestionnaire et les nœuds exécutent des opérations à distance pour la consulter ou la modifier. Bien que son implantation soit très simple, cette stratégie ne donne pas de bonnes performances. En effet, le site gestionnaire responsable de la donnée devient vite surchargé lorsque plusieurs sollicitations sont effectuées simultanément. De plus, le coût de l'accès aux données partagées est toujours égal au coût d'un accès à distance, sauf dans le cas particulier où le site gestionnaire référence lui-même la donnée.

Pour profiter du Principe de la Localité, les techniques de migration et duplication sur plusieurs nœuds sont employées. La migration place la donnée toujours sur le nœud qui l'a utilisée le dernier. Les accès suivants à cette même donnée par le même processeur se déroulent donc localement.

Dans la duplication sur plusieurs nœuds, plusieurs copies d'une même donnée sont générées, une pour chaque nœud qui l'accède. La duplication des données sur plusieurs sites peut conduire à des problèmes de cohérence du cache. Lorsqu'une copie est modifiée, la mise à jour des autres copies doit se faire. La duplication amortit le coût des lectures. La première lecture d'une donnée se traduit toujours par un accès à distance. En revanche, les lectures suivantes ne sont effectuées que localement. Toutefois, le coût de l'écriture est augmenté puisqu'un protocole de maintien de cohérence entre les copies devient nécessaire. Puisqu'en général le nombre de lectures effectuées par un programme est plus important que le

nombre d'écritures, la grande majorité des systèmes à mémoire virtuelle partagée emploient des techniques de duplication de données. C'est le cas de Mirage [FP89], Clouds [Moh93], Munin [BCZ91] et Midway [BZ91a].

2.1.3 Protocoles de cohérence du cache

Un protocole de cohérence du cache est utilisé quand il faut garantir que toutes les copies d'une même donnée possèdent la même valeur.

Le problème qui se pose ressemble beaucoup au problème de cohérence entre caches physiques, de sorte que les mêmes protocoles peuvent être utilisés sur la mémoire virtuelle partagée. Nous présentons les deux protocoles traditionnels de cohérence du cache - l'invalidation et la mise à jour - ainsi que les protocoles adaptatifs.

Invalidation

Un protocole d'invalidation invalide toutes les copies d'une donnée à chaque accès en écriture. La cohérence du cache est assurée car il n'existe plus de copies multiples d'une même donnée. Cette approche engendre un trafic d'information peu important sur le réseau d'interconnexion puisque seuls des messages de contrôle sont envoyés. Néanmoins, elle présente un surcoût si jamais la donnée invalidée sur un processeur doit être à nouveau utilisée par ce processeur.

Les stratégies d'invalidation sont classées en auto-invalidation et invalidation dirigée. Dans la première approche, le compilateur rajoute au programme des instructions qui forcent l'invalidation des données. Dans la deuxième approche, au contraire, l'invalidation est décidée par une entité externe au programme (en général, le système d'exploitation ou le matériel).

Mise à jour

Dans une approche de mise à jour, il existe toujours plusieurs copies d'une même donnée dans le système. Dès que la cohérence doit être garantie, la nouvelle valeur de la donnée est diffusée vers toutes les copies.

L'avantage de cette approche est que les prochains accès de lecture à cette donnée se dérouleront sans aucun délai. En revanche, le trafic du réseau est augmenté à cause du grand nombre de messages de mise à jour si un processeur écrit de nombreuses fois la même donnée.

Protocoles adaptatifs

Les performances des protocoles de cohérence dépendent aussi des caractéristiques de partage de la donnée. Par exemple, un programme où les données sont écrites par un seul processeur et lues simultanément par plusieurs atteint des meilleures performances sur un protocole de mise à jour. En revanche, dès que les données sont écrites plusieurs fois par un même processeur, le protocole d'invalidation présente des meilleures performances [Lil93].

Les protocoles adaptatifs essayent d'ajuster le protocole de cohérence aux caractéristiques de partage de l'application.

En général, les protocoles adaptatifs font d'abord la mise à jour des données. Si quelques copies ne sont pas référencées dans un délai donné, elles sont invalidées. Un cas extrême de protocole adaptatif est celui de Munin [Car93], dans lequel l'utilisateur fait des annotations à propos des caractéristiques de partage de chaque variable et le protocole est choisi selon ces annotations.

2.2 Modèles de cohérence de la mémoire

Dans un environnement parallèle, la cohérence du cache seule n'est pas suffisante pour assurer l'exécution correcte des programmes. Il faut aussi établir des règles concernant l'ordre dans lequel plusieurs données sont utilisées. Cet ensemble de règles est défini par un modèle de la mémoire ou modèle de cohérence de la mémoire partagée. Le choix du modèle de cohérence de la mémoire est une question fondamentale pour la conception des mécanismes de gestion du partage.

2.2.1 Introduction

D'une façon intuitive, les instructions qui composent un programme sont effectuées les unes après les autres, dans l'ordre specifié par le programme et les opérations d'accès à la mémoire sont effectuées de manière atomique. Ceci correspond donc au modèle traditionnel que le programmeur possède de la mémoire et, pour que son programme produise des résultats corrects, la mémoire doit se comporter comme telle. Un modèle de cohérence de la mémoire (MCM) formalise le concept décrit ci-dessus et définit l'ordre des opérations d'accès à la mémoire perçu par le programmeur. Il s'agit ici de l'ordre apparent d'éxecution des opérations mémoire plutôt que de son ordre réel.

Toujours en respectant ce modèle de cohérence intuitif, un grand nombre

d'optimisations ont été réalisées pour augmenter les performances des monoprocesseurs. Parmi ces optimisations, nous trouvons notamment les écritures non-atomiques (via tampons d'écriture) et le réordonnancement des instructions [Adv93]. Ceci est possible car les changements apportés ne modifient pas l'ordre perçu par le programmeur et les opérations paraissent être effectuées selon l'ordre défini par le programme.

Deux types fondamentaux d'optimisations existent: optimisations par matériel et optimisations par le compilateur. Les premières permettent qu'un accès à la mémoire soit effectué avant la fin des accès précédents. Les dernières réordonnent les instructions qui accèdent à la mémoire.

Hélas, il est impossible d'appliquer les mêmes optimisations au modèle de cohérence de la mémoire intuitif quand il s'agit des architectures parallèles. L'existence de plusieurs caches et mémoires privées rend le système mémoire beaucoup plus complexe.

Les basses performances atteintes par les systèmes qui simulent la mémoire unique sur les architectures parallèles ont mis en cause le modèle abstrait de la mémoire utilisé jusqu'alors. Plusieurs chercheurs ont proposé des modèles plus relâchés. Bien que ces modèles permettent potentiellement d'atteindre des performances élevées, la programmation des applications qui les utilisent devient plus complexe. Le programmeur doit dorénavant raisonner avec la multitude de copies et la nature distribuée de la mémoire.

Le modèle de cohérence de la mémoire joue un rôle essentiel dans deux caractéristiques d'un système: la simplicité de programmation et les performances. La simplicité de programmation dépend du modèle de cohérence car les programmeurs doivent l'utiliser pour raisonner à propos des résultats produits par leur programme. Les performances y sont aussi affectées puisque c'est le modèle de cohérence de la mémoire qui détermine trois aspects essentiels: les opérations d'accès à la mémoire qui peuvent être effectuées simultanément, le moment où les valeurs écrites deviennent visibles aux autres processeurs du système et le surcoût de communication apporté par une opération d'accès à la mémoire.

D'une façon générale, le modèle de programmation devient plus compliqué quand les conditions de cohérence de la mémoire s'afffaiblissent. En revanche, les performances atteintes par les modèles de cohérence résultants sont nettement plus hautes.

Les modèles de cohérence de la mémoire imposent des restrictions d'accès basées sur un nombre d'attributs: l'adresse mémoire accédée, le type de l'accès (lecture, écriture), la valeur transmise et la catégorie de l'accès. La figure 2.3 présente les catégories d'accès les plus répandues.

Les modèles de cohérence de la mémoire ne s'appliquent qu'aux données parta-

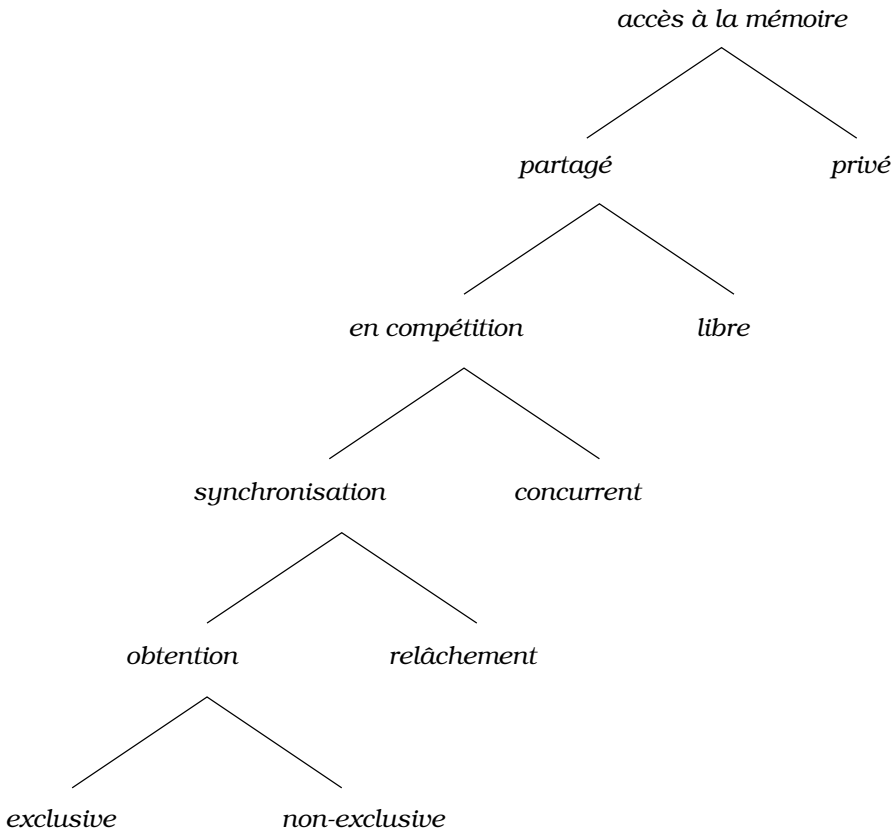


FIG. 2.3 – *Catégories d'accès à la mémoire*

gées par plusieurs processeurs. C'est le fait de créer plusieurs copies de la mémoire qui pose les problèmes de cohérence.

Les accès à la mémoire partagée sont classés comme *en compétition* ou *libres*. Deux accès sont en compétition s'ils accèdent de façon non ordonnée à la même position mémoire et qu'au moins un des accès est en écriture [Mos93].

Les accès en compétition sont à leur tour classés comme accès de *synchronisation* ou accès *concurrents*. Les accès de synchronisation sont utilisés pour établir un ordre des accès à la mémoire partagée. Ils servent à obtenir un objet de synchronisation (*accès obtention*) ou à le relâcher (*accès relâchement*). L'obtention d'un objet de synchronisation est vue comme une synchronisation en lecture et l'acte de relâcher un objet de synchronisation est un accès de synchronisation en écriture.

L'obtention d'objets de synchronisation peut être faite de manière *exclusive* ou *non exclusive*. Plusieurs obtentions non exclusives peuvent être réalisées simultanément tandis que l'obtention exclusive d'un objet de synchronisation n'est

accordée qu'à un processus à la fois.

Les modèles de cohérence de la mémoire qui emploient différentes restrictions d'ordre selon la catégorie de l'accès sont dits *hybrides* tandis que ceux qui traitent de la même manière toutes les catégories d'accès sont dits *uniformes*.

Dans la littérature, nous trouvons une grande variété de modèles de cohérence de la mémoire. La plupart de ces modèles a été définie soit par des formalismes non standard soit par des définitions qui prennent en compte des caractéristiques particulières du matériel. Ceci rend difficile la comparaison.

Quelques chercheurs tels que Raynal et Schiper dans [RS95] et Adve dans [Adv93] ont proposé des formalismes pour la définition des modèles de cohérence de la mémoire. Leur but principal est d'étudier le comportement de chaque modèle et de les comparer.

Dans la suite de ce chapitre, nous allons présenter les modèles de cohérence de la mémoire les plus répandus dans la littérature. D'abord, nous définissons les entités et les relations qui font partie de l'exécution d'un programme parallèle. La description de chaque modèle est faite par la suite et suit le schéma suivant: d'abord, le modèle est présenté tel qu'il a été originellement défini. Ensuite, une définition formelle du modèle est présentée et à la fin nous citons quelques implantations de systèmes à mémoire virtuelle partagée qui se servent du modèle.

2.2.2 Définitions et Notations

Afin de raisonner à propos des modèles de cohérence de la mémoire, nous avons besoin d'abord de définir les entités qui participent à l'exécution d'un programme parallèle. Les définitions présentées ici se sont inspirées des travaux de Adve dans [Adv93], Kohli et al. dans [KNA93], Heddaya et Sinha dans [HS93] et Raynal et Schiper dans [RS95]. Quelques concepts présentés dans ses travaux et notamment la définition d'un modèle de cohérence de la mémoire ont été modifiés pour mieux servir nos objectifs.

Exécution d'un programme parallèle

Un *programme parallèle* est exécuté par un système.

Un *système* est un ensemble fini de processeurs.

Chaque *processeur* p_i exécute un programme sequentiel.

Un *programme séquentiel* s'exécutant sur le processeur p_i est une séquence d'opérations $o_{p_i}(x)v$ sur la mémoire globale partagée.

La *mémoire globale partagée* M est une entité abstraite composée d'un ensemble de mémoires locales m_i (voir figure 2.4).

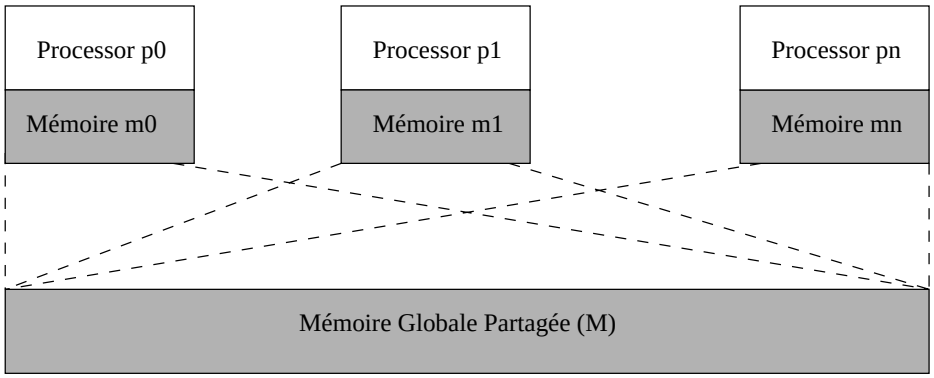


FIG. 2.4 – *Modèle du système mémoire*

Opérations sur la mémoire

Une *opération sur la mémoire partagée* $(o_{p_i}(x)v)$ est effectuée par un processeur p_i sur l'adresse x avec la valeur v. Il existe deux types de base d'opérations: la lecture (r) et l'écriture (w).

Une *opération d'accès à la mémoire est lancée* $(ini(o_{p_i}(x)v))$ quand le processeur exécute l'instruction correspondant à l'accès.

Une *lecture* $r_{p_i}(x)v$ *est terminée* $(fin(r_{p_i}(x)v))$ quand une écriture sur la même adresse x n'est plus capable de modifier la valeur v lue par p_i.

Une *écriture* $w_{p_i}(x)v$ *est terminée par rapport au processeur* p_j $(fin_j(w_{p_i}(x)v))$ quand la valeur v est attribuée à l'adresse x sur la mémoire m_j de p_j.

Une *écriture* $w_{p_i}(x)v$ *est terminée* $(fin(w_{p_i}(x)v))$ quand elle est terminée par rapport à tous les processeurs qui partagent x. Plusieurs accès peuvent se dérouler simultanément.

Un processeur lance une opération d'accès à la mémoire par une invocation. Ensuite, il attend jusqu'à ce que la mémoire envoie une réponse à cette invocation. Ayant reçu la réponse, le processeur peut lancer la prochaine opération d'accès à la mémoire. *L'intervalle d'une opération* est l'intervalle de temps entre le lancement de l'opération $(ini(o_{p_i}(x)v))$ et la réception de la réponse par le processeur qui l'a lancée $(rep(o_{p_i}(x)v))$.

A la fin de l'exécution d'un programme parallèle, toutes les opérations d'accès à la mémoire doivent être terminées.

Historiques d'exécution

Un *Historique Local d'Exécution d'un processeur* p_i *(*H_{p_i}*)* est une séquence ordonnée d'opérations de lecture et écriture exécutées par le processeur p_i. Dans la suite, nous utiliserons la représentation graphique montrée dans la figure 2.5 pour répresenter un historique local d'exécution.

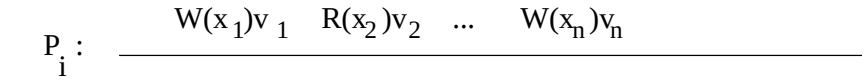


FIG. 2.5 – *Historique Local d'Exécution*

Le temps croit de la gauche vers la droite. Ainsi, $W(x_1)v_1 \xrightarrow{H_{p_i}} R(x_2)v_2$. Dans un historique d'exécution H_{p_i}, la notation $W(x)v$ représente le moment où l'écriture est lancée ($ini(w_{p_i}(x)v)$). De même, la notation $R(x)v$ représente le moment où la lecture est terminée ($fin(r_{p_i}(x)v)$).

Un *Historique d'Exécution (H)* est un ensemble d'historiques locaux d'exécution, un pour chaque processeur.

Si Q est un historique, une *séquence linéaire* de Q contient toutes les opérations de Q exactement une fois. Une séquence linéaire est *légale* si toute opération de lecture $r_{p_i}(x)v$ retourne la valeur écrite par l'opération d'écriture précédente la plus récente.

Ordre du programme ($\xrightarrow{po}$)

On dit qu'une opération $o1$ précede l'opération $o2$ selon l'ordre du programme, et on note $o1 \xrightarrow{po} o2$ si et seulement si:

1. Les opérations $o1$ et $o2$ sont lancées par le même processeur p_i et $o1$ est le précedent immédiat de $o2$ dans le code du programme du processeur p_i, ou
2. $\exists\, o3$ telle que $o1 \xrightarrow{po} o3$ et $o3 \xrightarrow{po} o2$.

Nous pouvons noter que l'ordre du programme $\xrightarrow{po}$ est partiel en H car il ne fait pas la relation entre les opérations lancées par des processeurs différents.

Modèle de cohérence de la mémoire

Un *modèle de cohérence de la mémoire (MCM)* est une relation d'ordre ($\xrightarrow{R}$) sur l'ensemble des accès à la mémoire partagée.

Un *historique d'exécution H respecte un modèle de cohérence* s'il respecte l'ordre $\xrightarrow{R}$ défini par le modèle.

Par la suite, nous présentons les modèles de cohérence mémoire les plus courants. Cinq modèles uniformes et trois modèles hybrides sont présentés. Dans chaque catégorie, l'ordre de présentation est celle du modèle le plus fort au modèle le plus relâché.

2.2.3 Cohérence atomique

Définition originelle

La mémoire atomique est le plus ancien et le plus fort des modèles de cohérence de la mémoire. Dans la cohérence atomique, le temps est divisé en intervalles de temps non simultanés. Les opérations d'accès à la mémoire sont terminées par rapport à tous les processeurs pendant l'intervalle de l'opération [HA90] [HS92] [Mos93]. En d'autres termes, l'opération o_{p_i} est terminée avant que le processeur qui a lancé o_{p_i} puisse lancer la prochaine opération. L'ordre réel des accès à la mémoire doit aussi être préservé.

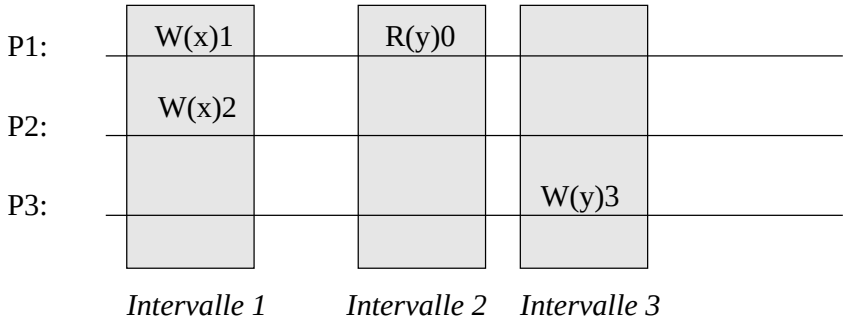


FIG. 2.6 – *Les intervalles de temps sur la cohérence atomique*

La figure 2.6 répresente les intervalles de temps considérés par la cohérence atomique. Les opérations $w_{p_1}(x)1$ et $w_{p_2}(x)2$ sont dites concurrentes car elles sont exécutées dans le même intervalle. En revanche, les opérations $r_{p_1}(y)0$ et $w_{p_3}(y)3$ sont dites non concurrentes.

Définition formelle

Pour présenter la relation d'ordre $\xrightarrow{R}$ qui définit la cohérence atomique, nous avons besoin d'introduire un concept suplementaire:

Temps global (t) - Pour les systèmes qui disposent d'une horloge globale, la fonction $t(\xi)$ nous donne la valeur de l'horloge globale au moment de l'événement ξ.

La définition présentée ci-dessous a été présentée par Raynal et Schiper dans [RS95]:

Un historique H respecte la cohérence atomique s'il existe une séquence linéaire légale $\xrightarrow{AT}$ des opérations de l'historique d'exécution H telle que

(i) $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{AT} o2$, *et*

(ii) $\forall o1, o2$ *où* $t(fin(o1)) < t(ini(o2))$ *alors* $o1 \xrightarrow{AT} o2$

Selon cette définition, tous les processeurs doivent percevoir le même ordre d'exécution de toutes les opérations d'accès à la mémoire (séquence linéaire légale de H). Dans cet ordre, toutes les opérations effectuées par un même processeur doivent être perçues selon l'ordre de son programme (condition (i)). De plus, les opérations d'accès à la mémoire qui ne sont pas simultanées doivent apparaître dans l'ordre $\xrightarrow{AT}$ selon le temps réel de l'exécution de l'opération (condition (ii)). La cohérence atomique nécessite donc d'une notion d'horloge globale.

Modèles dérivés

Dans la figure 2.6 nous pouvons noter que plusieurs accès à la même donnée peuvent se dérouler pendant le même intervalle d'opération (un accès par processeur). Ceci peut engendrer des problèmes de cohérence. Les différentes solutions à ces problèmes de cohérence des accès simultanés ont généré une diversité de modèles fondés sur la cohérence atomique [Lam86]. Dans ce qui suit, cinq modèles dérivés sont décrits. Les quatre premiers ont été définis en analysant le comportement des registres d'un processeur. Le dernier est une condition de correction

pour les objets concurrents.

- **Cohérence atomique statique (AT_STAT)**: Une opération d'accès à la mémoire se rend visible à tous les processeurs à un instant donné de l'intervalle de l'opération. En général, les opérations de lecture sont perçues en début d'intervalle et les écritures sont perçues à la fin de l'intervalle.

- **Cohérence atomique dynamique (AT_DYN)**: Une opération d'accès à la mémoire se rend visible à tous les processeurs à n'importe quel point de l'intervalle de l'opération. Néanmoins, les valeurs rétournées par une lecture doivent être cohérentes avec une des séquences linéaires légales de H possibles.

- **Mémoire Régulière (AT_REG)**: De façon similaire à la mémoire atomique dynamique, l'opération d'accès à la mémoire doit être perçue par tous les observateurs à n'importe quel point de l'intervalle de l'opération. Cependant, plusieurs lecteurs concurrents ne sont pas obligés de choisir des valeurs cohérentes avec une séquence linéaire de H. Les valeurs choisies doivent, néanmoins, être une des valeurs écrites. Pendant les accès concurrents, cette mémoire se comporte de façon très flexible. En d'autres termes, la condition de séquence linéaire de H n'est plus respectée dans le cas des accès simultanés. Les accès non concurrents restent pourtant atomiques.

- **Mémoire Sûre (AT_SURE)**: La mémoire sûre se comporte comme la mémoire régulière sauf pour le cas des lectures concurrentes avec l'écriture d'une même donnée. Dans ce cas, n'importe quelle valeur peut être retournée par les lectures concurrentes, même une valeur différente de celle qui y était et de celle qu'on veut écrire. De façon similaire au cas précedent, les accès non concurrents sont atomiques.

- **Linearisabilité**: Tous les observateurs doivent se mettre d'accord à propos de l'ordre de toutes les opérations d'accès à la mémoire. En outre, l'ordre réel des opérations non simultanées doit être préservé. Ainsi, une opération peut être perçue en dehors de son intervalle, mais jamais avant la fin de l'intervalle précédent ni après le début de l'intervalle suivant.

La figure 2.7 représente un historique d'exécution valide sur la mémoire atomique.

Dans ce cas, p_2 lit la valeur de x écrite par p_1. Comme les deux opérations ne sont pas exécutées simultanément, la lecture d'une valeur autre que $x = 1$ est impossible sur la mémoire atomique.

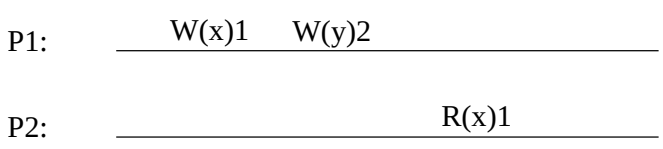


FIG. 2.7 – *Cohérence atomique*

2.2.4 Cohérence séquentielle (SC)

Définition originelle

La cohérence séquentielle a été définie par Lamport dans [Lam79] comme un critère de correction pour les multiprocesseurs à mémoire partagée. Selon Lamport, un multiprocesseur est séquentiellement cohérent si "*le résultat d'une exécution est équivalent à celui de l'exécution des opérations de tous les processeurs dans un ordre séquentiel et les opérations de chaque processeur apparaissent dans cette séquence suivant l'ordre spécifié par son programme*".

La cohérence séquentielle est moins stricte que la cohérence atomique puisque la préservation de l'ordre réel des accès non concurrents n'est pas nécessaire. Cependant, tous les processeurs doivent percevoir le même ordre de toutes les opérations d'accès à la mémoire partagée. La définition de la cohérence séquentielle permet que les effets des opérations d'accès à la mémoire soient retardés et même permutés entre eux.

Définition formelle

La définition de la cohérence séquentielle présentée ci-dessous a été proposée par Ahamad et al. dans [A+92]:

Un historique H respecte la cohérence séquentielle s'il existe une séquence linéaire légale $\xrightarrow{SC}$ des opérations de H telle que

(i) $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{SC} o2$

Cette définition diffère de celle présentée pour la cohérence atomique car le maintien de l'ordre reél des accès à la mémoire n'est plus nécessaire. Les pro-

cesseurs continuent à percevoir le même ordre d'exécution de tous les accès à la mémoire (séquence linéaire légale de H) et l'ordre du programme de chaque processeur doit être respecté dans cette séquence (condition (i)).

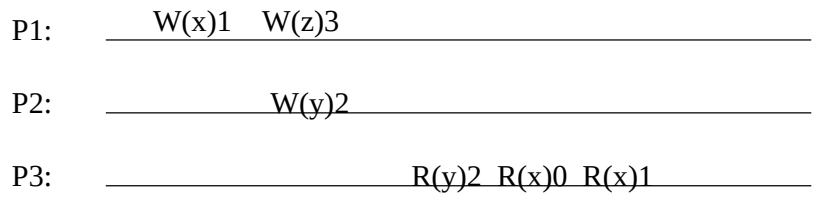


FIG. 2.8 – *Cohérence Séquentielle*

La figure 2.8 représente un historique d'exécution valide dans la cohérence séquentielle mais invalide dans la cohérence atomique car la lecture de $x = 0$ par p_3 est effectuée après la fin de l'écriture de $x = 1$ par p_1. Cependant, il existe une séquence linéaire légale de H: $w_{p_2}(y)2 \xrightarrow{SC} r_{p_3}(y)2 \xrightarrow{SC} r_{p_3}(x)0 \xrightarrow{SC} w_{p_1}(x)1 \xrightarrow{SC} r_{p_3}(x)1 \xrightarrow{SC} w_{p_1}(z)3$. Ceci rend cet historique possible sur la cohérence séquentielle.

Implantation

La plupart des premiers systèmes à mémoire virtuelle partagée proposés se comportent selon la cohérence séquentielle. Ils implantent en fait la condition suffisante pour la cohérence séquentielle proposée par Scheurich et Dubois dans [SD87]. Selon cette condition, toutes les opérations d'accès à la mémoire doivent être lancées selon l'ordre du programme et une opération d'accès à la mémoire est lancée uniquement quand l'opération précédente est terminée. De plus, la cohérence entre les différentes copies d'une même donnée doit être garantie.

Les systèmes Ivy [LS89], Mirage [FP89], Gothic [Roc90] et Memnet [TSF90] sont des exemples de systèmes qui implantent la condition suffisante énoncée ci-dessus.

D'autres chercheurs ont proposé des implantations plus performantes de la cohérence séquentielle. Les implantations proposées par Gharachorloo et al. dans [GGH91] et Afek et al. dans [ABM93] se servent du même principe. Dans les deux cas, les opérations continuent à être lancées selon l'ordre du programme. Néanmoins, plusieurs écritures peuvent être réalisées simultanément par le même processeur. C'est la prochaine lecture qui doit attendre que toutes les écritures précedentes soient terminées. Ce type d'implantation permet une sorte de concurrence des écritures.

Comme la cohérence séquentielle garantit que le même ordre de tous les accès à la mémoire est perçu par tous les processeurs, la plupart des applications

conçues pour une machine monoprocesseur s'exécutent correctement sur la cohérence séquentielle. Les seules applications qui peuvent obtenir des résultats inattendus sont celles qui raisonnent à propos du temps réel. En effet, l'utilisation des horloges doit se faire avec précaution.

2.2.5 Cohérence Causale

Définition originelle

La mémoire causale est fondée sur la relation de causalité potentielle définie par Lamport dans [Lam78]. La cohérence causale exige que tous les processeurs soient d'accord à propos de l'ordre d'exécution des opérations ordonnancées par l'ordre causal. Les opérations causalement indépendantes (dites concurrentes) peuvent être perçues dans un ordre différent par des processeurs distincts. Ainsi, la mémoire causale n'établit qu'un ordre partiel des opérations de H.

En général, une écriture dans la mémoire causale peut être vue comme l'envoi d'un message et une lecture se comporte comme la réception d'un message.

Définition formelle

Pour présenter la définition formelle de la cohérence causale, nous avons besoin d'introduire quelques fonctions et relations supplémentaires:

Type de l'opération (*type*) - L'opération d'accès à la mémoire $o1$ est du type w s'il s'agit d'une écriture en mémoire ($type(o1) = w$). L'opération $o1$ est du type r s'il s'agit d'une lecture en mémoire ($type(o1) = r$).

Adresse de l'opération (*adr*) - L'adresse $adr(o1)$ est l'adresse mémoire sur laquelle l'opération $o1$ agit.

Valeur de l'opération (*val*) - La valeur $val(o1)$ est la valeur manipulée par l'opération $o1$.

Processeur de l'opération (*proc*) - Le processeur $proc(o1)$ est le processeur qui a lancé l'opération $o1$.

Ordre "read-by" $\xrightarrow{rb}$ - Si une opération de lecture $o2$ effectuée par un processeur i lit la valeur v écrite par une opération $o1$ d'un processeur j et $i \neq j$, alors $o1 \xrightarrow{rb} o2$. En d'autres termes, si $proc(o1) \neq proc(o2)$ et $type(o1) = w$ et $type(o2) = r$ et $adr(o1) = adr(o2)$ et $val(o1) = val(o2)$ alors $o1 \xrightarrow{rb} o2$.

Comme la plupart des auteurs, nous supposons sans perte de généralité que deux opérations d'écriture sur la même adresse mémoire écrivent toujours des valeurs différentes. En d'autres termes, si $type(o1) = type(o2) = w$ et $adr(o1) = adr(o2)$ alors $val(o1) \neq val(o2)$.

La cohérence causale a été proposée par Ahamad et al. dans [AHJ90] comme la fermeture transitive de l'ordre du programme et de l'ordre "read-by".

Pour présenter la définition formelle de la cohérence causale proposée par John et Ahamad dans [JA93], il faut d'abord définir le sous-historique H_{p_i+w} de H suivant:

Historique des écritures - Soit H un historique d'exécution et p_i un processeur. On appele historique des écritures par rapport au processeur p_i et on note H_{p_i+w} le sous-historique de H constitué de toutes les opérations d'accès à la mémoire effectuées par p_i et de toutes les opérations d'écriture effectuées par tous les autres processeurs.

Un historique H respecte la cohérence causale s'il existe une séquence linéaire légale $\xrightarrow{CA}$ des opérations de H_{p_i+w} pour chaque processeur p_i telle que:

- *(i)* $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{CA} o2$ *et*
- *(ii)* $\forall o1, o2$ *où* $o1 \xrightarrow{rb} o2$ *alors* $o1 \xrightarrow{CA} o2$ *et*
- *(iii) si* $o1 \xrightarrow{CA} o2$ *et* $o2 \xrightarrow{CA} o3$ *alors* $o1 \xrightarrow{CA} o3$.

Nous pouvons noter que, au contraire des deux modèles précédents, il n'est plus nécessaire que tous les processeurs soient d'accord à propos de tous les accès à la mémoire partagée. Maintenant, chaque historique local d'exécution doit considérer uniquement ses propres opérations et les opérations d'écriture effectuées par les autres processeurs (H_{p_i+w}). Dans cette séquence, l'ordre du programme et l'ordre des lectures d'une écriture doivent être respectés (conditions (i) et (ii)). La transitivité doit aussi être assurée (condition (iii)).

La figure 2.9 représente un historique d'exécution valide sur la cohérence causale mais invalide sur la cohérence séquentielle.

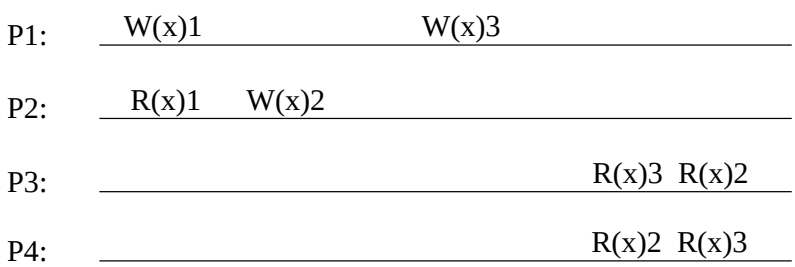


FIG. 2.9 – *Cohérence Causale*

Les processeurs p_3 et p_4 perçoivent les écritures à x dans un ordre distinct. Il n'existe pas de séquence linéaire légale sur H qui rende cette exécution valide dans la cohérence séquentielle. Cependant, puisque les écritures $w_{p_2}(x)2$ et $w_{p_1}(x)3$ ne sont pas ordonnées par $\xrightarrow{CA}$, cet historique est valide dans la cohérence causale.

Nous présentons maintenant des exemples d'historiques H_{p_i} légaux dans la cohérence causale pour l'historique H de la figure 2.9.

$\boldsymbol{H_{p_1+w}}$: $w_{p_1}(x)1 \xrightarrow{CA} w_{p_2}(x)2 \xrightarrow{CA} w_{p_1}(x)3$

$\boldsymbol{H_{p_2+w}}$: $w_{p_1}(x)1 \xrightarrow{CA} r_{p_2}(x)1 \xrightarrow{CA} w_{p_2}(x)2 \xrightarrow{CA} w_{p_1}(x)3$

$\boldsymbol{H_{p_3+w}}$: $w_{p_1}(x)1 \xrightarrow{CA} w_{p_1}(x)3 \xrightarrow{CA} r_{p_3}(x)3 \xrightarrow{CA} w_{p_2}(x)2 \xrightarrow{CA} r_{p_3}(x)2$

$\boldsymbol{H_{p_4+w}}$: $w_{p_1}(x)1 \xrightarrow{CA} w_{p_2}(x)2 \xrightarrow{CA} r_{p_4}(x)2 \xrightarrow{CA} w_{p_1}(x)3 \xrightarrow{CA} r_{p_4}(x)3$

Implantation

Les auteurs qui ont proposé la cohérence causale ont aussi proposé une implantation possible de ce modèle. Cette implantation se sert d'un ensemble d'estampilles pour maintenir la relation de causalité. Chaque fois qu'une valeur est visible à un processeur, toutes les valeurs plus anciennes sont invalidées. Cette implantation garantit une condition suffisante de la cohérence causale puisqu'elle génère plus d'invalidations que nécessaire [AHJ90]. Le mécanisme décrit ci-dessus a aussi été implanté dans le système Clouds [JA93]. Dans Clouds, quelques politiques sont proposées pour réduire le nombre d'invalidations superflues.

Ahamad et al. présentent dans [AHJ90] un algorithme synchrone pour la résolution d'équations linéaires qui s'exécute correctement dans la cohérence atomique et dans la cohérence causale. Dans le même papier, un algorithme est décrit qui résout le problème du dictionnaire réparti.

2.2.6 Cohérence du processeur

La cohérence du processeur est apparue pour offrir plus de concurrence aux accès d'écriture (*write pipelining*). Deux modèles très proches ont été définis dans ce but: PRAM et PC.

Définition originelle (PRAM)

Le modèle Pipelined Random Access Memory (PRAM) a été initialement défini par Lipton et Sandberg dans [LS88]. Il exige que les écritures effectuées par un processeur soient perçues par tous les autres processeurs dans l'ordre du programme du processeur qui a écrit la donnée. Les écritures effectuées par des processeurs distincts peuvent apparaître dans un ordre différent sur les historiques locaux des processeurs.

Définition formelle (PRAM)

Pour la définition de la relation $\xrightarrow{R}$ associée à la cohérence PRAM, il nous faut définir un nouveau sous-historique $H_{p_i}|w$ comme suit:

Historique des écritures de P_i - Soit H_{p_i} un historique local d'exécution et p_i un processeur. On appele historique des écritures de p_i et on note $H_{p_i|w}$ le sous-historique de H_{p_i} constitué de toutes les opérations d'écriture effectuées par p_i.

La définition présentée ci-dessous a été proposée par Ahamad et al. dans [A+92]:

Un historique H respecte la cohérence PRAM s'il existe une séquence linéaire légale $\xrightarrow{PRAM}$ des opérations de H_{p_i+w} pour chaque processeur p_i telle que:

- *(i)* $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{PRAM} o2$ *et*
- *(ii)* $\forall o1, o2$ *si* $\exists H_{p_j|w}$ *où* $o1 \xrightarrow{H_{p_j}|w} o2$ *alors* $o1 \xrightarrow{PRAM} o2$

La cohérence PRAM est garantie si l'ordre du programme de chaque processeur est respecté (condition (i)) et si les écritures effectuées par un même processeur sont perçues dans l'ordre de son historique des écritures par tous les autres processeurs (condition (ii) + séquence linéaire légale de H_{p_i+w}).

La figure 2.10 représente un historique d'exécution valide dans le modèle PRAM mais invalide dans la cohérence causale.

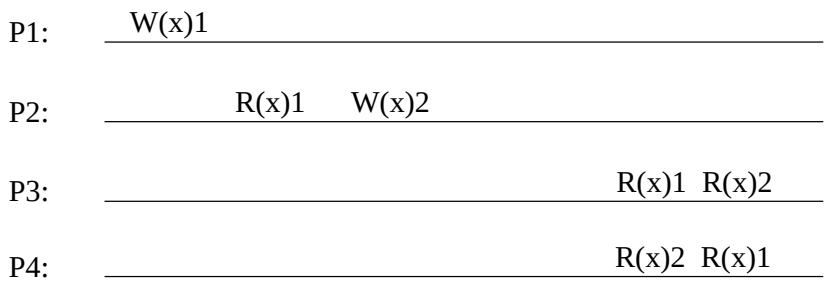


FIG. 2.10 – *Cohérence PRAM*

L'écriture $w_{p_1}(x)1$ est ordonnée avant $w_{p_2}(x)2$ par l'ordre causal $w_{p_1}(x)1 \xrightarrow{CA} r_{p_2}(x)1 \xrightarrow{CA} w_{p_2}(x)2$. L'historique H_{p4} n'est pas possible dans la cohérence causale puisque l'opération $r_{p_4}(x)2$ ne peut pas être suivie de $r_{p_4}(x)1$. Cet historique est pourtant possible sur la cohérence PRAM car l'ordre des écritures des processeurs est garantie.

Nous montrons par la suite les historiques locaux associés à chaque processeur:

$\boldsymbol{H_{p_1+w}}$: $w_{p_1}(x)1 \xrightarrow{PRAM} w_{p_2}(x)2$

$\boldsymbol{H_{p_2+w}}$: $w_{p_1}(x)1 \xrightarrow{PRAM} r_{p_2}(x)1 \xrightarrow{PRAM} w_{p_2}(x)2$

$\boldsymbol{H_{p_3+w}}$: $w_{p_1}(x)1 \xrightarrow{PRAM} r_{p_3}(x)1 \xrightarrow{PRAM} w_{p_2}(x)2 \xrightarrow{PRAM} r_{p_3}(x)2$

$\boldsymbol{H_{p_4+w}}$: $w_{p_2}(x)2 \xrightarrow{PRAM} r_{p_4}(x)2 \xrightarrow{PRAM} w_{p_1}(x)1 \xrightarrow{PRAM} r_{p_4}(x)1$

Définition originelle (PC)

Goodman [Goo89] a rajouté la cohérence du cache au modèle PRAM. Ceci a généré un nouveau modèle de cohérence appelé cohérence du processeur (PC). En plus de la préservation de l'ordre des écritures de chaque processeur, ce modèle exige que la cohérence entre les différentes copies d'une même donnée soit respectée.

Définition formelle (PC)

Pour la définition de la relation $\xrightarrow{R}$ associée à la cohérence PC, il nous faut définir un nouveau sous-historique $H|w(x)$ comme suit:

Historique des écritures sur x - Soit H un historique d'exécution. On appele historique des écritures sur x et on note $H|w(x)$ le sous-historique de H constitué de toutes les opérations d'écriture effectuées sur l'adresse x.

La définition de la cohérence PC est alors la suivante [A+92]:

Un historique H respecte la cohérence PC s'il existe une séquence linéaire légale $\xrightarrow{PC}$ des opérations de H_{p_i+w} pour chaque processeur p_i telle que:

- *(i)* $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{PC} o2$ *et*
- *(ii)* $\forall o1, o2$ *si* $\exists H_{p_j|w}$ *où* $o1 \xrightarrow{H_{p_j}|w} o2$ *alors* $o1 \xrightarrow{PC} o2$
- *(iii) Pour chaque adresse x, il existe une séquence linéaire légale $\xrightarrow{wb}$ de $H|w(x)$ et* $\forall o1, o2$ *où* $o1 \xrightarrow{wb} o2$ *alors* $o1 \xrightarrow{PC} o2$.

La définition présentée ci-dessus diffère de celle de la cohérence PRAM par la présence de la condition (iii) qui garantit la cohérence du cache.

Implantation

Des variations de la cohérence du processeur ont été implantées sur des machines parallèles VAX 8800 et PLUS au niveau du matériel [Mos93] [BR90].

Gharachorloo et al. ont proposé dans [G+90] deux conditions suffisantes pour implanter la cohérence du processeur. Pour chaque processeur p_i, une lecture ne se termine que quand toutes les lectures précedentes sont terminées et une écriture ne se termine que quand tous les accès précédents sont terminés.

Les systèmes qui se servent de la cohérence du processeur sont en général très performants puisque l'utilisation des tampons d'écriture peut se faire en sa totalité.

Ahamad et al. montrent dans [A+92] que l'algorithme d'exclusion mutuelle proposé par Peterson [Tan92] s'exécute correctement dans la cohérence du processeur. Cet algorithme se sert de la cohérence du cache pour garantir l'exclusion mutuelle entre les processus. Néanmoins, les auteurs montrent dans le même papier qu'un autre algorithme d'exclusion mutuelle ne s'y exécute pas correctement. Ceci arrive car, dans la cohérence du processeur, les écritures peuvent être ordonnées après la fin de toutes les lectures locales.

2.2.7 Mémoire lente (LE)

Définition originelle

La mémoire lente a été proposée par Hutto et Ahamad dans [HA90]. Dans la mémoire lente, les écritures se propagent lentement à travers le système. La seule exigence est que tous les processeurs doivent se mettre d'accord à propos des écritures effectuées par un même processeur sur chaque adresse mémoire [HA90]. Les lectures doivent retourner une valeur qui a été écrite auparavant. Une fois la valeur lue, aucune valeur écrite auparavant par le même processeur sur la même adresse ne peut être retournée. Les écritures locales sont visibles immédiatement.

Définition formelle

Pour la définition de la relation $\xrightarrow{R}$ associée à la mémoire lente, il nous faut un nouveau sous-historique de H_{p_i}, défini comme suit:

Historique des écritures de p_i sur l'adresse x - Soit H_{p_i} un historique local d'exécution et p_i un processeur. On appele historique des écritures du processeur p_i sur l'adresse x et on note $H_{p_i}|w(x)$ le sous-historique de H_{p_i} constitué de toutes les opérations d'écriture effectuées par le processeur p_i sur l'adresse x.

La définition présentée ci-dessous a été proposée par Heddaya et Sinha dans [HS92]:

Un historique H respecte la cohérence lente s'il existe une séquence linéaire légale $o1 \xrightarrow{LE} o2$ des opérations de H_{p_i+w} pour chaque processeur p_i telle que:

- *(i)* $\forall o1, o2$ *où* $o1 \xrightarrow{po} o2$ *alors* $o1 \xrightarrow{LE} o2$ *et*
- *(ii)* $\forall o1, o2$ *si* $\exists p_j, x$ *tel que* $o1 \xrightarrow{H_{p_j}|w(x)} o2$, *alors* $o1 \xrightarrow{LE} o2$

La condition (i) garantit que l'ordre du programme du processeur p_i est respectée dans l'historique H_{p_i}. La condition (ii) garantit que les écritures effectuées par un même processeur sur une même adresse seront perçues par tous les autres processeurs dans le même ordre.

La figure 2.11 représente un historique d'exécution valide dans la cohérence lente invalide dans la cohérence du processeur (l'ordre des écritures n'est pas respecté).

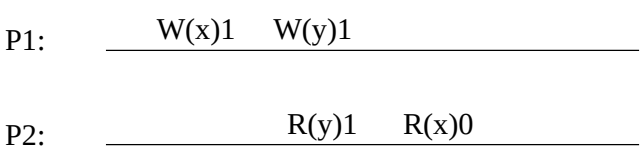


FIG. 2.11 – *Mémoire Lente*

Les historiques locaux résultants sont:

$\boldsymbol{H_{p_1+w}}$: $w_{p_1}(x)1 \xrightarrow{LE} w_{p_1}(y)1$

$\boldsymbol{H_{p_3+w}}$: $w_{p_1}(y)1 \xrightarrow{LE} r_{p_2}(y)1 \xrightarrow{LE} r_{p_2}(x)0 \xrightarrow{LE} w_{p_1}(x)1$

Implantation

Hutto et Ahamad montrent dans [HA90] comment utiliser la mémoire lente pour implanter l'exclusion mutuelle physique et pour résoudre le problème du dictionnaire. La mémoire lente a aussi été implantée comme un des modèles de cohérence offerts par le système Mermera [HS92]. Dans ce cas, les écritures sont propagées à travers une primitive de diffusion qui ne garantit aucune contrainte d'ordre.

2.2.8 Cohérence faible

Les modèles de cohérence hybrides sont nés d'une constatation faite déjà à propos des systèmes monoprocesseurs multiprogrammés: pour qu'un algorithme s'exécute correctement sur un modèle de programmation à mémoire partagée, les restrictions d'ordre d'accès doivent être garanties par les primitives de synchronisation [Tan87]. L'utilisation de ces primitives permet que les opérations ordinaires d'accès à la mémoire soient exécutées de façon très performante. La cohérence n'est assurée qu'au moment de l'exécution des accès de synchronisation.

Définition originelle

La cohérence faible est le premier modèle de cohérence de la mémoire à faire la distinction entre les opérations ordinaires d'accès à la mémoire et les opérations de synchronisation. Ce modèle a été proposé par Dubois et Scheurich dans [DSB86]. Les accès aux variables partagées sont divisés en accès ordinaires et accès de synchronisation. La cohérence séquentielle n'est assurée qu'au moment des accès de synchronisation.

Selon la définition originelle, "*un système mémoire est faiblement cohérent si:*

- *(i) les accès aux variables de synchronisation respectent la cohérence séquentielle, et*
- *(ii) aucun accès à une variable de synchronisation n'est effectué avant la terminaison de tous les accès aux variables ordinaires précedents, et*
- *(iii) avant la fin d'un accès de synchronisation aucun accès aux variables ordinaires postérieur à l'accès de synchronisation n'est lancé".*

Au moment de l'exécution d'un accès de synchronisation, tous les accès précédents sont terminés et aucun accès futur n'est initié. Un programme qui s'exécute sur la mémoire faible parait séquentiellement cohérent si:

- il n'existe pas de conflits d'accès, et
- le matériel est capable de distinguer les variables de synchronisation.

Définition révisée

Em 1990, Adve et Hill ont proposé dans [AH90] une nouvelle définition de la cohérence faible: "*un système est faiblement cohérent par rapport à un modèle de synchronisation si et seulement si le système paraît se comporter selon la cohérence séquentielle pour tous les programmes qui respectent le modèle de synchronisation*".

Pour définir la cohérence faible, il suffit alors de définir le modèle de synchronisation associé. La définition des modèles de synchronisation utilise un nouveau type d'opération: $synch_{p_i}(x)v$.

Un des modèles de synchronisation proposés par Adve et Hill est le DRF0 (*data-race-free-0*). La définition du modèle DRF0 utilise l'ordre de synchronisation ($\xrightarrow{so}$) et l'ordre "s'est-produit-avant" ($\xrightarrow{hb}$) définis comme suit:

Ordre de synchronisation ($\xrightarrow{so}$) - L'opération $o1$ est ordonnée avant l'opération $o2$ selon l'ordre de synchronisation $o1 \xrightarrow{so} o2$ si et seulement si $o1$ et $o2$ sont des opérations de synchronisation ($type(o1) = type(o2) = synch$) qui accèdent la même position mémoire et $o1$ se termine avant le début de $o2$.

Ordre "s'est-produit-avant" ($\xrightarrow{hb}$) - L'ordre s'est-produit-avant $\xrightarrow{hb}$ est la fermeture irreflexive transitive de l'union de l'ordre du programme et de l'ordre de synchronisation. En d'autres termes, $\xrightarrow{hb} = \left(\xrightarrow{po} \cup \xrightarrow{so}\right)^+$

Selon Adve et Hill, "*un historique d'exécution respecte le modèle de synchronisation DRF0 si et seulement si:*

- *(i) les opérations de synchronisation agissent sur une seule opération mémoire;*
- *(ii) l'ordre du programme de chaque processeur est respecté et*
- *(iii) tous les accès à la mémoire qui sont en conflit sont ordonnés par l'ordre* $\xrightarrow{hb}$ *".*

Implantation

Adve et Hill proposent une implantation de la cohérence faible où le processeur qui exécute l'accès de synchronisation n'attend pas que les accès précédents à l'accès de synchronisation soient terminés pour continuer son exécution.

Cette implantation ne respecte pas le modèle de cohérence faible tel qu'il a été initialement conçu (violation de la condition (iii) de Dubois et Scheurich). Néanmoins, l'implantation respecte le modèle de synchronisation DRF0.

2.2.9 Cohérence à la libération (RC)

Définition originelle

De façon similaire à la cohérence faible, la cohérence à la libération (RC) utilise aussi les points de synchronisation pour garantir la cohérence de la mémoire. Les opérations d'accès à la mémoire sont divisées en accès ordinaires et accès spéciaux. Les accès ordinaires sont des lectures et écritures qui ne sont jamais en compétition. Les accès spéciaux sont des accès en compétition et sont à leur tour divisés en: accès concurrents, accès d'obtention et accès de relâchement. Les accès concurrents sont des accès en compétition qui ne sont pas utilisés pour synchroniser des processus (e.g. relaxation chaotique [Mos93]). Un accès d'obtention garantit l'accès exclusif à un ensemble de variables partagées jusqu'à ce qu'un accès de relâchement soit exécuté. Les opérations de cohérence sont effectuées au moment de la libération [ZB92].

La cohérence à la libération permet plus d'optimisations que la cohérence faible. Contrairement à celle-ci, les accès postérieurs à la libération ne sont pas retardés jusqu'à ce que le relâchement soit terminé. De façon similaire, l'obtention n'est pas retardée à cause des accès qui le précèdent.

La cohérence à la libération a été initialement définie par Gharachorloo et al. dans [G+90] comme suit:

" *Un système obéit à la cohérence à la libération (RC) si et seulement si*

- *(i) Avant l'exécution d'un accès ordinaire, tous les accès d'obtention précédents sont terminés;*
- *(ii) Avant l'exécution d'un accès de relâchement, toutes les lectures et écritures ordinaires précedentes sont terminées;*
- *(iii) Les accès de synchronisation sont ordonnés par la cohérence du processeur".*

Les programmes dits "correctement étiquetés" produisent les mêmes résultats tant sur la mémoire *RC* que sur la mémoire *SC*. Un programme est correctement etiqueté si, pour tous les entrelacements légaux des accès ordinaires, deux accès en compétition sont séparés par une chaîne d'accès de synchronisation (obtention-relâchement).

La cohérence à la libération fait la distinction entre plusieurs types de synchronisation. En plus des types r et w, trois nouveaux types sont utilisés: *acq* pour l'obtention d'un objet de synchronisation, *rel* pour le relâchement d'un objet de synchronisation et *nsync* pour des accès spéciaux qui ne servent pas à la synchronisation.

Définition formelle

Nous présentons maintenant la définition formelle de la cohérence à la libération proposée par Kohli et al. dans [KNA93]. Les accès d'acquisition sont vus comme des lectures spéciales tandis que les accès de libération sont vus comme des écritures spéciales.

De façon similaire aux modèles uniformes, chaque processeur aura un historique H_{p_i+w} où les opérations de lecture et les opérations d'acquisition effectuées par les autres processeurs sont exclues.

La relation $\xrightarrow{R}$ est définie comme suit par la cohérence à la libération:

Un historique H respecte la cohérence RC s'il existe une séquence linéaire $o1 \xrightarrow{RC} o2$ des opérations de H_{p_i+w} telle que:

- $\forall o1, o2, o3$ *où* $o1 \xrightarrow{so} o2 \xrightarrow{po+} o3$ *en* H *et* $type(o1) = rel$ *et* $type(o2) = acq$ *et* $type(o3) \in \{r, w\}$, *alors* $o1 \xrightarrow{RC} o3$ *et*
- $\forall o1, o2$ *où* $o1 \xrightarrow{po+} o2$ *en* H_{p_i+w} *et* $type(o1) \in \{r, w\}$ *et* $type(o2) = rel$, *alors* $o1 \xrightarrow{RC} o2$.

Deux variations de la cohérence à la libération sont très répandues: la cohérence à la libération paresseuse (LRC) et la cohérence séquentielle à la libération (RC_{sc}). Les deux ont été proposées comme techniques d'implantation de la cohérence à la libération.

- ***LRC*** - Sur la cohérence paresseuse à la libération, la propagation des écritures sur la mémoire partagée est retardée jusqu'au moment de la prochaine exécution d'une opération d'obtention. A ce moment, tous les accès partagés qui précèdent l'opération d'obtention sont terminés par rapport au processeur qui exécute l'opération d'obtention [KCZ92]. Ainsi, la condition (ii) n'est plus respectée.
- ***RC$_{sc}$*** - Sur la cohérence séquentielle à la libération, la condition (iii) est plus forte que sur la cohérence RC car les accès de synchronisation doivent observer la cohérence séquentielle (SC).

Implantation

La cohérence à la libération a été implantée sur plusieurs systèmes à mémoire virtuelle partagée. Le système Munin, proposé par Carter dans [Car93] est un environnement de programmation qui implante la cohérence à la libération RC_{sc}. L'implantation proposée se sert parmi d'autres d'un protocole d'écrivains multiples où les modifications se rendent visibles aux autres processeurs au moment de l'exécution d'un accès de libération.

TreadMarks a été proposé par Keleher et al [KDCZ93]. C'est un système à mémoire virtuelle partagée qui implante la cohérence à la libération LRC. De façon similaire à Munin, TreadMarks utilise un protocole à écrivains multiples pour la diffusion des écritures. Contrairement à Munin, les écritures sont diffusées au

moment de la prochaine acquisition de l'objet de synchronisation. Pour maintenir l'ordre de synchronisation, le système se sert d'un ensemble d'estampilles.

DASH [LLJ+93] est une architecture parallèle où la cohérence à la libération est garantie par le matériel. Un accès de libération ne se termine que quand toutes les écritures précédentes sont terminées. Cette approche est différente de celle adoptée dans Munin, qui stocke les modifications et les diffuse toutes au moment de la libération.

2.2.10 Cohérence d'entrée

Définition

La cohérence d'entrée est un modèle hybride proposé par Bershad et Zekauskas [BZ91b] qui prend en compte la relation entre les variables de synchronisation qui protègent les sections critiques et les variables partagées qui se trouvent à l'intérieur des sections critiques. La mémoire n'est cohérente qu'au moment où le processeur obtient le verrou et entre en section critique. De plus, les données cohérentes ne sont que celles accédées à l'intérieur de cette région.

Les verrous sont obtenus soit de manière exclusive soit de façon non-exclusive. Les variables de synchronisation non-exclusives peuvent appartenir simultanément à plusieurs processeurs et implantent des sections critiques en lecture.

Selon Bershad et Zekauskas, "*un système mémoire est cohérent d'entrée si:*

- *(i) un accès d'obtention d'un verrou s ne s'exécute par rapport au processeur p que quand toutes les modifications précédentes faites sur les données gardées par s sont terminées par rapport à p.*
- *(ii) l'exécution d'un accès exclusif à une variable de synchronisation s par le processeur p n'est effectuée que quand aucun processeur ne détient s de façon exclusive.*
- *(iii) après l'obtention d'une variable de synchronisation s par le processeur p de manière exclusive, aucune obtention non-exclusive de s n'est effectuée jusqu'au moment du relâchement de s par p".*

Dans la cohérence d'entrée, plusieurs processeurs peuvent se trouver en même temps à l'intérieur de sections critiques distinctes. Ceci n'est pas possible sur la cohérence à la libération. De plus, plusieurs processeurs peuvent être simultanément à l'intérieur d'une même section critique gardée par une variable de synchronisation non-exclusive.

Implantation

La cohérence d'entrée a été implanté par les auteurs dans le système Midway. De façon similaire à TreadMarks, un ensemble d'estampilles est utilisé pour maintenir l'ordre de synchronisation. La cohérence est assurée au moment de la prochaine obtention de l'objet de synchronisation. Contrairement aux systèmes présentés jusqu'à maintenant, la cohérence n'est garantie que pour les variables gardées pour l'objet de synchronisation.

2.2.11 Autres modèles

Dans ce paragraphe, nous présentons quelques autres modèles de cohérence de la mémoire qui ont été définis dans la littérature. Ces modèles sont moins répandus que les modèles présentés dans les paragraphes précédents. Ainsi, nous ne décrivons ici que leur définitions originelles.

TSO (Total Store Order)

Le modèle TSO a été défini pour l'architecture SPARC [Sun93]. Dans ce modèle, les processeurs ont chacun une mémoire locale. Une mémoire globale est partagée par tous les processeurs. L'écriture est effectuée initialement dans la mémoire locale du processeur qui a lancé l'opération. Ensuite, la nouvelle valeur est stockée dans un tampon d'écriture. La vidange de ce tampon suit l'ordre FIFO. Dès qu'une valeur est écrite dans la mémoire globale, elle est supprimée de la mémoire locale du nœud qui a lancé l'écriture. Les lectures sont effectuées soit localement, quand la valeur se trouve encore dans la mémoire locale, soit dans la mémoire globale, dans le cas contraire.

Cohérence Rétardée

La cohérence rétardée (*delayed consistency*) a été proposée par Dubois et al. [D+91]. La cohérence rétardée permet que les invalidations soient stockées dans un tampon du processeur p_i jusqu'à l'exécution de la prochaine acquisition par le processeur p_i. Ce modèle est en fait un affaiblissement de la cohérence à la libération.

2.2.12 Relation entre les modèles

Les différents modèles de cohérence de la mémoire qui ont été présentés dans les paragraphes précedents peuvent être comparés. Pour la comparaison entre les

modèles, les diagrammes de Venn sont couramment utilisés. Pour le cas des modèles de cohérence, les diagrammes de Venn sont conçus en considérant l'ensemble des historiques possibles sur le modèle.

La taille du carreau associé au modèle donne une idée du nombre d'historiques possibles sur ce modèle. Ainsi, les modèles qui présentent une grande surface sont les modèles plus relâchés.

L'existence d'une intersection entre deux modèles a et b montre que quelques historiques valides dans le modèle a y sont aussi dans le modèle b. Un modèle a est contenu dans un modèle b si tous les historiques produits dans a peuvent être produits dans b.

La figure 2.12 montre la relation entre les modèles uniformes. Cette figure est en effet l'union des diagrammes de Venn présentés dans [HA90], [KNA93] et [A+92].

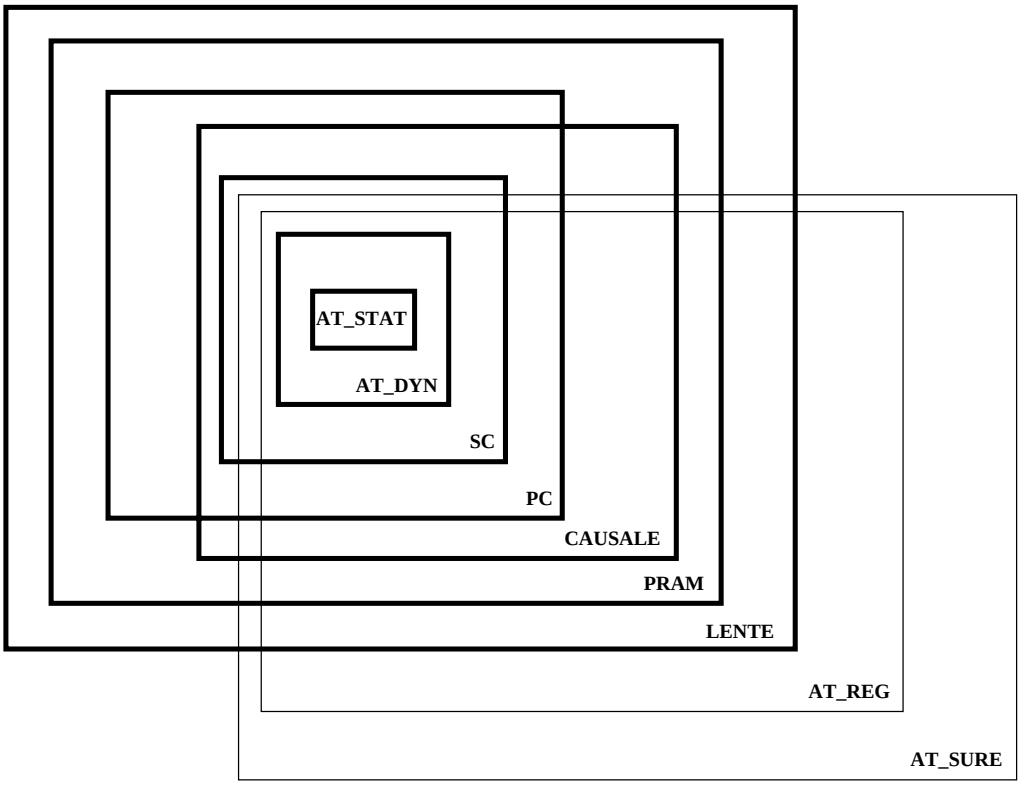


FIG. 2.12 – *Relation entre les modèles uniformes*

Dans la figure 2.12, trois hiérarchies sont réprésentées:

1. AT_STAT $\subset$ AT_DYN $\subset$ AT_REG $\subset$ AT_SURE

2. AT_STAT $\subset$ AT_DYN $\subset$ SC $\subset$ CAUSALE $\subset$ PRAM $\subset$ LENTE

3. AT_STAT $\subset$ AT_DYN $\subset$ SC $\subset$ PC $\subset$ PRAM $\subset$ LENTE

Nous pouvons noter qu'il existe des modèles qui permettent uniquement un sous-ensemble des historiques possibles dans un autre modèle. Ces modèles sont dits incomparables. C'est le notamment le cas des modèles CAUSAL et PC et des modèles AT_SURE et SC, par exemple.

La relation entre les modèles hybrides, répresentée dans la figure 2.13, est beaucoup plus simple.

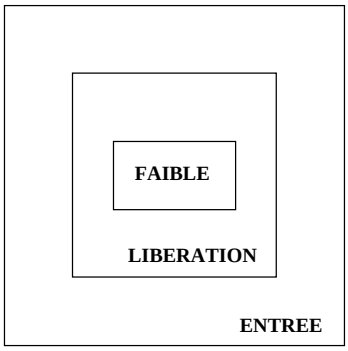


FIG. 2.13 – *Relation entre les modèles hybrides*

Nous voyons ici que la cohérence à la libération est en effet un affaiblissement des contraintes de cohérence imposées par la cohérence faible. De même, la cohérence d'entrée est strictement plus relâchée que les deux autres modèles.

2.3 Opérations de synchronisation distribuées

Les processus qui composent une application parallèle intéragissent souvent pour l'achèvement d'une tâche. Sur la mémoire partagée, cette interaction est réalisée sous la forme de lectures et écritures de variables et se transforme facilement en interférence. L'interférence entre processus peut générer des résultats inattendus et, pour cette raison, doit être contrôlée. Les mécanismes de synchronisation sont mis à disposition des programmeurs pour éviter l'interférence et garantir l'ordonnancement des accès à la mémoire partagée.

De façon générale, la synchronisation engendre la sérialisation des accès. Cette perte de parallélisme est naturelle et est déterminée par l'algorithme. Néanmoins, il existe un surcoût important rajouté par l'implantation des primitives de synchronisation elle-même. Un des défis présentés aux systèmes à mémoire virtuelle partagée est celui de réduire ce surcoût. Dans un environnement parallèle, où les

informations sont partielles et distribuées sur plusieurs nœuds, cette tâche s'avère difficile.

Nous présentons par la suite les mécanismes de synchronisation classiques qui existent dans la littérature aussi bien que les problèmes posés par leur implantation sur les systèmes parallèles.

2.3.1 Support matériel pour la synchronisation

Test&Set

En général, quelques primitives simples et de bas niveau sont implantées directement par le matériel. Le Test&Set est un exemple de primitive simple très répandue. Plusieurs mécanismes de synchronisation plus complexes ont été implantés à l'aide de cette primitive [DSB86].

La sémantique du Test&Set est la suivante:

```
Test&Set(l): temp = l; l = 1;
             return(temp);

Reset(l):    l = 0;
```

L'exclusion mutuelle est garantie par la répétition de $Test\&Set$ jusqu'à ce que la valeur de l soit égale à 0.

Compare&Swap

La primitive Compare&Swap est aussi implantée par le matériel. Sa sémantique est la suivante:

```
Compare&Swap(r1, r2, w): temp = w;
                         si (temp == r1)
                            w = r2; z = 1;
                         sinon
                            r1 = temp; z = 0;
                         return(z);
```

Le Compare&Swap possède trois paramètres: la valeur démandée, la nouvelle valeur et une position mémoire. Si la valeur originale de la position mémoire est égale à la valeur demandée, la nouvelle valeur est attribuée à la position mémoire, de façon atomique. Sinon, faux est retourné.

2.3.2 Primitives de synchronisation

Verrous

Les verrous sont des structures de données partagées qui garantissent l'exclusion mutuelle par logiciel. En général, le verrou est accordé à un processeur à la fois. Pour accéder à une section critique, un processeur doit d'abord obtenir la propriété du verrou (*verrouillage*). A la sortie de la section critique, le verrou est relâché de façon atomique.

La catégorie de verrou la plus simple est le *verrou spin*. Le processus qui désire acquérir un verrou reste en attente active jusqu'à la libération du verrou. L'utilisation de ce type de verrou doit être écartée car l'attente active gaspille le temps du processeur. Néanmoins, les verrous spin sont souvent utilisés sur les multiprocesseurs à cause de la simplicité de leur implantation. Quelques algorithmes performants et extensibles ont été proposés dans [MSG92].

Les *verrous mutex* sont des verrous bloquants. L'attente d'un verrou par un processus est accomplie par le blocage de son exécution. Dès que le verrou est libéré, le processus en attente reprend l'exécution.

Dans certains systèmes, les applications peuvent choisir le type de verrou le plus adapté à leur exécution (spin ou mutex) [MSG92]. D'autres systèmes proposent des verrous adaptatifs, où le système lui-même décide le type de verrou en observant les caractéristiques des applications parallèles.

Un verrou de lecture/écriture permet l'entrée simultanée de plusieurs lecteurs dans une même section critique. Ceci permet une sorte de concurrence dans l'exécution des sections critiques.

Variables condition

Les variables condition sont utilisées pour ordonnancer l'exécution des différents processus. Elles permettent qu'un processus se bloque en attendant une action d'un autre processus. Une variable condition est associée à quelques variables partagées protégées par un verrou et à une condition. Un processus obtient le verrou et examine la condition. Si elle n'est pas vraie, le verrou est relâché de façon atomique et le processus est bloqué. Dès qu'un processus modifie la valeur des variables partagées, les processus en attente reprennent leur exécution et testent la condition de nouveau.

Sémaphores

Les sémaphores ont été définis par Dijkstra dans [Dij65]. Un sémaphore est une variable entière qui n'admet que des valeurs positives. Deux opérations peuvent être exécutées sur un sémaphore: $P(sem)$ et $V(sem)$. L'opération P est définie comme suit:

```
P(sem): si (sem > 0) alors sem = sem − 1;
          sinon  bloque_le_processus();
```

L'opération P décrémente la valeur du sémaphore si cette dernière est supérieure à 0. Si la valeur du sémaphore est 0, le processus qui a exécuté l'opération P est mis en attente. L'exécution de ces opérations est toujours faite de manière atomique.

La sémantique de l'opération V est la suivante:

```
V(sem): si (pas_de_processus_en_attente) sem = sem + 1;
          sinon debloque_processus_en_attente();
```

L'opération V est également indivisible. Elle incrémente la valeur du sémaphore concerné. Les processus en attente sont aussi réveillés.

L'utilisation des sémaphores est très répandue dans l'implantation des systèmes d'exploitation [Les93]. Ils sont aussi utiles pour la construction de primitives de synchronisation plus élaborés.

Barrières de synchronisation

La barrière de synchronisation est utilisée pour synchroniser plusieurs processus parallèles. Tous les processus qui se synchronisent sur une barrière doivent l'atteindre avant que l'exécution continue. Une barrière de synchronisation est définie comme suit:

```
barriere(N): compteur = compteur + 1;
             if (compteur >= N)
               debloque_processus_en_attente();
               compteur = 0;
             sinon
               bloque_processus();
```

Les $N - 1$ premiers processus qui exécutent la barrière sont bloqués. L'exécution de la barrière par le N-ème processus fait que les processus bloqués sur la

barrière reprennent leur exécution.

2.3.3 Implantation des mécanismes de synchronisation

Dans une machine parallèle, l'implantation des mécanismes de synchronisation doit être réalisée de telle sorte que le surcoût rajouté soit petit. En leur essence, ces mécanismes sont critiques puisque généralement il existe une grande contention dans l'accès aux variables de synchronisation.

Les questions soulevées par une implantation parallèle de la synchronisation se posent sur trois aspects: la localisation des variables de synchronisation, la politique utilisée pour accorder une variable à un processus et le mécanisme de communication par lequel la synchronisation sera implantée.

Localisation des objets de synchronisation

La question qui concerne la localisation des objets de synchronisation est en général résolue par des algorithmes de recherche d'entités dans un environnement distribué. Les solutions les plus utilisées sont celles proposées par Li et Hudak dans [LH89]: l'algorithme centralisé, l'algorithme distribué fixe et l'algorithme distribué dynamique. Même si ces algorithmes ont été originellement définis pour la recherche des pages, ils peuvent être également appliqués au cas des variables de synchronisation. La description détaillée de ces algorithmes est présentée dans le paragraphe 2.4.

Politique d'attribution d'objets de synchronisation

Au moment de la libération d'un objet de synchronisation, plusieurs processeurs peuvent être en attente. La politique de verrouillage détermine le prochain processeur à acquérir l'objet. A fin d'assurer l'équité et éviter la famine, une politique FIFO stricte peut être utilisée. Dans ce cas, les processeurs sont servis selon l'ordre d'arrivée des requêtes. Toutefois, des protocoles plus complexes sont souvent utilisés pour réduire le trafic d'objets de synchronisation sur le réseau.

Implantation des objets de synchronisation

Deux approches sont souvent utilisées pour l'implantation des opérations de synchronisation: l'implantation par échange de messages et l'implantation par variables partagées.

Les variables partagées semblent être le choix naturel pour l'implantation des objets de synchronisation. Le système à mémoire partagée reste simple et élégant car il est aussi utilisé pour implanter ses propres primitives [LH89]. Néanmoins, ce type d'implantation ne prend pas en compte le principe de la localité, une propriété fondamentale à plusieurs modèle de programmation. Les accès aux variables de synchronisation respectent rarement ce principe. Par conséquent, la plupart des implantations de ce type sont très peu performantes, générant des grands surcoûts.

Bien que moins élégant, l'échange de messages est souvent préféré pour l'implantation des variables de synchronisation. Dans cette approche, les accès aux variables de synchronisation sont en réalité traduits en appels au système, qui se charge de gérer la synchronisation.

2.4 Gestion des pages

La plupart des systèmes à mémoire virtuelle partagée ont recours à la pagination. Ce mécanisme divise le programme en unités de même taille appelées pages. A la limite, l'exécution d'un programme est possible dès que la page couramment référencée est en mémoire. Ceci permet l'exécution de programmes dont la taille est plus grande que celle de la mémoire principale.

Néanmoins, le rajout de cette fonctionnalité entraîne des coûts additionnels au système. D'abord, à chaque accès à la mémoire on doit établir la correspondance entre l'adresse virtuelle (générée par le compilateur) et l'adresse physique (la position de la mémoire physique où la donnée se trouve). Ceci est réalisé par un dispositif matériel appelé MMU[1].

Le partage des données dans un environnement paginé devient aussi plus complexe. Etant l'unité de transfert, la page est souvent choisie aussi comme l'unité de partage. Dans ce contexte, le problème du faux partage y est rajouté.

Dans les paragraphes qui suivent, nous détaillons le mécanisme de gestion des pages ainsi que les solutions couramment adoptées pour résoudre le faux partage.

2.4.1 Le gestionnaire des pages

Dans un système parallèle à mémoire virtuelle partagée, la localisation physique d'une page change souvent au cours du temps. Toutefois, le système doit quand même être capable de retrouver une page pour laquelle il existe une requête en cours. Le gestionnaire de la page est l'entité chargée de garder la trace de la localisation de la page dans le système. Un gestionnaire est souvent associé

1. MMU - Memory Management Unit

à chaque page. La fonction principale de ce gestionnaire est de gérer les informations concernant la page comme les permissions d'accès, l'usage et la localisation. En d'autres termes, c'est le gestionnaire de la page p qui maintient l'entrée de la table de pages correspondant à la page p.

La structure du gestionnaire des pages peut être centralisée, distribuée fixe ou distribuée dynamique [LH89].

Gestionnaire centralisé

Dans l'approche centralisée, il n'existe qu'un gestionnaire pour toutes les pages et il se trouve sur un nœud unique. Chaque fois qu'un processeur désire connaître la localisation d'une page, il envoie un message au gestionnaire.

Selon le type de l'implantation, le gestionnaire peut répondre par l'identité du processeur qui détient la page ou bien envoyer la requête de la page au processeur qui la possède. La dernière approche n'utilise que 3 messages pour le chargement d'une page tandis que la première en utilise 4. De toute façon, ce style de gestionnaire est rarement utilisé. Même pour un nombre moyennement important de requêtes, il devient vite surchargé.

Gestionnaire distribué fixe

La manière la plus simple de distribuer la gestion des pages est d'attribuer statiquement la gestion d'un sous-ensemble des pages à chaque processeur. Cette technique, connue sous le nom de gestion distribuée fixe, est souvent employée pour la gestion des pages. Le choix des pages gérées par un processeur peut être fait par des techniques très simples comme "l'affectation modulo N" ou par des techniques de hachage plus élaborées.

Quand un défaut de page sur la page p se produit, le processeur envoie la requête au processeur $G(p)$. Ayant reçu la requête, le gestionnaire $G(p)$ se comporte comme dans le cas centralisé. Dans quelques systèmes, la fonction de hachage utilisée pour obtenir le gestionnaire $G(p)$ peut être fournie par l'utilisateur.

Gestionnaire distribué dynamique

Dans l'approche distribuée dynamique, le gestionnaire de la page est un des processeurs qui possèdent une de ses copies. La localisation du nœud gestionnaire est donnée par un champ dans un tableau local à chaque processeur. Comme ce champ ne nous donne qu'une notion approximative du gestionnaire de la page, il est appelé "propriétaire probable" (*ProbOwner*). Quand un processeur est en défaut de page, il envoie un message au processeur *ProbOwner*. Si ce

même processeur est le gestionnaire, l'algorithme se comporte comme dans le cas d'un gestionnaire centralisé. Sinon, la requête de la page est envoyée au processeur indiqué sur son propre champ *ProbOwner*. A la fin de la chaîne *ProbOwner(ProbOwner(ProbOwner(...)))*, nous retrouvons le vrai gestionnaire de la page. Selon [LH89], le gestionnaire distribué dynamique présente les meilleures performances. Harold Castro propose dans [Cas95] une variation de l'algorithme distribué dynamique qui profite des méthodes de routage pour retrouver le gestionnaire de la page. L'algorithme proposé est prouvé correct et extensible.

2.4.2 Table de Pages

Les tables de page sont des structures de données qui maintiennent la correspondance entre l'adresse virtuelle et l'adresse physique. De plus, elles stockent des informations de contrôle d'accès et des bits d'états des pages. Les tables de pages sont souvent organisées selon la "projection directe" (*forward mapped*) ou selon la "projection inverse" (*inversed*).

Projection directe

Les tables projetées directement utilisent en général des bits de l'adresse virtuelle pour accéder à une hiérarchie de tableaux. Le niveau le plus bas de cette hiérarchie contient l'entrée associée à l'adresse virtuelle. La figure 2.14 montre ce type d'organisation.

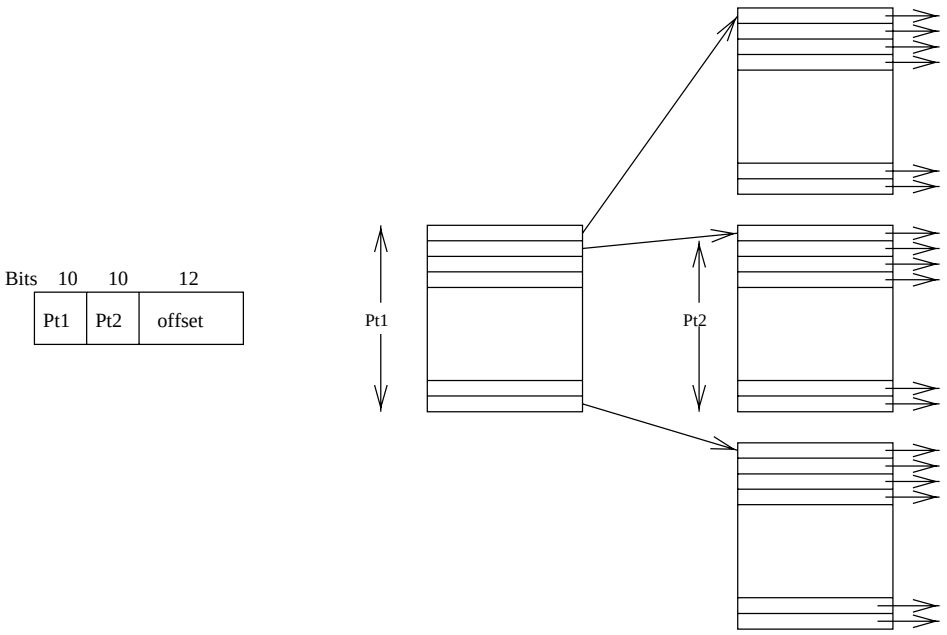


FIG. 2.14 – *Table de pages directement projeté*

Dans la figure 2.14, les dix bits plus significatifs de l'adresse virtuelle (PT1) sont utilisés pour accéder un tableau. Ce tableau contient en fait l'adresse de base d'un autre tableau. Cette adresse de base associée aux dix bits suivants (PT2) donnent le numéro de la page dans laquelle la donnée réside. Les bits moins significatifs de l'adresse virtuelle sont le déplacement de la donnée dans cette page. Cette figure illustre une organisation à deux niveaux.

La taille d'une table directement projetée dépend de la taille de la mémoire virtuelle allouée aux processus. Ainsi, le nombre de niveaux doit augmenter quand la taille de l'espace d'adressage croit. Par ailleurs, il existe une table de pages par processus. D'après les considérations faites ci-dessus, nous pouvons noter que cette organisation, bien que très flexible, n'est pas adaptée aux architectures à 64 bits. Il leur faut de trop nombreaux niveaux pour bien la gérer ([JH93] estime qu'il faut cinq niveaux).

Afin de résoudre ce problème, les tables inversées ont été proposées.

Table de page inversée

Au contraire des tables directement projetées, la table inversée est organisée par adresse physique. L'obtention d'une adresse virtuelle à partir d'une adresse physique est immédiate. Néanmoins, c'est la correspondance inverse qui nous intéresse. Pour la déterminer, nous avons recours à une table de hachage auxiliaire (HAT) accédée à travers une fonction qui prend comme argument l'adresse physique. L'organisation inversée est illustrée par la figure 2.15.

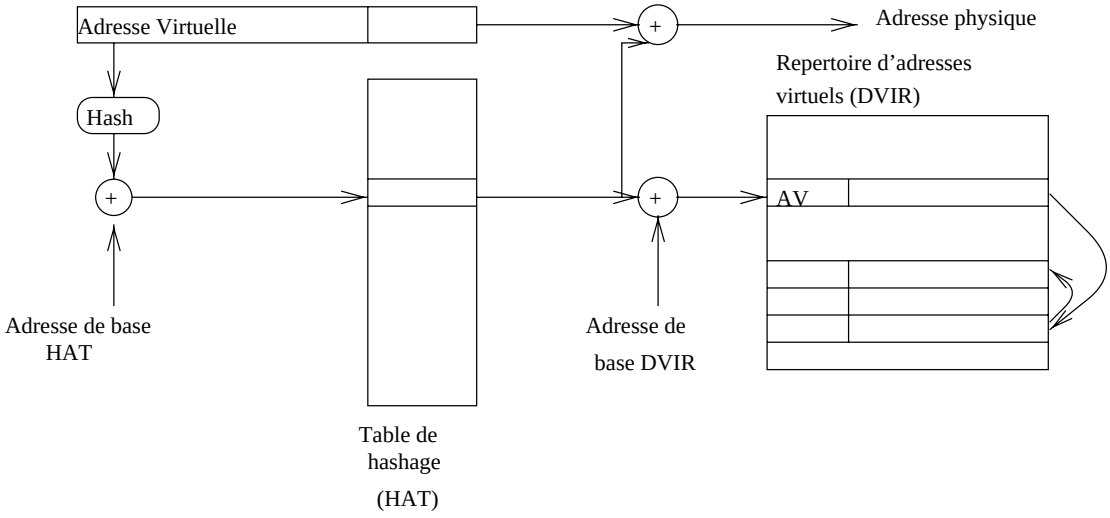


FIG. 2.15 – *Table de pages inversée*

Dans la figure 2.15, une fonction de hachage est appliquée à l'adresse virtuelle. Le résultat de cette opération associé à l'adresse de base de la table de hachage

(HAT) donnent un numéro de page physique. Ce numéro de page est l'indice du répertoire d'adresses virtuelles (DVIR), qui est en fait une table de pages organisée par adresse physique. La comparaison entre l'adresse virtuelle contenue dans le DVIR et l'adresse virtuelle originelle détermine si la page trouvée est bien celle qu'on recherche. Si les adresses ne sont pas les mêmes, une recherche est faite dans une liste d'entrées en conflit. De façon similaire au cas précedent, les bits moins significatifs sont le déplacement dans la page.

Dans le cas des tables de page inhversées, il existe une table de page par processeur et cette table est partagée par tous les processus qui s'y exécutent. Cette organisation est utilisée en général sur les processeurs qui offrent un très grand espace d'adressage car la taille de la table des pages ne dépend que de la taille de la mémoire physique.

Les informations concernant la page

La fonction principale d'une table de pages est de réaliser la correspondance entre l'adresse virtuelle de la page et son adresse physique. Un des champs contenus dans une entrée de la table de pages est alors son adresse physique.

Néanmoins, il existe plusieurs autres informations associées à une page. En dehors de l'adresse physique, les champs les plus courants dans une table de pages sont: les droits d'accès, le verrou, l'indicateur de référence, l'indicateur de modification et l'indicateur de blocage en mémoire, l'indicateur de "cacheable".

L'utilisation de la mémoire virtuelle partagée a rajouté au moins un champ à l'ensemble d'informations associées aux pages. Il s'agit de l'ensemble de processeurs qui détiennent une copie de la page. Ce champ est maintenu dans une structure de données annexe. La structure du champ "ensemble de copies" est détaillée ci-après.

L'ensemble de copies - A un instant donnée, plusieurs copies d'une même page peuvent se trouver sur des nœuds distincts. Dans le pire des cas, des copies d'une même page peuvent être placées sur tous les nœuds du système. Afin de garder la trace d'une page, un champ composé, l'ensemble de copies, est souvent utilisé. L'ensemble de copies contient les processeurs qui possèdent une copie de la page dans leur mémoire locale. Il existe un ensemble de copies par page et il peut être implanté de plusieurs façons [LH89]:

- **vecteur de bits**: si le processeur détient la page, le bit correspondant est mis à 1. Sinon, le bit reste 0. Cette implantation, bien que simple, n'est pas extensible car la taille de l'ensemble de copies dépend du nombre de processeurs qui composent le système. Ce schéma est aussi utilisé dans les

protocoles de gestion du cache sur le nom de répertoire totalement projeté ("fully mapped directory").

- **liste chaînée**: l'ensemble de copies est représenté par une liste chaînée des processeurs qui détiennent une copie de la page. Au contraire de la première approche, la taille de l'ensemble de copies dépend cette fois-ci du nombre de processeurs qui partagent la donnée et non du nombre total de processeurs du système. Bien que l'approche de liste chaînée soit extensible, les opérations sur les copies sont exécutées de manière séquentielle. Le SCI[2] [DVJ90] est un exemple de protocole de cohérence du cache qui se sert des listes doublement chaînées pour maintenir l'ensemble des copies.

- **vecteur de bits des voisins**: dans cette approche, le *copyset* ne tient compte que des nœuds physiquement connectés (voisins directs) qui possèdent la donnée. La taille du *copyset* est constant et égale au nombre de voisins directs. Comme le cas précédent, cette approche est extensible. Néanmoins, une opération d'invalidation ou mise à jour doit être faite par propagation.

2.4.3 Placement des pages

Résoudre le problème du placement de pages consiste à déterminer la meilleure manière d'allouer les pages aux processeurs. Cette distribution a une influence sur le temps d'exécution d'un programme. Si les pages référencées par un processeur se trouvent sur un nœud éloigné, l'opération de chargement devient coûteuse en trafic sur le réseau aussi bien qu'en temps.

Dans les systèmes monoprocesseur, où le mouvement de données se fait entre la mémoire et le disque, les pages sont initialement placées sur le disque et chargées en mémoire à la demande. Dans un environnement multiprocesseur, où les mémoires distantes peuvent être utilisées, des méthodes alternatives ont été étudiées.

Un schéma générique de distribution de pages attribue les pages aux nœuds de façon circulaire (p mod n). Si le modèle de programmation en question est le SPMD[3], une copie de la première page de données du programme peut être placée sur tous les processeurs afin d'éviter des défauts de page à l'occasion des premiers accès.

Deux méthodes d'allocation plus performantes ont été proposées par Clancey et al. dans [CF90]. Le premier fait la division des nœuds du système en groupes et

2. SCI- Scalable Coherent Interface
3. SPMD - Single Program Multiple Data

distribue une copie complète du programme à chaque groupe. Bien que cette méthode réduise la distance moyenne entre les nœuds qui participent au traitement du défaut de page, son coût en mémoire est très élevé.

Une seconde méthode consiste à déterminer les données qui présentent une haute fréquence de défauts de page à l'aide du compilateur ou des exécutions précédentes du même programme. Ces données sont alors dupliquées sur chaque nœud. Cette méthode présente un coût en mémoire encore élevé bien qu'inférieur à celui de la méthode précédente. De plus, l'exactitude de l'estimation des données fréquemment utilisées dépend soit du compilateur soit des exécutions précédentes et, dans le cas le plus général, cette estimation est très conservative.

2.4.4 Chargement des pages

Le problème du chargement des pages consiste à déterminer le moment où une page sera chargée en mémoire. Deux stratégies sont souvent utilisées: le chargement à la demande et le préchargement.

La première stratégie effectue le chargement d'une page en mémoire au moment de la référence à son contenu. Cette technique est appelée "pagination à la demande" car les pages sont chargées à la demande et non à l'avance. Plusieurs remarques peuvent être faites à propos d'une telle politique.

D'abord, le chargement n'est effectué que pour les pages référencées. Ceci évite le gaspillage du temps d'exécution aussi bien que d'occupation mémoire car les pages non-référencées ne sont pas chargées. Néanmoins, à chaque opération de chargement, il faut interrompre l'exécution du processus pendant un délai considérable.

Si toutes les pages d'un processus sont référencées, alors il est préférable de charger toutes les pages en mémoire avant le début de l'exécution. Heureusement, ceci n'est pas le cas typique et le chargement à la demande donne fréquemment des très bons résultats.

La deuxième approche est le préchargement des pages. Elle consiste à déterminer les pages qui seront utilisées dans un futur proche par le processus et les charger en mémoire avant qu'une référence à celles-ci ne se soit produite [LE91].

L'objectif d'un module de préchargement des pages est de réduire le délai causé par les défauts de page et, par conséquent, de réduire le temps d'exécution de l'application.

Toutefois, le coût d'une opération de préchargement est important car il comprend une opération de chargement de page en mémoire locale. Des mauvaises décisions de préchargement entraînent des opérations inutiles de chargement de

pages. Ainsi, le réseau d'interconnexion et les mémoires locales des nœuds peuvent devenir surchargées avec des pages qui ne seront jamais référencées.

A cause de son coût élevé, le mécanisme de préchargement de pages doit être conçu de manière à maintenir petite la probabilité d'occurrence des mauvaises décisions.

L'efficacité d'une stratégie de préchargement dépend de la pertinence de l'estimation des références futures à la mémoire. Pour l'améliorer, quelques systèmes parallèles utilisent des annotations fournies soit par le programmeur soit par le compilateur. D'autres exécutent le programme au préalable pour avoir une trace des références générées et l'utiliser pour prévoir son comportement lors de ses prochaines exécutions [SC93].

2.4.5 Remplacement des pages

Il n'existe pas de solution idéale pour le remplacement de pages car un tel algorithme nécessite de connaître les références futures, ce qui est impossible dans le cas le plus général. Parmi les algorithmes optimaux proposés pour le remplacement des pages, le plus connu est celui de Belady [Bel66]. Bien qu'optimal, cet algorithme est dit non réalisable, car il nécessite de la chaîne complète des références aux pages avant le début de l'exécution du programme.

Pour le cas des monoprocesseurs, les algorithmes de remplacement de pages ont été beaucoup étudiés tant du point de vue théorique que pratique [Tan87]. La grande majorité des algorithmes réalisables proposés tire profit du principe de la localité et considèrent qu'une page peu référencée récemment a une petite probabilité d'être référencée dans un futur proche.

Dans un système à mémoire virtuelle partagée, un niveau intermédiaire est introduit entre la mémoire locale et le disque. Ce niveau compreend les mémoires de tous les nœuds à distance. En plus de choisir la page à remplacer, nous devons donc déterminer où la mettre.

Nous présentons dans ce paragraphe les solutions proposées dans la littérature pour résoudre ces deux problèmes: le choix de la page à remplacer et le choix de la nouvelle localisation de cette page.

Le choix de la page à remplacer

En général, la page choisie pour libérer de la place est celle qui a été le moins récemment utilisée (LRU) [Den70]. La plupart des systèmes monoprocesseurs implantent des variations de cet algorithme. Quand plusieurs processus partagent la mémoire, l'algorithme LRU peut considérer soit les pages du processus en exé-

cution (politique locale) soit toutes les pages en mémoire (politique globale). De façon générale, les algorithmes globaux donnent des meilleurs résultats [Tan87].

Le choix d'une page modifiée entraîne des opérations de mise à jour de cette page sur le disque. Evidemment, le coût de cette décision est plus élevé que celui du choix d'une page inaltérée. Néanmoins, les algorithmes de remplacement proposés pour les architectures monoprocesseur ne prennent pas ce coût en considération et aucune distinction n'est réalisée entre les deux types de page.

Au contraire, la plupart des algorithmes proposés pour les architectures parallèles prennent en compte les types de pages. Ils divisent les cadres de pages en catégories, attribuent des priorités à chaque catégorie et exécutent l'algorithme LRU pour chaque catégorie de cadres [LS89] [LP92]. Il faut noter qu'une page en lecture peut avoir été modifiée. Dans un moment passé, elle a pu avoir la permission d'écriture qui lui a été postérieurement retirée.

Le tableau ci-dessous nous montre un exemple d'attribution de priorités par catégories de pages qui a été utilisé dans Shiva [LS89]:

catégorie de la page	priorité
écriture	1
lecture-prop[4]	2
lecture	3
libre	4

Les catégories de priorité plus élevée sont cherchées d'abord.

La nouvelle localisation de la page

Une fois la page à supprimer de la mémoire locale choisie, il faut déterminer sa nouvelle localisation. La plupart des algorithmes proposés envoient la page à remplacer au disque, s'il y a eu des modifications. Toutefois, la plupart des architectures parallèles comportent des réseaux d'interconnexion point-à-point et un transfert vers le disque nécessite en général de traverser plusieurs nœuds intermédiaires. Si une mémoire à distance a de la place disponible, l'envoi de la page à cette mémoire est souvent une opération moins coûteuse que l'écriture de la page sur le disque.

Ce problème est illustré par la figure 2.16. Sur un réseau d'interconnexion en grille, une page modifiée sur le nœud 12 est choisie par l'algorithme de remplacement. Le coût de cette opération est de six transferts entre nœuds et un transfert

4. lecture-prop: la page est chargée en lecture dans le nœd gestionnaire

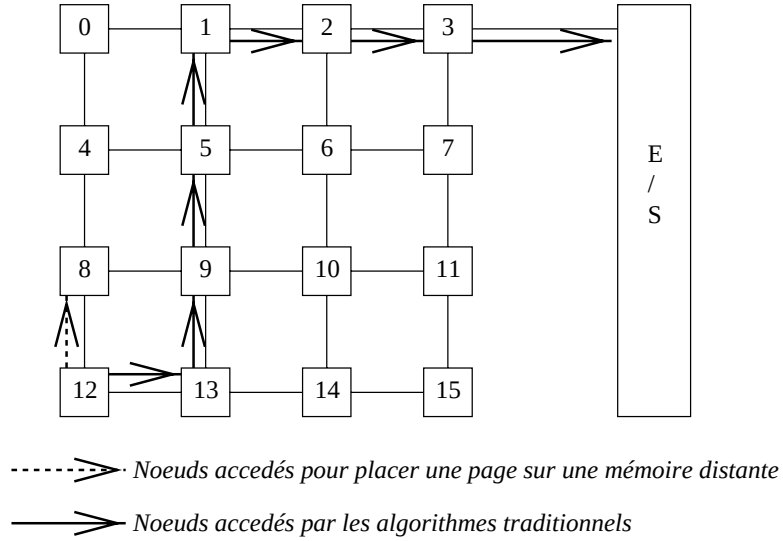


FIG. 2.16 – *Algorithmes de remplacement de pages*

vers le système d'entrée/sortie. Sur les systèmes où le remplacement est fréquent, cette solution crée un trafic important sur le réseau d'interconnexion et fait que le système de E/S devienne rapidement un goulot d'étranglement.

Afin d'éviter ces problèmes, nous pouvons placer la page en question sur un nœud qui a de la place libre dans sa mémoire. Dans le futur, la page sera écrite sur le disque, mais cette opération est retardée.

Le cœur des algorithmes qui placent les pages sur les nœuds à distance est la façon de choisir le nœud qui a de la place libre dans sa mémoire. Ce problème ressemble beaucoup au problème de placement des processus sur les processeurs, qui est NP-complet. Comme celui-ci, le placement des pages sur les nœuds n'a pas de solution optimale.

La nature dynamique de la mémoire rend l'évolution de son taux d'occupation imprévisible. Par exemple, une mémoire peut avoir beaucoup d'espace libre à un moment donné et ne plus en avoir du tout à la suite d'une opération d'allocation dynamique à l'instant suivant. De même, une grande partie de la mémoire peut être libérée d'un moment à l'autre par la terminaison de l'exécution d'un processus.

Dans un système parallèle, la réception d'un message contenant l'occupation $oc(m)$ d'une mémoire veut simplement dire que l'occupation de la mémoire m était $oc(m)$ dans un passé proche. Il n'est pas sûr que l'occupation $oc(m)$ n'ait pas changé entre l'envoi du message et sa réception.

Ainsi, la décision de placer une page dans une mémoire est toujours prise sur des informations partielles à propos de l'état des mémoires à distance. Il est

possible, alors, d'envoyer une page à un nœud dont la mémoire est entièrement occupée. L'algorithme de remplacement doit prévoir ce qu'il faut faire dans le cas où cette situation se produit.

Une autre solution consiste à réserver la mémoire vive de quelques nœuds au stockage des pages virtuelles. La page à remplacer est alors envoyée à un de ces nœuds, appelés *serveurs de mémoire* [CG92] [Cas95].

Cette approche ne présente pas les inconvénients de l'approche précédente puisqu'on fait souvent l'hypothèse que la détermination de la nouvelle localisation de la page ne dépend pas de l'état d'occupation de la mémoire des serveurs de mémoire. Cependant, des goulots d'étranglement peuvent se produire sur le réseau d'interconnexion si l'activité de remplacement est fréquente.

Une autre question importante est celle du coût de l'algorithme. En évaluant le coût par le nombre de messages échangés entre les nœuds, nous avons la formule suivante: $c_{remplacement} = c_{direct} + c_{indirect}$. Le coût direct c_{direct} correspond au nombre de messages échangés au moment de l'exécution de l'algorithme. Le coût c_{direct} est égal à 0 si une page inaltérée est choisie. Sinon, il est au moins égal à 2 (envoi de la page + acquittement)[5]. Le coût indirect $c_{indirect}$ est fonction du nombre de messages échangés pour maintenir les informations concernant l'occupation des mémoires à distance.

Comme l'algorithme de remplacement est en général exécuté à la suite d'un défaut de page, son exécution fait croître le temps nécessaire pour récuperer la page. Afin de réduire la probabilité d'exécution de l'algorithme de remplacement au moment du défaut de page, nous avons souvent recours aux démons. Le démon de remplacement est un processus qui s'exécute en arrière plan. Il est réveillé dès que l'occupation de la mémoire atteint un seuil pré-défini et il remplace des pages jusqu'à ce que l'occupation de la mémoire soit de nouveau au-dessous de ce seuil [LS89].

2.4.6 Faux partage

Le partage des données entraîne des problèmes de cohérence. Un problème de "vrai partage" existe quand plusieurs processeurs accèdent et modifient la même adresse partagée simultanément. Néanmoins, deux variables indépendantes peuvent être placées sur la même page. La page étant l'unité d'accès, les protocoles de cohérence sont appliqués même si la donnée référencée n'est pas la même. Ce problème est connu sous le nom de "faux partage" et est un phénomène qui se produit quand l'unité de partage est différente de l'unité d'accès.

5. Normalement, il faudrait considérer la taille des messages mais, pour simplifier, nous ne considérons que leur nombre.

Sur les machines parallèles, le problème du faux partage s'est aggravé et, dans quelques programmes, il cause jusqu'à 40% des défauts de page [KJe91]. Afin de le résoudre, deux approches sont fréquemment utilisées.

La première consiste à restructurer le code du programme de manière à placer les données accédées simultanément sur des pages différentes. L'inconvénient de cette approche repose sur le besoin d'informations supplémentaires apportées soit par le compilateur soit par le programmeur lui-même.

La deuxième approche, adoptée le plus souvent, consiste à permettre plusieurs écritures simultanées sur la même page (protocole MRMW[6]). Le problème du faux partage cesse d'exister car la restriction d'un seul processeur en écriture à la fois est relâchée. Puisqu'il existe plusieurs versions d'une même page dans le système, le chargement d'une page sur un nœud dévient une opération plus complexe. Il faut, maintenant, effectuer une "opération $\oplus$ (XOR)" sur toutes les copies existantes avant de rendre la page au processeur qui la demande. Il est nécessaire aussi de garantir que la même donnée n'est pas accédée simultanément. Ceci est réalisé par les primitives de synchronisation [BH90] [Car93].

2.4.7 Taille de la page

La taille de la page est un paramètre fixé par le matériel qui dépend de plusieurs facteurs en conflit. Le problème du faux partage et celui de la fragmentation interne[7] sont réduits quand les pages ont une petite taille. Mais dans ce cas il existe l'augmentation de la taille de la table de pages et du trafic d'informations sur le réseau. Déterminer la bonne taille de la page c'est trouver un compromis entre ces facteurs.

En utilisant le critère de minimisation des pertes en stockage, la taille optimale de la page est donnée par une formule qui fait la relation entre la taille moyenne du programme et la taille d'une entrée de la table de pages [Tan92].

Toutefois, la manière dont les informations sont accédées joue un rôle important sur la définition de la taille des pages. Pour cette raison, plusieurs systèmes ont proposé des tailles de page adaptatives composées par des pages qui sont accédées selon la taille la plus adaptée aux besoins de l'application [Hol89].

6. MRMW - Multiple Readers Multiple Writers
7. fragmentation interne - remplissage partielle de la dernière page du processus

2.5 Niveau d'implantation

La mémoire virtuelle partagée peut être réalisée directement par le matériel. En principe, un tel choix rend cette fonctionnalité incontestablement performante. Néanmoins, les systèmes résultants sont peu flexibles et leur portage est impossible. DASH [LLJ+93], FLASH [Ka94] et PLUS [BR90] sont des exemples de systèmes à MVP implantés par le matériel.

La MVP peut aussi être implantée dans le système d'exploitation. Les arguments en faveur d'un tel choix sont nombreux. Tout en conservant les hautes performances, une implantation au niveau système est plus flexible que celle au niveau matériel. De plus, une telle implantation est robuste par rapport aux processus utilisateurs car elle se sert des mécanismes de protection du système d'exploitation.

Cependant, des erreurs de programmation à l'intérieur de la mémoire virtuelle partagée peuvent nuire à l'exécution du système d'exploitation. Son portage s'avère aussi une tâche ardue: le système d'exploitation original doit être entièrement remplacé par la version qui implante la mémoire virtuelle partagée. De plus, cette approche va à l'encontre de la tendance des systèmes d'exploitation modernes qui tentent de déplacer les fonctions dans les couches supérieures pour que le noyau reste minimal. KOAN [Lah93] et Mirage [FP89] sont des exemples de systèmes à mémoire virtuelle partagée implantés au niveau du système d'exploitation.

Un système à mémoire virtuelle partagée peut aussi être implanté comme un serveur. Un serveur est un processus utilisateur autonome dont le rôle consiste à répondre aux requêtes qui lui sont adressées [Tan92]. Cette approche n'apporte pas de modifications au système d'exploitation et les systèmes résultants sont potentiellement portables. Evidemment, les performances sont plus basses que celles des approches précédentes. Un troisième processus - le serveur - intervient dans le dialogue autrefois exclusif entre le processus utilisateur et le noyau système. MYOAN [CPP94] est un exemple de serveur de mémoire virtuelle partagée.

Un système à MVP peut être aussi un environnement de programmation, avec des extensions de langage et des bibliothèques runtime. Ces environnements de programmation sont parfois très complets et offrent aussi des outils de mise au point et monitoring. Afin de traiter les extensions de langage créées, des modifications dans le compilateur sont souvent nécessaires. Munin [Car93] et Midway [BZ91b] sont des exemples de systèmes à mémoire virtuelle partagée implantés comme des environnements de programmation.

2.6 Exemples de systèmes à mémoire virtuelle partagée

Dans ce paragraphe, nous présentons à un niveau général les principaux choix de conception de trois systèmes à mémoire virtuelle partagée: Ivy, Munin et Midway.

Ivy a été choisi car c'est le premier système à MVP proposé et nous montre la mémoire virtuelle partagée telle qu'elle a été conçue initialement. Munin est un système à mémoire virtuelle partagée qui se sert d'un modèle de cohérence hybride et utilise plusieurs protocoles de cohérence. Midway est un des rares systèmes qui permettent plusieurs modèles de cohérence de la mémoire.

A la fin du chapitre, nous présentons un tableau comparatif des choix faits par plusieurs autres systèmes à mémoire virtuelle partagée.

2.6.1 Ivy

Ivy a été proposé par Li dans [Li86] à Yale University. C'est le premier système à utiliser le concept de mémoire virtuelle partagée. Un programme Ivy est un ensemble de processus légers qui partagent un espace d'adressage unique divisé en pages. Les processus légers sont placés sur plusieurs processeurs et la migration est possible.

Les mémoires locales à chaque processeur sont vues comme des caches de l'espace d'adressage. Une référence à une page distante génère un défaut de page. Les défauts de page en écriture déclenchent aussi l'exécution des mécanismes qui assurent un type de cohérence forte de la mémoire, la cohérence séquentielle.

L'implantation d'Ivy a été réalisée au niveau système et les processus y accèdent à travers des primitives. Chaque processeur exécute un serveur Ivy composé de cinq modules, comme le montre la figure 2.17.

Le module de projection a deux fonctions principales: retrouver la page à la suite d'un défaut de page et garantir la cohérence entre les copies. La recherche de la page est effectuée à l'aide d'un algorithme centralisé, distribué fixe ou distribué dynamique. Ces trois algorithmes, déjà décrits dans le paragraphe 2.4.1, ont été proposés et implantés par Ivy. La cohérence forte des pages est garantie à l'aide d'un protocole MRSW[8] d'invalidation.

Une table de pages simplifiée a été implantée. La structure directement projetée est utilisée et le tableau est entièrement stocké en mémoire.

8. MRSW - Multiple Readers Single Writer

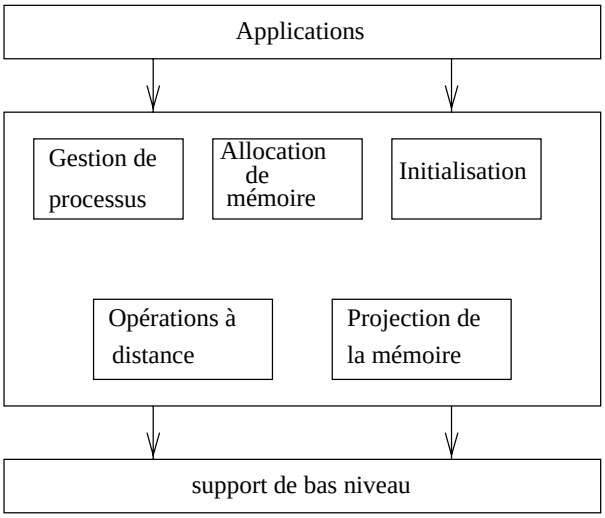


FIG. 2.17 – *L'organisation d'Ivy*

Le module de gestion de processus implante des opérations de contrôle de processus, migration et synchronisation. Une primitive est offerte aux utilisateurs pour signaler à Ivy si un processus a ou non le droit de migrer. A cause de la mémoire virtuelle partagée, le mécanisme de migration est très simple. Il consiste à transférer les informations de contrôle du processus (PCB[9]) et la page sur laquelle le processus s'exécute.

Le mécanisme de synchronisation utilisé par Ivy est le "compteur d'événements" [Li88]. Son implantation utilise la mémoire virtuelle partagée.

L'allocation dynamique de la mémoire est réalisée à l'aide d'un algorithme *first fit* [Tan92]. Le module d'opérations à distance implante un mécanisme d'appel à procédures à distance.

Un prototype d'Ivy a été implanté sur un réseau Apollo [Li88]. Sous le nom de Shiva [LS89], Ivy a été porté sur un hypercube Intel iPSC/2 avec 128 nœuds. Ce système implante un mécanisme de remplacement de pages qui se sert des priorités. La priorité d'une page est calculée selon la formule 2.1:

$$prio(p) = type\_page(p)\alpha + t(1-\alpha) \tag{2.1}$$

9. PCB - Processus Control Block

où: $type\_page(p)$ est le type de la page; t est le temps pendant lequel la page n'a pas été référencée; et α est un paramètre dynamique.

La page qui possède la plus grande priorité $prio(p)$ est remplacée. L'algorithme de remplacement est exécuté par un démon qui se met en marche dès que la mémoire est surchargée [LS89].

Le mécanisme de synchronisation offert par Shiva est le sémaphore. Les primitives P et V sont implantées à l'aide de l'échange de messages [LS89]. Les sémaphores sont gérés selon la stratégie dynamique distribuée.

2.6.2 Munin

Munin est un système à mémoire virtuelle partagée conçu à Rice University [BCZ91] [Car93]. Il utilise un type relâché de cohérence, la cohérence à la libération. Les écritures sont stockées localement. Au moment de la libération d'un verrou, toutes les autres copies sont mises à jour.

Selon la philosophie de Munin, il n'existe pas un protocole de cohérence qui offre des bonnes performances à toutes les applications. En effet, le choix du protocole de cohérence dépend de la manière dont les variables sont accédées. Ainsi, Munin propose à l'utilisateur de fournir des informations additionnelles dans le but de l'aider à choisir le protocole le plus adapté à une variable partagée. Les protocoles proposés dans [Car93] sont: conventionnel, migration, lecture seulement et partagé en écriture. Pour les variables partagées en écriture, Munin applique un protocole de mise à jour à écrivains multiples (MRMW).

Un programme Munin est un ensemble de processus légers. La migration des processus et l'allocation dynamique de la mémoire ne sont pas supportées. Le partage est fait au niveau des variables. Une variable partagée doit être annotée comme telle. A priori, chaque variable partagée est placée sur une page distincte.

Pour la synchronisation entre les processus, Munin offre les verrous, les barrières de synchronisation et les variables condition. L'implantation de ces mécanismes est réalisée par échange de messages.

Munin implante des verrous bloquants. Une file distribuée contient les processus en attente du verrou. Les barrières de synchronisation et les variables de condition sont gérées de façon centralisée.

Un prototype Munin a été implanté sur un réseau de 16 SUN-3/60 sous le système V [Car93]. Son organisation est montrée dans la figure 2.18.

Le répertoire des objets maintient des informations de contrôle des variables partagées. Il existe une entrée par variable. Le runtime Munin est accédé par le

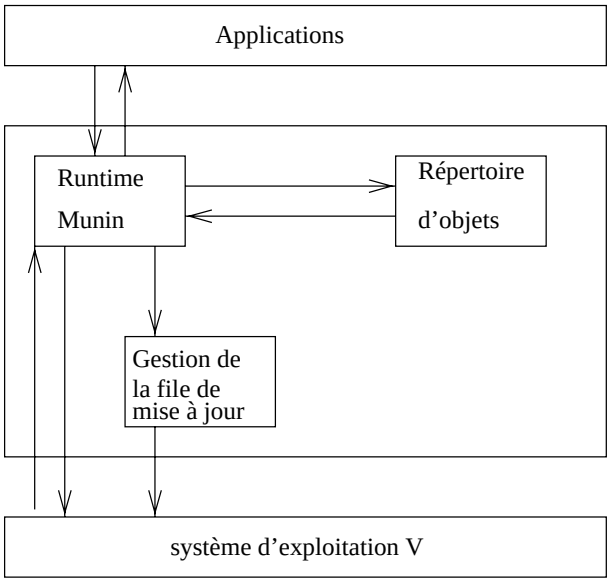


FIG. 2.18 – *L'organisation de Munin*

processus soit directement, à travers des appels aux primitives Munin, soit via le noyau V, par des défauts de page. La file de mise à jour est la structure de données qui stocke les modifications apportées aux variables partagées.

2.6.3 Midway

Midway a été conçu par Bershad et al. [BZS93a] à Carnegie-Mellon University. C'est un environnement de programmation à variables partagées implanté entièrement par logiciel. Il est composé de:

- un ensemble de mot-clés et d'appels au système, nécessaires pour annoter les variables partagées;
- un système runtime qui implante les modèles de cohérence de la mémoire;
- un compilateur qui génère du code pour maintenir des informations de contrôle des variables partagées.

Un programme Midway est un ensemble de processus légers. Les variables de synchronisation et les variables partagées doivent être annotées comme telles. Par l'intermédiaire d'un appel système, une variable partagée peut être associée à la variable de synchronisation qui la garde.

Les mécanismes de synchronisation entre les processus légers offerts par Midway sont les verrous et les barrières de synchronisation. La localisation des verrous est obtenue par un algorithme de files distribuées tandis que les barrières de synchronisation sont gérées à l'aide de l'algorithme distribué fixe.

Midway supporte trois modèles relâchés de cohérence de la mémoire: la cohérence du processeur, la cohérence à la libération et la cohérence d'entrée. La spécification du modèle de cohérence utilisé est faite variable par variable. Les variables associées à un objet de synchronisation se comporteront selon la cohérence d'entrée. Les variables associées à un intervalle de flush suivront la cohérence du processeur, l'intervalle de flush étant le taux de propagation des mises à jour. Toutes les autres variables partagées obéiront à la cohérence à la libération. La cohérence d'entrée est assurée par un mécanisme d'estampillage [BZ91b].

2.6.4 Tableau comparatif des systèmes

L'objectif de ce paragraphe est de présenter les principales caractéristiques de différents systèmes à mémoire virtuelle partagé, y compris ceux qui viennent d'être décrits, à titre comparatif. Le tableau récapitulatif est montré dans la figure 2.19. Les systèmes présentés dans le tableau sont: Ivy [Li88], KOAN [Lah93], Midway [BZS93b], Munin [Car93], Mirage [FP89], TreadMarks [KDCZ93], Mermera [HS93], Galactica/Net [W$^+$93] et CarlOS [KFJ94].

La première colonne contient le nom du système à MVP. La seconde colonne spécifie le schéma d'implantation utilisé. Dans le cas d'un environnement de programmation, l'existence de modifications sur autres entités du système est précisée (compilo = compilateur; SE = système d'exploitation). La troisième colonne donne le modèle ou les modèles de cohérence supportés par le système à MVP. Ensuite, nous précisons l'unité de partage et l'existence de mécanismes de remplacement de pages.

La sixième colonne précise comment le système à MVP est informé d'un accès à une donnée partagée. Dans Midway, le compilateur a été modifié de façon à ce que le programme parallèle écrive sur une structure auxiliaire chaque fois qu'une donnée partagée est mise à jour. Les mécanismes de synchronisation offerts par les différents systèmes sont montrés dans la colonne 7.

La colonne 8 nous donne les universités où des travaux de recherche sur le système à MVP ont été ménés. A la fin, nous montrons les machines sur lesquelles un prototype du système à MVP a été implanté.

Dans le chapitre suivant, nous présentérons nos travaux pour la conception du système $\mathcal{DIVA}$ qui suit une approche originale comparée aux systèmes décrits dans ce tableau. Les caractéristiques et la spécificité de l'approche sont présentés dans les pages suivantes.

SYSTEME	NIVEAU D'IMPLANTATION	MODELE DE COHERENCE	UNITE DE PARTAGE	REMPLACEMENT	ACCES AUX DONNEES	TYPE DE SYNCHRO	UNIVERSITE	ARCHITECTURE CIBLE
Ivy	système d'exploitation	séquentielle	page	oui	défaut de page	compteur d'événements	Princeton	réseau Appolo
Koan	système d'exploitation	séquentielle faible	page	oui	défaut de page	sémaphore	Rennes I	iPSC/2
Midway	environnement prog (compilo)	processeur libération entrée	variables	non	compilateur	verrous barrières	CMU	réseau DEC
Munin	environnement prog(compilo,SE)	libération	page	non	défaut de page	verrous barrières var condition	Rice	réseau SUN
Mirage	système d'exploitation	séquentielle	page	non	défaut de page	___	U. California	réseau VAX
TreadMarks	environnement prog	libération (LRC)	page	non	défaut de page	verrous barrières	Rice	réseau DEC
Mermera	environnement prog	séquentielle processeur lente / locale	variables	non	primitives	___	Boston	reseau SUN
Galactica/Net	matériel + SE	libération	page	non	défaut de page	instruction XMEM	Worcester	Lynx
CarlOS	environnement prog	libération (LRC)	page	non	défaut de page	verrous	Copenhagen	réseau DEC

FIG. 2.19 – *Les différents systèmes à MVP*

Chapitre 3

DIVA: un système à modèles de cohérence multiples

3.1 Motivation

Nous avons vu dans le paragraphe 2.2 que le choix d'un modèle de cohérence de la mémoire est un compromis entre la simplicité de programmation et les performances. En général, plus le modèle de cohérence est faible plus on peut obtenir des performances élevées. Hélas, les gains en performance sont presque toujours accompagnés d'une plus grande complexité du modèle de programmation.

Dans une situation idéale, le modèle de cohérence de la mémoire doit offrir à chaque application exactement les restrictions d'ordre des accès à la mémoire partagée dont elle a besoin pour s'exécuter correctement.

D'une façon intuitive, une exécution correcte d'une application rend au programmeur les résultats qu'il attend de son programme. Les résultats possibles d'une exécution sont définis par le modèle de cohérence de la mémoire. Dans le cas d'une machine monoprocesseur, les résultats qui sont attendus d'un programme sont ceux qui peuvent être générés par le modèle de cohérence séquentielle ou par la cohérence atomique. Pour le moment, nous considérons qu'un programmeur d'une machine parallèle attend que l'ensemble des résultats produits par son programme parallèle soit identique à l'ensemble des résultats produits quand le même programme s'exécute sur une machine monoprocesseur.

Si le modèle de cohérence de la mémoire introduit plus de restrictions d'ordre que celles strictement nécessaires à l'exécution correcte de l'application, les résultats produits seront encore corrects. Néanmoins, plusieurs opérations inutiles seront exécutées pour garantir un ordre qui n'est pas nécessaire. Par conséquent, les performances se dégradent.

Si le modèle de cohérence offre moins de garanties d'ordre d'accès que celles nécessaires à l'application, des résultats "incorrects" peuvent être produits. Pour éviter cette situation, l'application doit être reprogrammée. Cette fois-ci, la programmation de l'application doit aussi considérer les garanties d'ordre d'accès offertes par le modèle. Par conséquent, la programmation se complique.

Les deux situations citées ci-dessus ne sont pas idéales. La première amène à des pertes parfois très importantes en performance. La seconde rend le modèle de programmation plus complexe.

Certains modèles de cohérence, tels que la cohérence à la libération et la cohérence du processeur, ont spécifié des classes d'applications (programmes correctement étiquetés et programmes de calcul sans mémoire[1], respectivement) pour lesquelles les résultats produits dans le modèle de cohérence en question sont exactement les mêmes que ceux produits dans un modèle de cohérence forte [Adv93]. En d'autres termes, les applications qui appartiennent à ces classes peuvent profiter des gains en performances apportés par ces modèles de cohérence relâchés toujours en gardant un modèle de programmation simple, la cohérence forte.

Il existe d'autres classes d'applications, essentiellement parallèles, qui ont abandonné l'exigence d'un comportement séquentiel global. L'ensemble de résultats corrects admis par ces applications comprend les résultats produits par des exécutions séquentielles aussi bien que plusieurs autres résultats. Ces applications béneficient beaucoup des modèles de cohérence plus rélâchés. C'est le cas par exemple des algorithmes de relaxation chaotique [G+90].

Nous pouvons noter alors qu'il existe un grand nombre de classes d'applications et chaque classe a des besoins spécifiques de cohérence.

Les premiers systèmes à mémoire virtuelle partagée ont essayé d'offrir aux programmeurs un modèle de cohérence de la mémoire qui soit à la fois simple et performant pour un ensemble important de classes d'applications. Très vite, les chercheurs ont remarqué que cette tâche était très difficile, sinon impossible. L'analyse des applications parallèles a montré que la plupart des algorithmes ont des caractéristiques particulières de partage de données et que le choix du modèle de cohérence de la mémoire dépend aussi de l'application parallèle en question.

Cette constatation a fait apparaître un nouveau type de système à mémoire virtuelle partagée, les systèmes à modèles multiples de cohérence de la mémoire [HS92] [BZS93a]. Ces systèmes permettent le choix par l'application du modèle de cohérence dans lequel elle s'exécutera. En général, la cohérence séquentielle est offerte aussi bien que quelques modèles plus relâchés. La coexistence entre plusieurs modèles dans une même exécution est aussi possible.

1. programme de calcul sans mémoire ("oblivious computations") - les données utilisées dans une phase de calcul ne dépendent pas des données calculées dans la phase précedente

Le grand nombre de modèles de cohérence de la mémoire existants dans la littérature indiquent que le choix parmi un sous-ensemble de ces modèles est une caractéristique nécessaire au support de cohérence des applications parallèles. Cependant, nous pensons que cette caractéristique, bien que nécessaire, n'est pas suffisante pour deux raisons. En premier, la recherche dans le domaine des modèles de cohérence de la mémoire est en pleine activité. Plusieurs nouveaux modèles vont sûrement encore surgir. Second, l'offre d'un sous-ensemble pré-établi de modèles peut toujours exclure des modèles de cohérence dans lesquels certaines classes d'applications parallèles sont exécutées de manière plus performante et/ou programmées de manière plus simple.

Dans $\mathcal{DIVA}$, nous rajoutons une nouvelle dimension à la gestion de la cohérence des applications parallèles. En plus du choix entre des modèles pré-définis, nous offrons à l'utilisateur la possibilité de rajouter de nouveaux modèles de cohérence à notre système. Ainsi, ce que nous proposons est en fait un modèle d'exécution de base sur lequel plusieurs modèles de cohérence peuvent être bâtis. Nous pensons qu'une telle flexibilité est nécessaire et suffisante pour que les systèmes à mémoire virtuelle partagée supportent de manière efficace une large gamme de classes d'applications.

Ce chapitre décrit les mécanismes de base proposés pour la conception et la réalisation du module de gestion de modèles de cohérence multiples de $\mathcal{DIVA}$.

Le paragraphe 3.2 décrit les choix initiaux des fonctionnalités qui seront offertes par notre module à modèles multiples. Le paragraphe 3.3 décrit l'approche utilisée pour offrir ces fonctionnalités. Notamment, nous décrivons ici le système parallèle PAROS et l'insertion de $\mathcal{DIVA}$ dans ce contexte comme une machine virtuelle.

Ensuite, nous présentons le fonctionnement du module qui gère les modèles de cohérence multiples. Nous détaillons par la suite l'interface d'implantation des nouveaux modèles de cohérence de la mémoire aussi bien que l'interface de choix du modèle de cohérence qui doit être utilisé dans l'exécution de l'application. Nous décrivons aussi l'approche que nous avons adoptée pour gérer la synchronisation.

A la fin du chapitre, nous faisons la comparaison de notre approche avec d'autres travaux de recherche qui ont été menés dans le domaine des modèles de cohérence multiples.

3.2 Applications supportées

Nous proposons l'intégration des modèles multiples de cohérence de la mémoire dans un module générique de $\mathcal{DIVA}$ et ce pour servir à un ensemble important d'applications. Plusieurs mécanismes ont été conçus pour offrir un

support simple et flexible aux différents modèles de cohérence.

Comme notre système doit être utilisé par plusieurs types d'utilisateurs, nous nous sommes efforcés pour garantir que la flexibilité apportée par les mécanismes nécessaires à l'implantation de modèles de cohérence multiples ne soit pas pénalisante pour les utilisateurs qui ne veulent pas se servir de cette fonctionalité.

Nous mettons à disposition du programmeur qui ne veut pas se soucier des modèles de cohérence, un modèle de cohérence de la mémoire par défaut.

Par contre, nous offrons au programmeur qui veut réaliser une implantation spécifique de son algorithme la possibilité d'associer à son application un modèle de cohérence pré-défini. Ceci est fait au début du programme et, une fois l'association faite, elle reste valable jusqu'à la fin de l'exécution.

Un programmeur système est au courant de toutes les potentialités de sa machine et de son logiciel. Néanmoins, ces potentialités ne lui suffisent plus. Il veut profiter davantage des caractéristiques du matériel. Pour y arriver, il crée ses propres protocoles et les programme au plus bas niveau. La possibilité de définir des modèles de cohérence de la mémoire autres que ceux offerts par notre système est accordée aux programmeurs système. Une interface de définition de modèles de cohérence de la mémoire est prévue à cet effet.

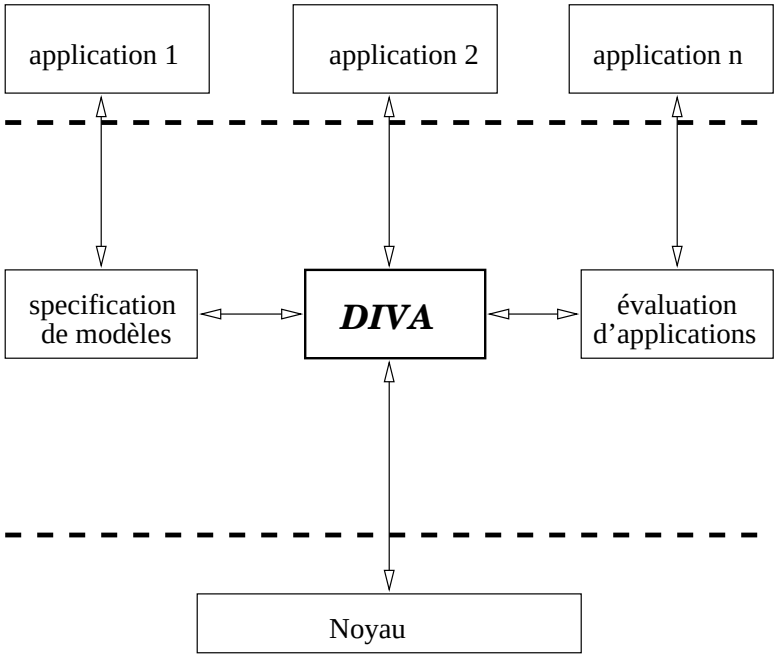


FIG. 3.1 – *Les différentes utilisations de DIVA*

La décision de servir à plusieurs types d'utilisateurs à la fois a conduit à un système à mémoire virtuelle partagée qui peut être utilisé directement par des ap-

plications qui veulent se servir du modèle de programmation à mémoire partagée ainsi que par des outils internes au système tels que des programmes de spécification de modèles de cohérence et des programmes d'évaluation d'applications parallèles (c.f. figure 3.1).

Les premiers se serviront de $\mathcal{DIVA}$ pour générer et évaluer des nouveaux modèles de cohérence de la mémoire. Les programmes d'évaluation d'applications pourront se servir de $\mathcal{DIVA}$ pour évaluer le comportement d'une même application dans différents modèles de cohérence.

Quelques systèmes à mémoire distribuée partagée tels que Mether [HS92], Midway [BZS93a] et KOAN [Lah93] permettent la coexistence de plusieurs modèles de cohérence dans une même exécution d'une application. Cette approche utilise en général des concepts de la théorie des bases de données, où chaque donnée ou ensemble de données est accédé selon des règles spécifiées au préalable.

Bien que rien dans notre module de gestion de modèles de cohérence ne l'interdise, nous avons fait le choix d'offrir un modèle unique pour chaque exécution d'une application. Nous pensons que la gestion de la complexité résultante de la coexistance entre plusieurs modèles pour une même exécution d'une application peut parfois générer des inefficacités. Les risques d'occurrence de ces inefficacités sont plus grands notamment dans les machines parallèles de taille importante.

C'est uniquement pour préserver les performances des applications que nous avons choisi de réaliser l'association entre une application et un modèle de cohérence au début de l'exécution de l'application. De plus, nous avons interdit à l'utilisateur de changer le modèle de cohérence une fois l'association faite.

Néanmoins, notre module de gestion de modèles de cohérence multiples est très flexible. Les utilisateurs qui le souhaitent peuvent définir un modèle de cohérence dont le comportement est dicté par des caractéristiques spécifiques à chaque donnée. Bien que pour $\mathcal{DIVA}$ les données partagées soient encore vues comme une mémoire unique, le nouveau modèle peut traiter les différentes données de façon différenciée.

Pour conclure, nous proposons les modèles multiples de cohérence de la mémoire pour deux raisons. D'abord, l'utilisateur doit être libre pour choisir le modèle de cohérence le plus adapté à son application. Ensuite, puisqu'il n'existe pas un modèle de cohérence global et que plusieurs modèles nouveaux vont encore être proposés, le rajout dynamique de modèles au système est une propriété souhaitable.

3.3 Architecture PAROS

Pour le choix du support système nécessaire à $\mathcal{DIVA}$, nous avons considéré deux critères: la flexibilité et l'extensibilité.

Les mécanismes système doivent être assez flexibles et génériques pour permettre le support par $\mathcal{DIVA}$ de l'ensemble d'applications et d'utilisateurs décrit dans le paragraphe précédent. De plus, le support système doit être extensible.

En observant ces critères, nous avons choisi le système d'exploitation PAROS pour servir de support à $\mathcal{DIVA}$ (voir figure 3.2).

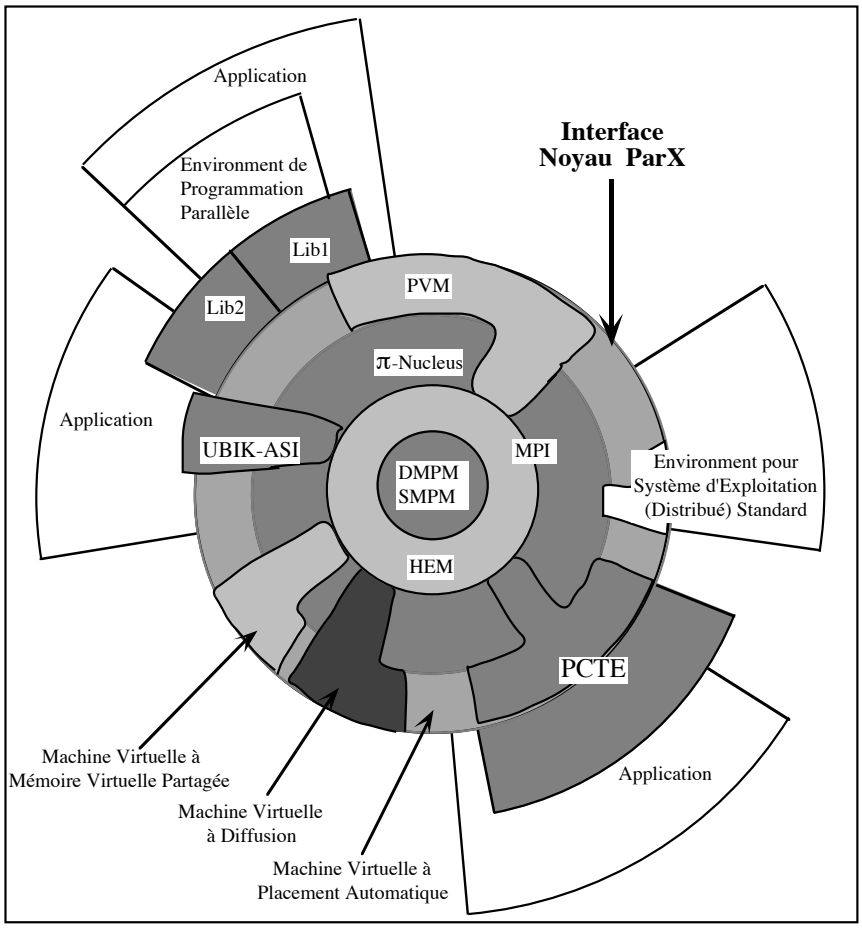


FIG. 3.2 – *L'architecture do système PAROS*

PAROS est un système d'exploitation qui a été conçu pour les machines massivement parallèles dans le cadre du projet ESPRIT Supernode II [ESP91]. Il utilise la notion de couches et de machines virtuelles pour servir à une large gamme d'applications. PAROS offre ainsi tous les services de base pour construire des machines virtuelles capables d'offrir à l'utilisateur plusieurs interfaces et environ-

nements de programmation.

PAROS repose sur le noyau ParX [M+89], qui implante le modèle de processus et celui de la communication. En effet, ParX offre des mécanismes génériques et corrects pour la construction de protocoles. Ces protocoles servent à la réalisation de diverses machines virtuelles (ou niveaux d'abstraction) [Cas95]. Chaque utilisateur peut avoir ainsi le système d'exploitation le mieux adapté aux besoins spécifiques de ses applications.

Au plus bas niveau, nous avons les mécanismes qui permettent la virtualisation des ressources matérielles (HEM). Grâce à des mécanismes de routage extensibles, l'HEM offre l'interface d'un réseau complètement connecté.

Au-dessus de l'HEM, nous avons l'ensemble de machines virtuelles offertes par le noyau ParX. Pour offrir le support à ces différentes machines virtuelles, ParX dispose d'un micro-noyau (π-nucleus) qui s'occupe de la gestion de base des ressources telles que processeurs et mémoires. Jusqu'à maintenant, les machines virtuelles offertes par ParX réalisent la diffusion [DM93], le placement automatique de processus [Tal93] et un modèle de programmation mixte [Gia93]. Le portage des environnements PVM et PCTE en tant que machines virtuelles de ParX est en cours.

Dans ce cadre, $\mathcal{DIVA}$ a été conçue comme une machine virtuelle de ParX qui offre un espace d'adressage unique à modèles de cohérence multiples aux couches supérieures du système PAROS. Dans la figure 3.2, $\mathcal{DIVA}$ est représenté comme une machine virtuelle à mémoire virtuelle partagée.

Au-dessus de ParX sont bâtis des outils tels que des systèmes d'exploitation, ou des environnements de programmation aussi bien que des bibliothèques. Les applications parallèles peuvent aussi se servir directement de ParX.

PAROS et son noyau ParX offrent alors la flexibilité et l'extensibilité nécessaires à notre système à mémoire virtuelle partagée. Les applications et les utilisateurs discutés dans le paragraphe précédent peuvent ainsi être complètement supportés.

Dans la suite de ce chapitre, nous décrivons la gestion des modèles de cohérence dans la machine virtuelle $\mathcal{DIVA}$.

3.4 Gestion des modèles de cohérence

La conception de notre module de gestion des modèles de cohérence a été faite dans le but de concilier deux caractéristiques qui sont souvent en conflit: la flexibilité et les hautes performances.

Le premier critère consiste à garantir que le module de gestion des modèles offre le support nécessaire à l'implantation d'un nombre important de modèles de cohérence. Cette flexibilité est obtenue quand les caractéristiques particulières de chaque modèle ne sont pas prises en compte. Le module doit alors gérer un modèle de cohérence générique. Les particularités de chaque modèle sont additionnées uniquement dans le traitement prévu pour la gestion spécifique du modèle.

De cette manière, nous sommes capables de supporter plusieurs modèles de cohérence existantes ainsi que plusieurs modèles définis ultérieurement. Pour qu'un modèle soit supporté par $\mathcal{DIVA}$, la seule exigence est de suivre le comportement du modèle générique traité par notre module de gestion de modèles.

Le second critère doit garantir que la flexibilité offerte n'est pas trop pénalisante pour les performances du système. En observant ce critère, nous avons écarté l'utilisation de couches intermédiaires et nous avons conçu notre module comme une couche logicielle unique.

3.4.1 Définition du modèle de cohérence générique

Il est très difficile de définir le comportement d'un modèle de cohérence générique. Ceci arrive parce que les modèles de cohérence ont été définis au fur et à mesure que les besoins de performances s'imposaient. Ainsi, plusieurs modèles ont été définis dans le but spécifique de résoudre les problèmes de performances d'architectures particulières ou d'une classe unique d'applications. En plus, les définitions qui existent sont souvent descriptives. Ce type de définition peut générer diverses interprétations qui ne sont pas toujours équivalentes.

Dans l'étude des modèles de cohérence de la mémoire, nous nous sommes déparés alors avec une multitude de modèles définis chacun à sa manière. Dans ce cadre, le concept de modèle de cohérence en tant qu'entité se perd.

Plusieurs chercheurs ont eu le même sentiment et ont essayé de séparer les caractéristiques spécifiques d'un modèle du comportement du modèle de cohérence en tant qu'entité du système. La plupart des études menées dans ce sens [Adv93] [KNA93] [RS95] sont des études théoriques dont le but n'est pas d'implanter les modèles de cohérence mais simplement de les comprendre.

Pour la spécification de notre module de gestion de modèles de cohérence, la définition de modèle de cohérence est primordiale. Nous nous sommes inspirés des définitions formelles de chaque modèle présentées dans le paragraphe 2.2 pour dériver notre définition d'un modèle de cohérence générique. Nous avons ainsi abouti à une définition assez générale qui nous a permis de traiter des différents modèles de cohérence.

Pour $\mathcal{DIVA}$, un modèle de cohérence de la mémoire est *une relation d'ordre*

$\xrightarrow{R}$ *sur un ensemble* $\mathcal{O}$ *de types d'opérations traitées par le modèle* [BM96].

La lecture et l'écriture en mémoire partagée, notées respectivement r et w, sont les deux types d'opérations obligatoires. Ces types d'opérations sont traités par n'importe quel modèle défini dans $\mathcal{DIVA}$. Les opérations de synchronisation peuvent être rajoutées à l'ensemble $\mathcal{O}$. C'est en effet le cas pour les modèles hybrides.

3.4.2 Exécution dans un modèle de cohérence générique

Ayant défini le modèle de cohérence, il nous faut maintenant définir comment il se comporte pour garantir que la relation d'ordre soit respectée. Nous nous intéressons plutôt à l'implantation des modèles de cohérence qui peuvent être définis selon la notation proposée dans le paragraphe précédent. Il est nécessaire de noter qu'il existe plusieurs implantations possibles d'un même modèle de cohérence. Le rôle de $\mathcal{DIVA}$ est d'offrir les mécanismes de base nécessaires à la réalisation d'une ou plusieurs de ces implantations.

L'exécution d'un programme dans un modèle de cohérence générique est montrée dans la figure 3.3.

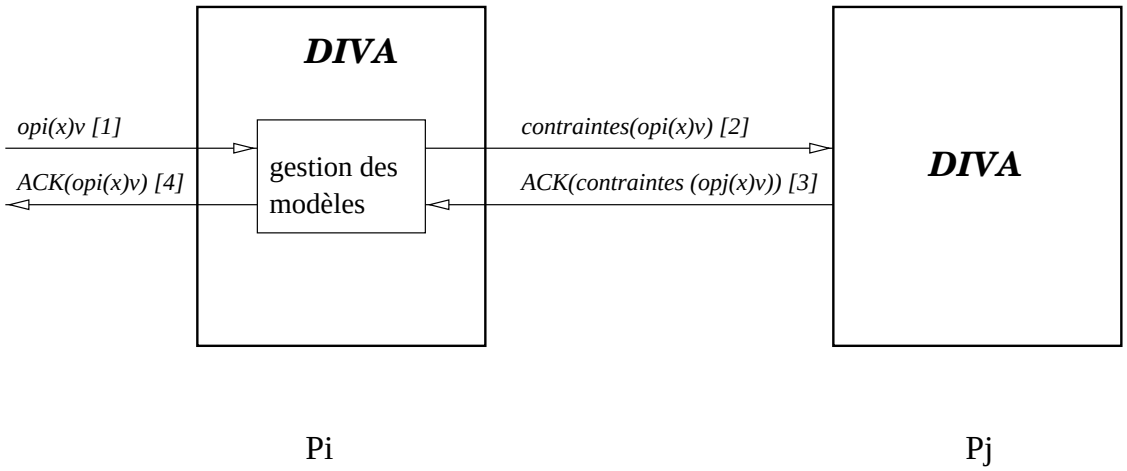


FIG. 3.3 – *Exécution d'un modèle de cohérence générique*

Le module de gestion des modèles est une entité qui reçoit des requêtes d'exécution d'opérations $req(o_{p_i}(x)v)$, vérifie les contraintes d'ordre qui doivent être respectées avant de donner suite à l'exécution de l'opération, exécute les opérations associées aux contraintes imposées et, à la fin, donne suite à l'exécution de l'opération $o_{p_i}(x)v$ (voir figure 3.3).

Les contraintes d'ordre qui seront assurées dépendent du modèle de cohérence courant et peuvent être nulles. Par la suite, nous détaillons la partie de l'exécu-

tion de chaque type d'opération qui se déroule indépendamment du modèle de cohérence.

Exécution d'une lecture : Les lectures $r_{p_i}(x)v$ sont toujours effectuées dans la mémoire locale du nœud qui a sollicité l'opération.

Exécution d'une écriture : Les écritures $w_{p_i}(x)v$ sont toujours effectuées d'abord sur la mémoire locale du nœud qui a sollicité l'opération. Le moment où l'écriture sera effectuée sur les autres nœuds dépend du modèle de cohérence courant. Une opération d'écriture en mémoire $w_{p_i}(x)v$ est alors divisée en un ensemble de sous-opérations d'écriture. Nous avons une sous-opération d'écriture pour chaque nœud qui détient une copie de x.

Selon le modèle de cohérence courant, les opérations qui suivent l'écriture dans le code du processeur p_i peuvent être lancées et même terminées avant que l'écriture soit terminée par rapport aux autres processeurs du système.

Exécution d'un autre type d'opération $(o_{p_i}(x)v)$: L'exécution des opérations autres que la lecture et l'écriture de données doit être entièrement spécifiée par le modèle de cohérence courant. Parmi ces opérations, nous pouvons trouver l'exécution d'opérations de cohérence au moment de la synchronisation et l'exécution de primitives définies par le modèle courant, comme l'association de variables à un verrou dans Midway [BZS93a].

3.4.3 Structure du module de gestion de modèles

Le module de gestion des modèles de cohérence est chargé d'assurer que l'exécution d'une application produit uniquement les résultats valides selon le modèle de cohérence courant. Pour y arriver, il retarde l'exécution des opérations sur la mémoire partagée jusqu'à ce que les contraintes de cohérence définies pour le type de l'opération soient assurées.

La structure du module de gestion des modèles de cohérence est montrée dans la figure 3.4.

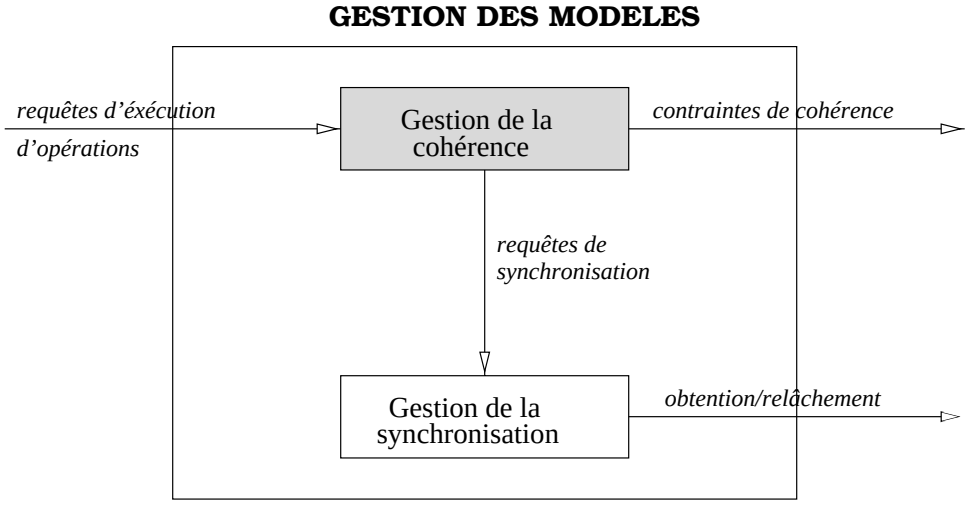


FIG. 3.4 – *Structure du module de gestion de modèles*

Le module de gestion de modèles est activé par la réception d'une requête d'exécution d'opération. Le module qui s'occupe de la gestion de la cohérence vérifie si des opérations de synchronisation doivent être exécutées avant que la cohérence soit assurée. Dans ce cas, une requête de synchronisation est envoyée au module de gestion de la synchronisation. Dès que l'opération de synchronisation est effectuée, le module de gestion de la cohérence vérifie s'il existe des contraintes de cohérence qui doivent être satisfaites. Les opérations qui doivent être exécutées pour satisfaire ces contraintes de cohérence dépendent du type de l'opération. Quand l'exécution de toutes les opérations associées aux contraintes est terminée, le module de gestion de modèles envoie un acquittement au module qui l'a appelé. La réception de cet acquittement permet enfin l'exécution de l'opération.

Le module de gestion de la synchronisation est responsable de l'obtention et du relâchement des objets de synchronisation. Il est activé par le module de gestion de la cohérence quand l'exécution d'une de ces opérations est sollicitée.

Le module de gestion de la cohérence est responsable de l'implantation des différents modèles de cohérence. Son fonctionnement interne est montré dans la figure 3.5.

Les requêtes d'exécution d'opérations sont reçues par le modèle de cohérence générique qui les achemine au modèle de cohérence courant. C'est le modèle de

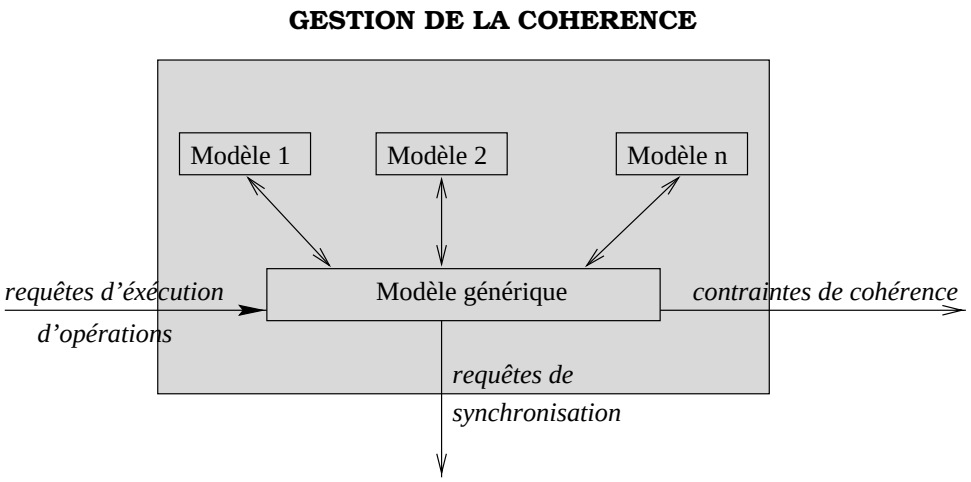


FIG. 3.5 – *Structure du module de gestion de la cohérence*

cohérence courant qui assure l'ordre d'exécution des opérations. Dans certains cas, des requêtes de synchronisation et des contraintes de cohérence peuvent être générées.

3.5 Interface d'implantation des nouveaux modèles

Une des caractéristiques les plus originales de $\mathcal{DIVA}$ est la possibilité accordée à l'utilisateur de rajouter des modèles de cohérence à la machine virtuelle. Le rajout des modèles de cohérence à $\mathcal{DIVA}$ permet que notre système soit aussi utilisé comme une plate-forme de tests des modèles de cohérence de la mémoire.

Un modèle de cohérence de la mémoire dans $\mathcal{DIVA}$ est une relation d'ordre $\xrightarrow{R}$ qui agit sur les opérations d'accès à la mémoire partagée. Si la relation $\xrightarrow{R}$ change, nous avons alors un nouveau modèle de cohérence.

L'exécution d'un programme selon un modèle de cohérence consiste à appliquer les restrictions d'ordre imposées par la relation $\xrightarrow{R}$ aux accès aux données effectués par le programme. A la fin de l'exécution, on a un historique d'exécution H_{p_i} par processeur.

L'implantation d'un modèle de cohérence doit garantir que tous les historiques d'exécution H_{p_i} produits se comportent selon le modèle. Il est possible que des historiques H_{p_i} valides dans un modèle soient interdits dans une implantation particulière de ce modèle. Dans ce cas, l'implantation permet uniquement un sous-ensemble des historiques possibles dans le modèle. Une telle implantation est dite conservative.

Nous verrons par la suite qu'implanter un modèle de cohérence sur $\mathcal{DIVA}$ consite à implanter la relation d'ordre qui définit le modèle. L'implantation des modèles est réalisée en deux étapes: la programmation du modèle et l'incorporation du modèle à notre machine virtuelle.

3.5.1 La programmation d'un nouveau modèle

L'objectif du module d'implantation de modèles de cohérence de $\mathcal{DIVA}$ n'est pas d'offrir à tous les utilisateurs une interface simple qui permet l'ajout dynamique de modèles. L'implantation d'un modèle de cohérence est une tâche complexe qui nécessite une connaissance assez complète du modèle à implanter aussi bien que des fonctions internes de $\mathcal{DIVA}$.

Nous décrivons ici d'une manière simplifiée les étapes nécessaires à la programmation d'un modèle sur $\mathcal{DIVA}$. Chaque modèle de cohérence a ses propres particularités et protocoles qui ne seront pas décrits dans ce paragraphe. Dans l'annexe A, nous donnons une présentation détaillée de l'implantation de la cohérence séquentielle.

Dans $\mathcal{DIVA}$, un modèle $\mathcal{M}$ implante la relation d'ordre qui lui est associée par le schéma suivant:

```
modèle_M( type_opération )
{
  case(type_opération)
    TYPE_OP1: contraintes_type_op1();
    TYPE_OP2: contraintes_type_op2();
    TYPE_OPn: contraintes_type_opn();
}
```

Le modèle de cohérence est programmé par un schéma de description de modèles. En réalité, le programmeur doit associer le type de l'opération aux contraintes d'ordre imposées par le modèle. Ces contraintes d'ordre sont spécifiées par les méthodes de cohérence.

La programmation d'un modèle de cohérence sur $\mathcal{DIVA}$ comprend alors deux étapes: la définition des opérations qui sont traitées par le modèle et la spécification de la méthode de cohérence qui doit être exécutée lors qu'une de ces opérations est effectuée.

Première étape: définition des types d'opération

La relation $\xrightarrow{R}$ agit sur l'ensemble $\mathcal{O}$ de types d'opérations d'accès à la mémoire considéré par le modèle (voir paragraphe 3.4). Le premier pas vers l'implantation du modèle $\mathcal{M}$ consiste donc à identifier l'ensemble $\mathcal{O}$. Pour les modèles uniformes, cet ensemble est composé uniquement des opérations de lecture et écriture en mémoire:

$$\mathcal{O} = \{o_{p_i} \mid type(o_{p_i}) \in \{r, w\}\}$$

Les modèles hybrides rajoutent les opérations de synchronisation à l'ensemble $\mathcal{O}$. $\mathcal{DIVA}$ met à disposition des programmeurs deux types de base de primitives de synchronisation: `diva_lock` et `diva_unlock`. Le programmeur système doit se servir de ces primitives pour implanter la cohérence de la mémoire. Par exemple, pour implanter la cohérence à la libération, l'ensemble $\mathcal{O}$ sera le suivant:

$$\mathcal{O} = \{o_{p_i} \mid type(o_{p_i}) \in \{r, w, diva\_lock, diva\_unlock\}\}$$

D'autres primitives peuvent être définies, selon les besoins du modèle de cohérence de la mémoire en question.

Dans $\mathcal{DIVA}$, les opérations r et w sont en fait les opérations LOAD et STORE d'accès à la mémoire. Ces opérations se déroulent sans l'intervention de notre système quand la page qui les contient est chargée en mémoire. Les seules opérations visibles à $\mathcal{DIVA}$ sont des défauts de page lors d'une lecture ou d'une écriture.

Nous devons alors prévoir des méthodes pour traiter des défauts de page en lecture et écriture à la place des procédures de traitement des opérations r et w, respectivement. Sauf dans le cas d'un défaut de page de protection, le nœud en défaut n'a pas la page chargée dans sa mémoire locale. Il doit alors s'adresser au gestionnaire de la page pour connaître la localisation et l'état de la page dans le système.

Seconde étape: définition de la méthode associé à l'opération

La seconde étape consiste à déterminer le comportement de chaque type d'opération o_{p_i} qui appartient à $\mathcal{O}$. Ce comportement est défini par la méthode `méthode_type_`o_{p_i}`()`.

L'ensemble $\mathcal{P}$ de méthodes de cohérence est défini de la façon suivante:

$$\mathcal{P} = \{methode \mid methode = methode\_type\_o_{p_i}()\}$$

Nous voulons souligner que, pour chaque type d'opération, il est nécessaire de définir une méthode associée. Une méthode comprend souvent un ensemble de procédures qui seront utilisées pour traiter un ensemble de messages échangés entre les nœuds.

3.5.2 Incorporation du modèle au serveur

L'incorporation d'un nouveau modèle à $\mathcal{DIVA}$ est réalisée à l'aide d'un fichier de configuration montré dans la figure 3.6.

Le fichier est divisé en trois parties. La première partie (1) comprend le nom du modèle implanté.

La deuxième partie (2) décrit les opérations considérées par le modèle. Les opérations r et w sont obligatoires. En ce que concerne les opérations de synchronisation, seules les opérations qui exécutent des actions de cohérence doivent être précisées. A ce moment, l'utilisateur peut définir des nouvelles primitives. Sur la figure 3.6, c'est le cas pour la primitive `diva_x_nouveau`.

La troisième partie (3) consiste à décrire les méthodes qui implantent les opérations selon les restrictions du nouveau modèle. Chaque méthode est composée d'une procédure exécutée lorsque l'opération est effectuée et d'un ensemble de procédures exécutées lors de la réception d'un message envoyé par la méthode. Pour toute opération définie en (2), la méthode associée doit être fournie. Le fichier de configuration est terminé par le caractère #.

Un programme d'implantation de modèles lit le fichier de configuration et rajoute le modèle au code de $\mathcal{DIVA}$ (cf. figure 3.6). Pour préserver les performances, nous avons opté pour l'insertion du nouveau modèle de cohérence directement dans le code de $\mathcal{DIVA}$. La compilation du nouveau code de $\mathcal{DIVA}$ crée une nouvelle machine virtuelle qui est maintenant capable d'accepter le modèle X. Pour qu'une application soit exécutée selon X, il suffit de spécifier X comme le modèle courant dans le code de l'application. Selon ce schéma, plusieurs modèles qui exécutent des opérations o_{p_i} selon le type de l'opération peuvent être implantés.

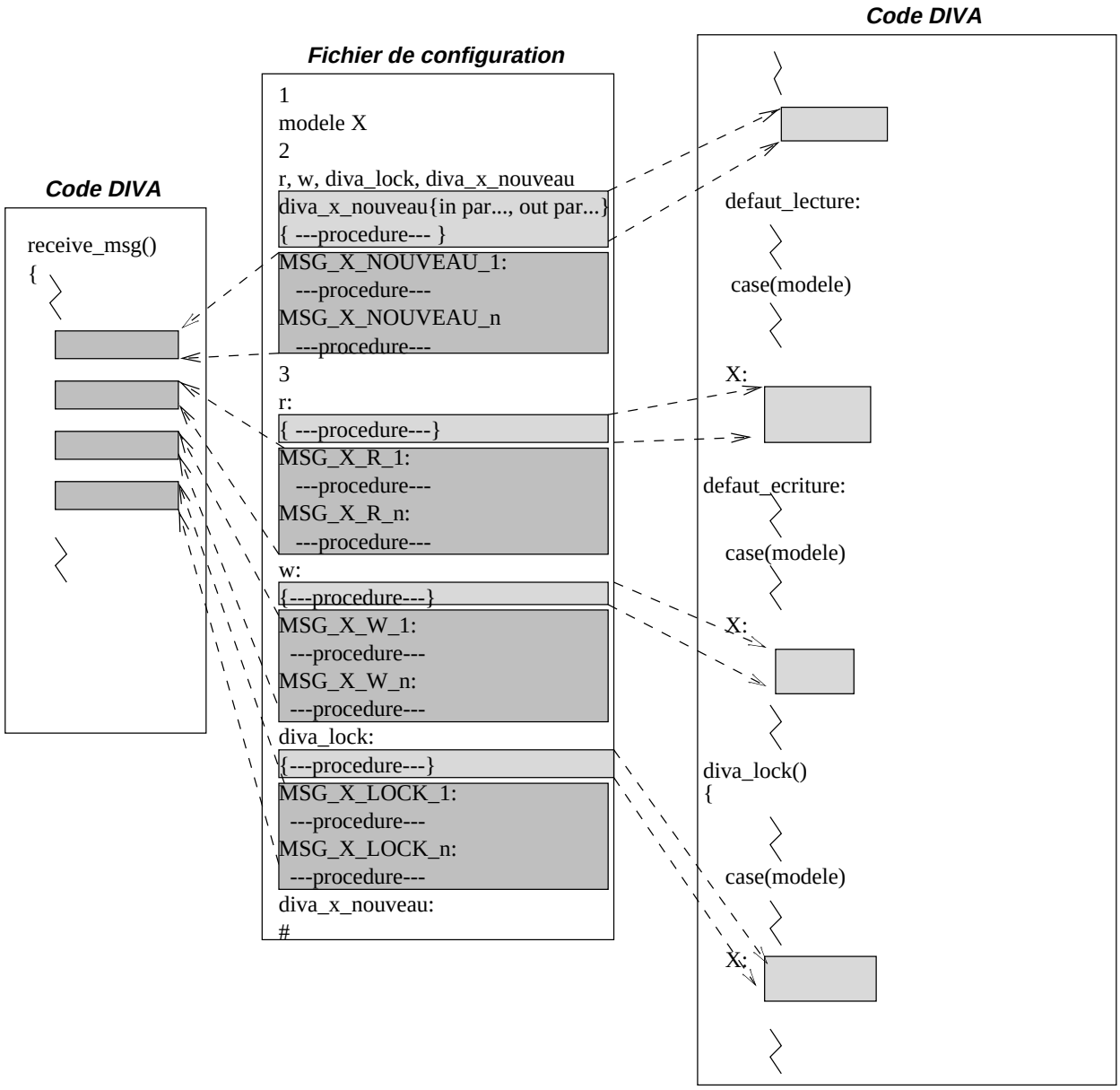


FIG. 3.6 – *Implantation du modèle X*

3.6 Interface de spécification du modèle de cohérence

Pour le cas où le programmeur ne veut pas se soucier des modèles de cohérence de la mémoire, $\mathcal{DIVA}$ se comporte comme un système de mémoire virtuelle partagée traditionnel. Il n'est pas nécessaire d'utiliser des nouvelles primitives. Le modèle de cohérence par défaut est la cohérence à la libération (voir paragraphe 2.2.9).

Nous avons choisi la cohérence à la libération comme le modèle de cohérence par défaut car c'est le modèle qui présente le meilleur compromis entre la simplicité de programmation et les performances pour un grand nombre d'applications [Mos93].

Afin de spécifier un modèle de cohérence de la mémoire autre que la cohérence à la libération, le programmeur doit faire appel à la primitive `diva_set_memory_model`.

La primitive `diva_set_memory_model` établit le modèle de cohérence qui sera utilisé par l'application pendant son exécution. Elle doit être appelée par l'utilisateur avant tout accès aux variables partagées. Sa syntaxe est la suivante:

<model_set> = `diva_set_memory_model`(<model_wanted>)

où <model_set> est le modèle à être utilisé par l'application
<model_wanted> est un modèle parmi ceux offerts par le serveur

Un des processus qui composent l'application demande le modèle <model_wanted>. Si un autre modèle de cohérence de la mémoire a déjà été spécifié par un autre processus de la même application, $\mathcal{DIVA}$ retourne dans le champ <model_set> le modèle spécifié au préalable. En d'autres termes, le premier modèle spécifié est toujours retourné.

Une fois spécifié, le modèle de cohérence est diffusé à tous les nœuds du système. Dans ce contexte, nous pouvons arriver à une situation de conflits d'accès si deux nœuds sollicitent des modèles de cohérence mémoire distincts simultanément.

Nous proposons un protocole pour traiter correctement cette situation. L'exécution du protocole est montrée dans la figure 3.7. Les lignes verticales correspondent au temps, qui croît vers le bas. Ce protocole génère quatre types de messages (VERIF_MM, SET_OK, SET_ERROR et SET_MM). L'échange de messages entre processeurs est représenté par les lignes pointillées.

L'objectif du protocole est de mettre à jour la variable <memory_model> de

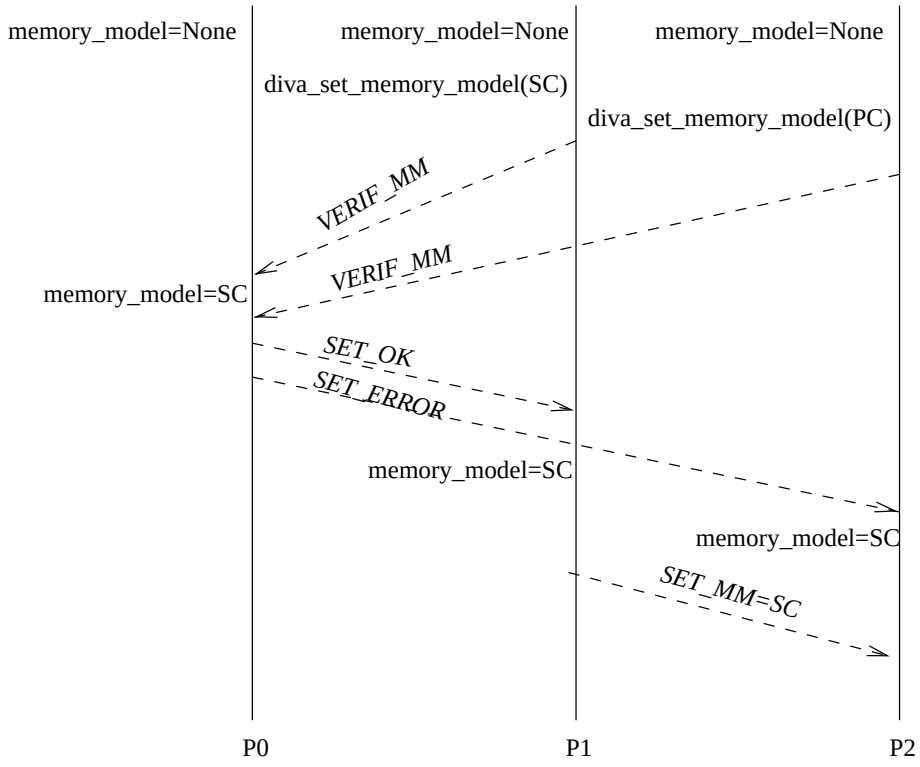


FIG. 3.7 – *Exécution du protocole*

chaque processeur en lui affectant la valeur spécifiée par l'utilisateur. A la fin de l'exécution du protocole, les variables <memory_model> de chaque processeur contiennent toujours des valeurs identiques.

Afin de résoudre les conflits d'accès, nous avons établi un ordre dans la diffusion des modèles de cohérence mémoire: les messages sont toujours envoyés d'abord au nœud 0. Le nœud 0 est donc l'arbitre de l'attribution.

Dès que le nœud 0 reçoit le message VERIF_MM, il vérifie si le modèle de cohérence de la mémoire a déjà été attribué. Dans le cas affirmatif, il envoie le message SET_ERROR au nœud origine. Ce message spécifie aussi le modèle courant. Dans le cas contraire, le nœud 0 affecte à sa variable <memory_model> le modèle envoyé par le nœud origine et envoie le message SET_OK à celui-ci.

Le nœud origine reste bloqué en attendant la réponse. S'il reçoit SET_ERROR, il affecte la valeur du modèle de cohérence contenue dans le message envoyé par le nœud 0 à sa propre variable <memory_model> et cette valeur est retournée au processus qui a fait appel à la primitive. S'il reçoit SET_OK, il affecte la valeur spécifiée par l'utilisateur à sa variable <memory_model> et continue la diffusion. Cette fois-ci, l'envoi de messages est asynchrone et le processeur ne reste pas bloqué en attendant la réponse des nœuds destinataires.

3.7 Gestion de la synchronisation dans $\mathcal{DIVA}$

Dans le paragraphe 3.4, nous avons vu que $\mathcal{DIVA}$ supporte plusieurs modèles de cohérence de la mémoire. Une classe de modèles, dits hybrides, exécute des opérations de cohérence au moment de la synchronisation. La conception des mécanismes de synchronisation doit alors permettre l'exécution de quelques opérations de cohérence.

Une première approche consiste à offrir aux utilisateurs de notre système des mécanismes de synchronisation traditionnels tels que les opérations d'obtention et relâchement de verrous. Chaque modèle hybride se sert alors des mécanismes de synchronisation de base pour implanter les primitives de synchronisation dont il a besoin.

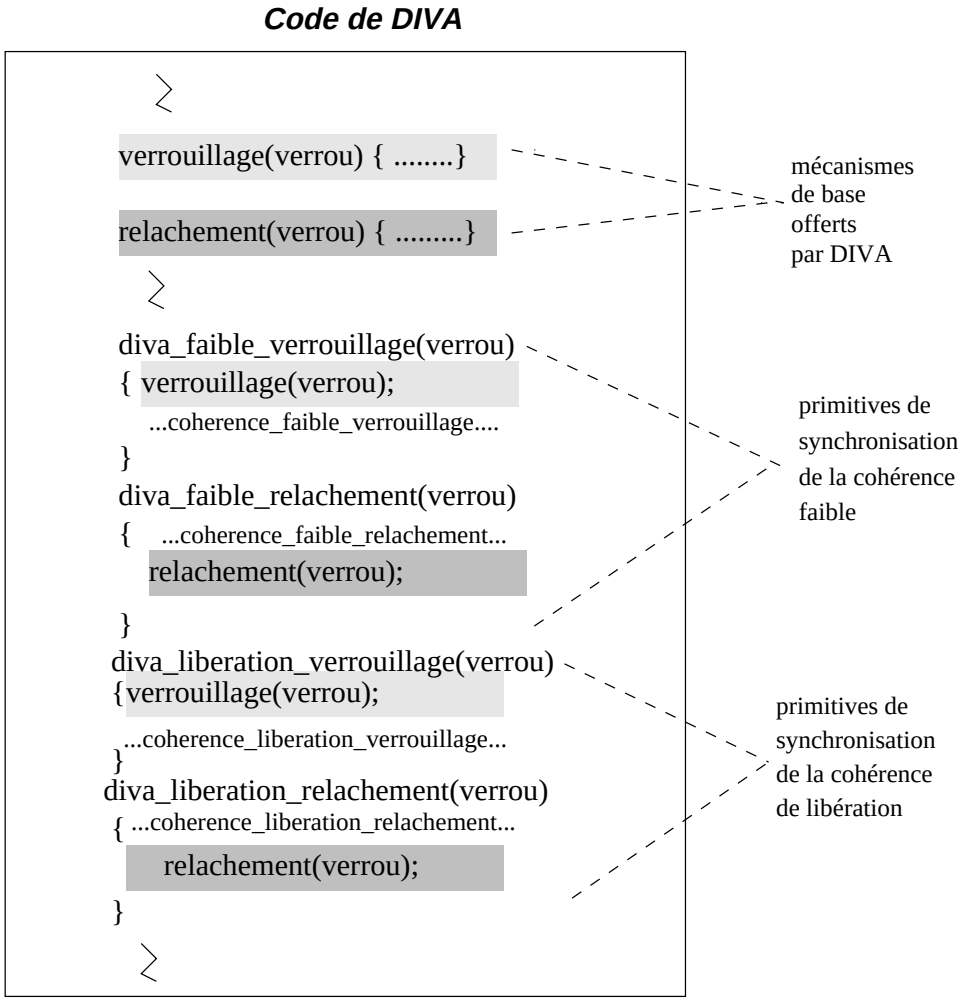


FIG. 3.8 – *Mécanismes de synchronisation selon l'approche traditionnelle*

La figure 3.8 illustre un système $\mathcal{DIVA}$ conçu suivant cette première approche.

Dans la figure, le système offre deux modèles: la cohérence faible et la cohérence à la libération.

Nous pouvons noter que cette approche entraîne une prolifération de primitives de synchronisation, au moins deux primitives par modèle de cohérence hybride. De plus, pour changer de modèle de cohérence d'une application, il faut modifier le code pour appeler la bonne primitive.

Cette approche traditionnelle n'est donc pas adaptée à un système à modèles de cohérence multiples tel que $\mathcal{DIVA}$.

Il faut rechercher des solutions qui préservent la simplicité de programmation; c'est à dire que le changement du modèle de cohérence courant doit causer un petit nombre de modifications dans le code de l'application.

Nous avons adopté l'approche qui consiste à concevoir une interface unique de synchronisation; la syntaxe des primitives de synchronisation ne dépend pas du modèle de cohérence courant, seule la sémantique de la primitive change en fonction du modèle.

Code de DIVA

```
 >
diva_lock(verrou)
{ verrouillage(verrou) { .....}
  case(memory_model)
  { FAIBLE: coherence_faible_verrouillage
    LIBERATION: coherence_liberation_verroullaige
  }
}

diva_unlock(verrou)
{case(memory_model)
  { FAIBLE: coherence_faible_relachement
    LIBERATION: coherence_liberation_relachement
  }

 relachement(verrou) { .....}
}
<
```

FIG. 3.9 – *Primitives de synchronisation de* $\mathcal{DIVA}$

$\mathcal{DIVA}$ offre deux primitives de synchronisation (`diva_lock` et `diva_unlock`) qui sont montrées dans la figure 3.9. A titre comparatif, nous présentons sur cette figure le code des primitives du modèle de cohérence faible et du modèle de cohérence à la libération selon l'approche utilisée par $\mathcal{DIVA}$.

Les primitives de synchronisation de $\mathcal{DIVA}$ servent à deux objectifs différents. D'une part, elles permettent de garantir l'exclusion mutuelle entre les différents processus qui composent l'application. D'autre part, elles activent des opérations de cohérence quand le modèle courant de cohérence de la mémoire est un modèle hybride.

Les opérations directement liées à la synchronisation sont `verrouillage` (<verrou>) et `relachement` (<verrou>). Ces opérations sont exécutées dans le module de gestion de la synchronisation de façon identique pour tous les modèles de cohérence. Elles implantent l'obtention et la libération d'un verrou, respectivement.

C'est la partie qui implante la cohérence au moment de la synchronisation qui est exécutée selon le modèle spécifié dans la variable <memory_model>.

Les opérations de cohérence sont en effet un ensemble de procédures exécutées par le gestionnaire de la cohérence. La procédure exécutée est choisie selon le modèle de cohérence courant. Pour les modèles uniformes, la procédure de cohérence est nulle.

Si un utilisateur veut changer le modèle de cohérence courant, il suffit de remplacer la primitive `diva_set_memory_model`(FAIBLE) par `diva_set_memory_model`(LIBERATION). Les appels aux primitives de synchronisation restent inchangés. C'est à $\mathcal{DIVA}$ d'appliquer la sémantique associée au modèle courant.

3.7.1 Opérations de synchronisation

Les opérations de synchronisation proposées dans notre serveur sont très simples. Des mécanismes plus complexes peuvent être conçus avec ces primitives de base. La simplicité est une caractéristique souhaitable quand on veut offrir une même interface de synchronisation à tous les modèles de cohérence de la mémoire.

Les entités de synchronisation traitées par $\mathcal{DIVA}$ sont les verrous. Ils sont gérés par un gestionnaire distribué fixe (voir paragraphe 2.3).

Les opérations sur les verrous se traduisent en appels à notre machine virtuelle. $\mathcal{DIVA}$ met à la disposition de l'utilisateur quatre primitives: `diva_get_lock`, `diva_lock`, `diva_unlock` et `diva_remove_lock`.

La correspondance entre le gestionnaire et le verrou s'effectue lors de l'exécution de la primitive `diva_get_lock`. Cette primitive retourne à l'utilisateur le numéro de verrou n_v. C'est en fait un indice pour accéder une table distribuée de verrous. Le gestionnaire du verrou est obtenu par la fonction n_v MOD n_{proc} où n_{proc} est le nombre de processeurs du système. Pour toute manipulation postérieure, le numéro du verrou n_v doit être fourni.

Un processus demande l'accès exclusif à un verrou à travers la primitive `diva_lock`. Un verrou n'est accordé qu'à un processus à la fois. Si n processus essayent d'obtenir le même verrou simultanément, $n-1$ processus seront bloqués et mis en attente. Ainsi, à chaque verrou est associée une liste chaînée de processus en attente. Cette liste est gérée selon la politique FIFO.

Le relâchement d'un verrou est réalisé par la primitive `diva_unlock`. Un processus en attente est alors retiré de la liste et peut reprendre son exécution.

Un verrou est supprimé par la primitive `diva_remove_lock`. L'exécution de cette primitive rend le verrou inaccessible à tous les processus qui composent l'application.

3.7.2 Opérations de cohérence

Les opérations de cohérence associées à une primitive de synchronisation sont celles spécifiées au moment de la définition des modèles de cohérence de la mémoire (voir paragraphe 3.5).

Typiquement, les opérations de cohérence appliquées consistent à vider un tampon de pages modifiées par le processeur. Dès lors, les modifications sont transmises aux processeurs qui ont ces mêmes pages cachées sur leurs mémoires locales.

3.8 Relation avec les autres travaux

L'objectif de ce paragraphe est de discuter la relation entre $\mathcal{DIVA}$ et d'autres études ménées précédemment dans le domaine des modèles de cohérence multiples.

3.8.1 Relation avec le travail de Heddaya et Sinha

A. Heddaya et H. Sinha ont proposé un formalisme pour décrire les modèles de cohérence de la mémoire dans [HS92] et ont implanté un prototype (Mermera)

qui se sert de ce formalisme. L'implantation de Mermera est décrite dans [HS93].

Selon les auteurs, la description d'un modèle de cohérence de la mémoire est fondée sur des "événements de mémoire". Un événement de mémoire est, par exemple, l'exécution d'une opération d'accès à la mémoire. Un modèle de cohérence de la mémoire spécifie les conditions qui garantissent la sémantique de la mémoire. Ces conditions génèrent un ensemble d'ordres d'événements qui sont permis.

Les modèles de cohérence suivants ont été décrits selon le formalisme proposé: la cohérence séquentielle, la cohérence causale, la cohérence du processeur, la mémoire lente, la cohérence locale et la mémoire faible.

Les auteurs justifient l'existence de plusieurs modèles de cohérence par l'observation de certaines classes de programmes qui tournent sans modification de code sur les modèles de cohérence plus relâchés. Ainsi, ces programmes profitent des améliorations en performance apportées par ces modèles sans que la programmation devienne trop compliquée. Le modèle le plus adapté à une application dépend de ses caractéristiques d'accès.

De manière similaire à Heddaya et Sinha, nous croyons qu'il est nécessaire d'offrir à l'utilisateur la possibilité de choisir entre plusieurs modèles de cohérence de la mémoire. Nous avons eu aussi besoin de définir les modèles selon un formalisme unique avant de les implanter.

Nous avons opté pour une description selon les historiques d'exécution à la place de la description selon les événements de mémoire proposée par Heddaya et Sinha. Pour nous, un modèle de cohérence de la mémoire est une relation unique d'ordre sur les opérations d'accès à la mémoire qui sont permises par le modèle. Notre description a été développée dans le but spécifique de rendre possible et simple l'implantation d'un modèle sur notre module de modèles multiples. En définissant le modèle comme une relation unique, il nous suffit de changer la relation d'ordre pour changer de modèle. Nous croyons que l'existence d'une seule relation rend plus simple la tâche d'implantation des modèles.

Dans Mermera, plusieurs modèles de cohérence peuvent être actifs dans une même exécution d'une application. L'accès aux données est effectué par des primitives de lecture et écriture.

Il existe uniquement une primitive de lecture de données. La lecture est toujours effectuée sur la copie locale de la donnée. Il existe une primitive d'écriture par modèle de cohérence. Jusqu'à maintenant, les écritures suivantes ont été implantées: `CO_write(adr, valeur)` (cohérence séquentielle), `PRAM_write(adr, valeur)` (cohérence PRAM), `slow_write(adr, valeur)` (mémoire lente), `local_write(adr, valeur)` (cohérence locale).

Contrairement à Mermera, nous ne nous servons pas de primitives pour accé-

der à la mémoire partagée. Dans notre cas, l'accès est effectué par les opérations d'accès à la mémoire (LOAD et STORE). Nous pensons que l'utilisation de primitives présente deux inconvénients. D'abord, le modèle de programmation devient plus complexe. Une opération simple d'affectation entre variables partagées requiert dans ce cas deux appels de primitives (`x_write(adr1, read(adr2))`). De plus, l'exécution des primitives d'accès à la mémoire a toujours le coût additionnel d'un appel à primitives. Nous avons alors opté pour l'utilisation des mécanismes de pagination dans le but de rendre l'interface de programmation la plus proche possible de la programmation monoprocesseur.

Selon notre schéma d'exécution, un seul modèle doit être associé à une exécution d'une application. La possibilité de coexistence entre plusieurs modèles dans une même exécution est exclue dans notre système. Nous voulons offrir à l'utilisateur l'abstraction d'une mémoire unique qui se comporte selon des règles pré-définies. Ces règles doivent être appliquées indistinctement à toutes les données qui composent cette mémoire. Nous croyons que la préservation du concept d'unicité de la mémoire rend la programmation plus simple.

Néanmoins, comme nous accordons à l'utilisateur la possibilité de définir ses propres modèles de cohérence, rien ne l'empêche de définir des nouveaux types de données et comportements différents associés à chaque type. Pour notre système, cependant, le modèle est encore unique.

Le dernier aspect à considérer concerne la possibilité de définition de nouveaux modèles. Dans le cas de Mermera, l'utilisateur doit choisir parmi un ensemble de modèles pré-définis. Dans notre système, nous offrons de plus la possibilité de définition de nouveaux modèles. Nous pensons que l'interface d'implantation de nouveaux modèles donne à notre machine virtuelle la flexibilité nécessaire pour l'exécution performante de plusieurs classes d'applications. Cette caractéristique la transforme aussi en une plate-forme de tests des modèles de cohérence.

3.8.2 Relation avec le travail de Bershad et Zekauskas

Bershad et Zekauskas ont défini l'environnement de programmation Midway en 1991 [BZ91b]. Midway a été initialement proposé pour implanter la cohérence d'entrée, un modèle proposé par les auteurs.

Dans [BZS93b], une nouvelle version de Midway est proposée qui supporte plusieurs modèles de cohérence. Etant donné que la cohérence d'entrée est un modèle de cohérence très relâché, les auteurs ont opté pour offrir quelques autres modèles plus forts de cohérence. Les modèles offerts par cette nouvelle version sont la cohérence du processeur et la cohérence à la libération. Evidement, la cohérence d'entrée est encore supportée.

De façon similaire à Heddaya et Sinha, Bershad et Zekauskas permettent la coexistence entre plusieurs modèles de cohérence de la mémoire dans une même exécution d'une application. Le modèle de cohérence par défaut est la cohérence à la libération. L'accès aux variables associées à un intervalle de vidage ("flush") se comportent selon la cohérence du processeur. La cohérence d'entrée est appliquée aux variables associées à un objet de synchronisation. L'addition de nouveaux modèles n'est pas possible dans Midway. Comme ces aspects sont similaires dans Midway et dans Mermera, les mêmes commentaires réalisés à ce propos dans le paragraphe précédent sont aussi valides.

Midway supporte trois modèles de cohérence qui traitent différemment les opérations de synchronisation. Cependant, Midway offre une interface unique de synchronisation. La diffusion des modifications est toujours effectuée au moment de l'obtention d'un objet de synchronisation. Sur la cohérence du processeur, les modifications sont en plus diffusées périodiquement. Sur la cohérence d'entrée, uniquement les modifications apportées aux variables gardées par l'objet de synchronisation sont diffusées.

De manière similaire à Midway, nous proposons une interface unique de synchronisation. Cependant, notre approche est beaucoup plus flexible. L'interface unique de synchronisation est offerte dans notre cas aux modèles existants dans $\mathcal{DIVA}$ ainsi qu'aux modèles qui y seront incorporés.

Dans Midway, l'utilisation d'une interface unique de synchronisation pour des modèles autres que ceux actuellement supportés semble improbable. Dans Midway, l'interface unique de synchronisation semble être une simple conséquence de l'addition des deux modèles de cohérence au modèle d'exécution. Dans le cas de $\mathcal{DIVA}$, cette facilité est une décision prise lors de la conception du système.

L'accès aux variables partagées est effectuée à travers des opérations d'accès à la mémoire (LOAD et STORE). Néanmoins, Midway ne traite pas des défauts de page. Au contraire, des modifications ont été réalisées dans le compilateur pour qu'une structure auxiliaire soit modifiée après chaque écriture en variable partagée.

Nous pouvons citer au moins deux inconvénients dans cette approche. D'abord, la portabilité de Midway est compromise car elle requiert des modifications dans le compilateur. Ensuite, chaque opération d'écriture comprend au moins deux accès à la mémoire: un accès pour écrire la donnée et autre pour écrire la structure auxiliaire.

Nous pensons que l'utilisation des mécanismes de mémoire virtuelle rend le système plus facilement portable et aussi plus performant, si des effets causés par la pagination tels que faux partage sont identifiés et traités.

Chapitre 4

Gestion des pages dans $\mathcal{DIVA}$

L'introduction des mécanismes qui gèrent les modèles de cohérence multiples a eu des conséquences sur la gestion de la mémoire virtuelle. Nous avons eu besoin d'adapter quelques mécanismes, notamment le remplacement et le préchargement de pages, pour supporter les différents modèles de cohérence.

Pour le remplacement des pages, nous avons noté que la suppression d'une page de la mémoire n'est pas souhaitable quand les modifications effectuées sont en train d'être propagées aux autres copies. En plus, le temps rajouté de transfert d'une page au disque était trop important. Nous proposons alors un mécanisme qui décide de la page à remplacer selon le type de page et place la page choisie dans un nœud distant [BM94a].

En observant que le temps de chargement d'une page en mémoire est très important, nous avons proposé un mécanisme de préchargement de pages pour essayer de réduire ce délai. L'existence des modèles de cohérence multiples a compliqué la conception de ce mécanisme puisque la cohérence doit aussi être assurée pour les pages préchargées. Les contraintes de cohérence à assurer dépendent du modèle de cohérence courant.

Ce chapitre est organisé en deux grandes paragraphes. Le premier paragraphe présente le mécanisme de remplacement de pages proposé dans $\mathcal{DIVA}$. De même, le second paragraphe présente le mécanisme de préchargement.

4.1 Remplacement des pages dans $\mathcal{DIVA}$

Nous proposons ici un algorithme simple pour traiter le problème du remplacement de pages. L'algorithme se sert du niveau de stockage intermédiaire créé par la mémoire virtuelle partagée pour stocker les pages à supprimmer de la

mémoire locale.

L'algorithme proposé s'exécute sur tous les nœuds du système. Il prend la décision du choix du nœud vers lequel une page doit migrer en consultant l'état d'occupation des mémoires des nœuds voisins. Si toutes les mémoires sont pleines, l'algorithme se comporte comme un algorithme de remplacement de pages traditionnel, en envoyant la page au disque.

Avant de présenter l'algorithme de remplacement, nous analysons d'abord la dynamique de l'occupation de la mémoire locale de chaque nœud. Nous présentons ensuite l'algorithme de remplacement proposé. Nous détaillons aussi les deux procédures principales de l'algorithme: celle qui choisit la page à remplacer et celle qui définit la nouvelle localisation de la page.

4.1.1 Dynamique d'occupation de la mémoire

Pour simplifier, nous supposons ici qu'une application parallèle s'exécute de manière exclusive sur un ensemble de n nœuds et que chaque nœud exécute un seul processus. D'abord, nous supposons que la mémoire locale à chaque nœud a une taille infinie.

Au début de l'exécution d'un processus sur un nœud s, la mémoire locale de s ne contient que des cadres de pages libres. Autrement dit, le nombre de cadres de pages libres est égal au nombre de cadres de pages de la mémoire. A mesure que les pages sont référencées par le processus qui s'exécute sur s, elles sont chargées en mémoire et occupent des cadres de page qui étaient autrefois libres. Un processus peut référencer un nombre quelconque de pages.

Quand un processus termine son exécution, les cadres de pages occupés sont libérés et le système retourne à son état initial ($cp_{memoire} = cp_{libre}$).

Dans les systèmes réels, où la mémoire a une taille finie, un processus peut référencer un nombre de pages plus important que le nombre de cadres de page de sa mémoire locale. L'introduction d'une mémoire finie complique donc la dynamique d'occupation de la mémoire. Il faut maintenant supprimer une page de la mémoire pour libérer de la place pour charger la page référencée. Cette décision est prise par l'algorithme de remplacement.

L'algorithme de remplacement est fréquemment exécuté par un démon qui maintient l'occupation de la mémoire au-dessous d'un seuil pré-défini. L'objectif du démon est de réduire la probabilité d'occurrence de la condition *mémoire_pleine*. Dans $\mathcal{DIVA}$, l'algorithme de remplacement a aussi été implanté comme un démon.

Le comportement de la mémoire locale dans $\mathcal{DIVA}$ est montré dans la fi-

gure 4.1.

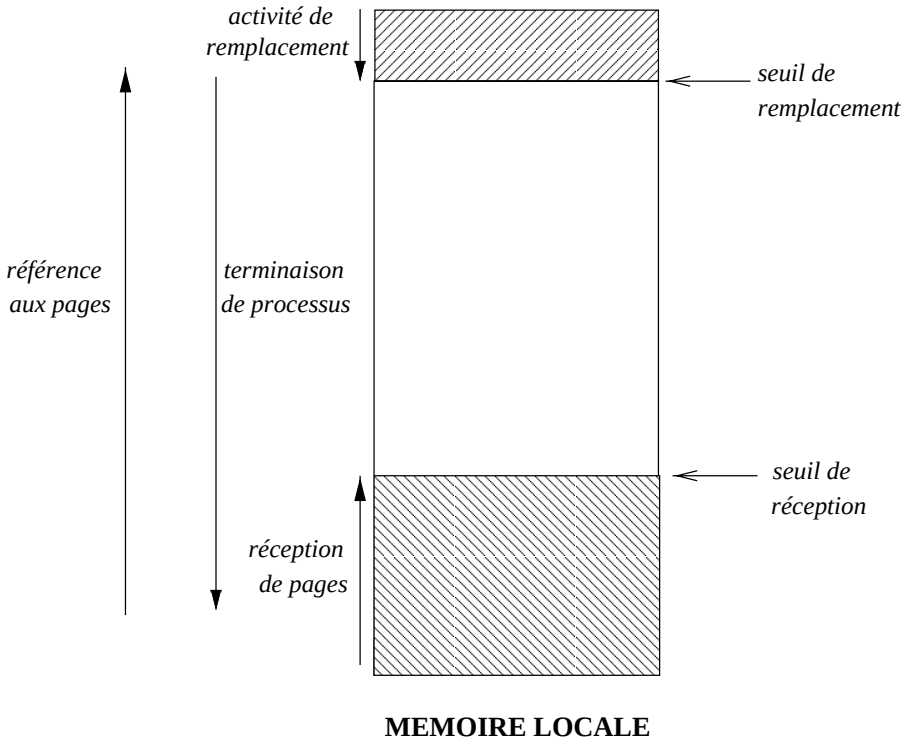


FIG. 4.1 – *Dynamique d'occupation de la mémoire*

Chaque page référencée par un processus est chargée en mémoire locale. Ceci fait croître l'occupation de la mémoire. Dès que l'occupation de la mémoire atteint le seuil du remplacement, le démon de remplacement commence à libérer des pages. La libération des pages ne cesse que quand l'occupation de la mémoire est inférieure au seuil de remplacement.

Les pages libérées sont envoyées à un nœud voisin dont l'occupation de la mémoire est inférieure au seuil de réception. Le fait de recevoir des pages fait aussi croître l'occupation de la mémoire. La fin de l'exécution d'un processus entraîne la libération de toutes ses pages.

Comme les nœuds stockent des pages référencées par des processus distants, la mémoire locale à un nœud peut contenir des pages même si aucun processus ne s'y exécute localement.

4.1.2 Algorithme de remplacement

L'algorithme de remplacement de pages du serveur $\mathcal{DIVA}$ est présenté ci-dessous:

```
démon_de_remplacement()
{
    recevoir_message(mémoire_surchargée);
    TANT QUE occupation_mémoire ≥ seuil_remplacement
        algorithme_de_remplacement();
}

algorithme_de_remplacement()
{
    page = choisir_page_à_remplacer();
    SI (modifiée(page))
    {
        nœud = choisir_nœud();
        SI existe(nœud)
            envoyer_page_au_nœud(nœud);
        SINON
            envoyer_page_au_disque();
        envoyer_message(gestionnaire, page, IN_TRANSIT);
    }
    SINON
        envoyer_message(gestionnaire, page, UNMAP);
    mettre_à_jour_table_pages(page, UNMAP);
    occupation_mémoire = occupation_mémoire - 1;
}

recevoir_page_à_remplacer()
{
    TANT QUE vrai
    {
        recevoir_page(page, nœud_origine);
        TANT QUE (! allouer_page(page))
            algorithme_de_remplacement();
        mettre_à_jour_table_pages(page, type_de_page);
        envoyer_message(gestionnaire, page, type_de_page);
        occupation_mémoire = occupation_mémoire + 1;
    }
}
```

Dans le paragraphe 2.4.5, nous avons vu que résoudre le problème du remplacement des pages dans les systèmes parallèles consiste à trouver les réponses à deux questions. D'abord, il faut décider quelle page sera supprimée de la mémoire locale. Ensuite, il faut trouver de la place pour stocker la page à supprimer, si

jamais elle a été modifiée.

La première procédure à être exécutée est celle qui choisit la page à supprimer de la mémoire. Si la page choisie a été modifiée, l'algorithme choisit le nœud vers lequel la page va migrer. Le gestionnaire de la page est informé à propos de la nouvelle localisation de la page et la page est envoyée au nœud destinataire. Sinon, le nœud local informe le gestionnaire que la page a été supprimée. Dans les deux cas, le cadre de page[1] correspondant à la page est marqué FREE et l'occupation de la mémoire est décrementée.

L'algorithme présenté est un algorithme traditionnel de remplacement des pages. L'originalité de notre approche repose sur la conception des procédures `choisir_page_à_remplacer()` et `choisir_nœud()`.

Dans le reste de ce paragraphe, nous présentons les solutions qui ont été retenues dans $\mathcal{DIVA}$ pour le choix de la page à remplacer et le choix du nœud vers lequel la page modifiée doit migrer. D'abord, le diagramme d'états des pages est présenté. Ensuite, nous décrivons la procécure de choix de la page à remplacer. Enfin, nous présentons le mécanisme de décision de la nouvelle localisation de la page à supprimer dans le cas où celle-ci a été modifiée.

4.1.3 Diagramme de transition d'états des pages

Dans $\mathcal{DIVA}$, un cadre de page peut se trouver dans l'un des cinq états décrits dans la figure 4.2.

FREE	Le cadre de page est libre
READ	Le cadre de page contient la copie unique d'une page en lecture
MREAD	Le cadre de page contient une copie d'une page en lecture
WRITE	Le cadre de page contient une page en écriture
BORROWED	Le cadre de page contient une page qui a été choisie pour le remplacement dans un autre nœud

FIG. 4.2 – *Les états des pages*

Le diagramme de transition d'états des pages est présenté dans la figure 4.3.

1. cadre de page ("page frame") - unité de la mémoire physique utilisée pour stocker les pages

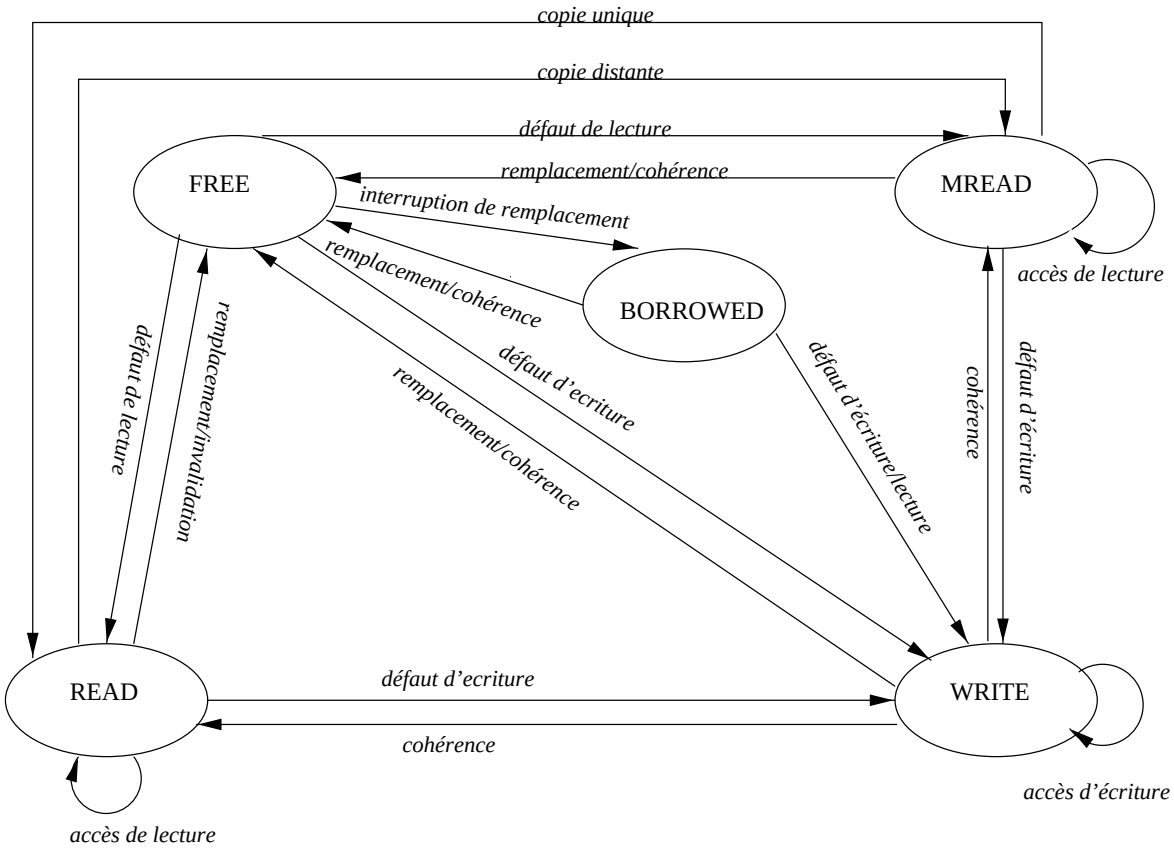


FIG. 4.3 – *Diagramme d'états de page*

Au début de l'exécution, toutes les cadres sont dans l'état libre (FREE). Si un défaut de page en lecture est généré et il n'existe aucune copie de la page dans le système, la page est chargée à partir du disque et mise dans un cadre de page. L'état de ce cadre est changé en READ. S'il existe déjà des copies de la page dans des mémoires distantes, la copie est mise dans un cadre marqué MREAD (lecture multiple). La première copie de la page devient elle aussi MREAD. Les pages en défaut d'écriture sont marquées comme WRITE.

Au cours de l'exécution du programme, des messages de cohérence peuvent arriver pour une page. Si le message en question est une invalidation, le cadre correspondant à la page est mise dans l'état FREE. Le fait d'être choisie par l'algorithme de remplacement peut aussi libérer le cadre de page (état FREE).

Les pages empruntées (BORROWED) sont créées par une interruption de remplacement générée par un nœud distant. Deux évenements peuvent faire qu'un

cadre de page BORROWED soit mis dans l'état FREE. Le premier correspond au cas où la page est référencée par un nœud distant et est migrée vers ce nœud. Le second événement correspond au cas où l'algorithme de remplacement choisit une page BORROWED. Dans ce cas aussi, le cadre correspondant est mis dans l'état FREE. Si jamais une page empruntée est référencée par le nœud local, son état est changé en WRITE.

Le diagramme d'états présenté dans la figure 4.3 ne montre que les états stables de la page. D'autres états, tels que EN_TRANSIT et EN_MODIFICATION sont dits états volatiles ou instables car ils modélisent la transition entre deux états stables.

Les pages qui se trouvent dans les états instables ne sont pas choisies pour le remplacement. Pour interdire le choix de ces pages, $\mathcal{DIVA}$ réalise le verrouillage en mémoire de toutes les pages instables. Un exemple d'état instable est celui produit par la modification des pages qui se trouvent à l'intérieur d'une section critique sur le modèle de cohérence à la libération. Avant que l'opération de relâchement soit exécutée, ces pages sont dans l'état EN_MODIFICATION.

4.1.4 La page à remplacer

Le choix de la page à remplacer est réalisé par la procédure *choisir_page_à_remplacer*. De manière similaire à la plupart des algorithmes de remplacement de pages dans les machines monoprocesseurs, ce choix est réalisé dans notre système selon la politique LRU. Nous avons créé une liste LRU par type de page et ensuite nous avons attribué des priorités de remplacement à ces listes: BORROWED > MREAD > READ > WRITE.

Les pages qu'il faut considérer en premier pour le remplacement sont les pages empruntées. Ces pages ont été chargées dans la mémoire locale à la suite d'une opération de remplacement initiée par un autre processeur. Les pages empruntées sont tolerées uniquement quand la mémoire locale n'est pas surchargée. Dans le cas de surcharge, les pages empruntées sont envoyées vers un autre nœud qui a de la place.

Les pages du type MREAD ont la plus grande priorité de remplacement parmi les pages référencées par le nœud local. Puisqu'il existe plusieurs copies de ces pages dans le système, la copie locale est simplement supprimée, en accord avec le gestionnaire de la page. Cette opération cause pourtant une réduction du degré de partage de la page.

S'il n'existe pas de page MREAD dans la mémoire locale, la prochaine catégorie recherchée est celle des pages READ. Le remplacement d'une page READ peut causer une opération de mouvement si la page a été modifiée. Si la page n'a pas été modifiée, elle est simplement supprimée de la mémoire locale. Une prochaine référence à cette page nécessite un transfert du disque.

La dernière catégorie recherchée est celle des pages WRITE. Le remplacement d'une page WRITE occasionne toujours une migration.

4.1.5 La nouvelle localisation de la page

Algorithme

La mémoire virtuelle partagée simule un niveau intermédiaire de mémoire qui est composé de toutes les mémoires locales des nœuds. La capacité de stockage de ce niveau est la somme de la taille de toutes les mémoires des processeurs qui composent le système.

Contrairement aux algorithmes traditionnels de remplacement, notre algorithme essaye de profiter de ce niveau intermédiaire de mémoire et stocke les pages à remplacer sur un nœud distant.

La décision de placer une page sur un nœud distant est en effet une décision d'allocation de ressources. Pour résoudre ce problème d'allocation, il nous faut envoyer la page à un nœud qui a de la place pour la stocker. Le maintien de l'état de l'occupation de la mémoire des nœuds est essentiel pour le choix de la nouvelle localisation de la page.

Notre choix du nœud vers lequel la page va migrer est basé sur une table des voisins physiques. La table de voisins physiques contient un indicateur LIBRE/PLEINE par processeur. Une mémoire est considérée LIBRE par notre algorithme si son occupation est inférieure au seuil de réception. La seule exigence concernant le seuil de réception est que celui-ci doit être inférieur au seuil de remplacement (voir figure 4.1). Dès que l'occupation de la mémoire dépasse le seuil de réception, l'indicateur LIBRE/PLEINE est positionné à PLEINE.

A chaque changement d'état, un message est envoyé à tous les nœuds voisins. La réception de ce message déclenche la mise à jour de l'indicateur LIBRE/PLEINE correspondant au nœud qui a envoyé le message.

Le choix de la nouvelle localisation de la page est realisé par la procédure

choisir_nœud() montrée ci-dessous.

```
choisir_nœud()
{
    TANT QUE il_existe(voisin)
    {
    SI (table_de_voisins[voisin] == LIBRE)
        retourne(voisin);
    voisin = prochain(voisin);
    }
    retourne(AUCUN);
}
```

Un seul message est envoyé au nœud choisi. Ce message contient la page et la requête pour la placer.

Extensibilité

Pour atteindre l'extensiblité, il faut que l'algorithme soit indépendant de la taille du système. Selon Kermier et al. dans [KK92], un algorithme extensible doit être distribué de manière symétrique parmi les nœuds et chaque nœud ne doit maintenir qu'une vision partielle de l'état du système.

Une copie de notre algorithme s'exécute sur chaque nœud du système. La condition de distribution symétrique est donc respectée.

Le choix de la nouvelle localisation de la page est basé sur une table de voisins physiques. Cette table ne contient que les processeurs qui ont des liens directs avec le processeur local. Ainsi, ni la taille de la table de voisins ni le nombre de messages échangés ne dépendent de la taille du système. Chaque nœud ne connaît que l'état des nœds qui lui sont physiquement connectés (voisins directs).

Au moment de l'exécution de l'algorithme, la décision prise est entièrement fondée sur des informations locales.

Stabilité

Un algorithme est dit stable s'il est capable d'éviter des mauvaises décisions d'allocation [KK92]. Dans notre cas, une mauvaise décision est celle d'envoyer une page à un nœud qui n'a pas de place pour la stocker.

Nous verrons par la suite qu'il est impossible de garantir que cette propriété soit toujours observée. Ce que nous faisons c'est de prévoir des mécanismes des-

tinés à réduire la probabilité d'occurrence d'une telle situation.

Dans un système parallèle faiblement couplé, l'état d'occupation de la mémoire d'un nœud ne peut être connu qu'en interrogeant le nœud.

En général, le nœud observe son état et le transmet aux autres nœuds. La décision prise dépend alors de l'état d'occupation reçu. Si, entre-temps, l'état change, la valeur observée par le nœud qui exécute le remplacement n'est plus valide. Néanmoins, comme c'est l'information la plus récente qu'il possède, elle sera considérée jusqu'à ce que le nouvel état soit reçu.

A cause de la nature distribuée du système et de la nature dynamique du comportement de l'occupation de la mémoire, il est toujours possible qu'une page soit envoyée vers un nœud dont la mémoire est pleine. Le seuil de réception a été prévu pour réduire la probabilité d'envoi d'une page à un nœud qui ne peut pas la stocker. Pour illustrer cette situation, nous présentons le cas d'un système avec 2 nœuds x et y dans la figure 4.4.

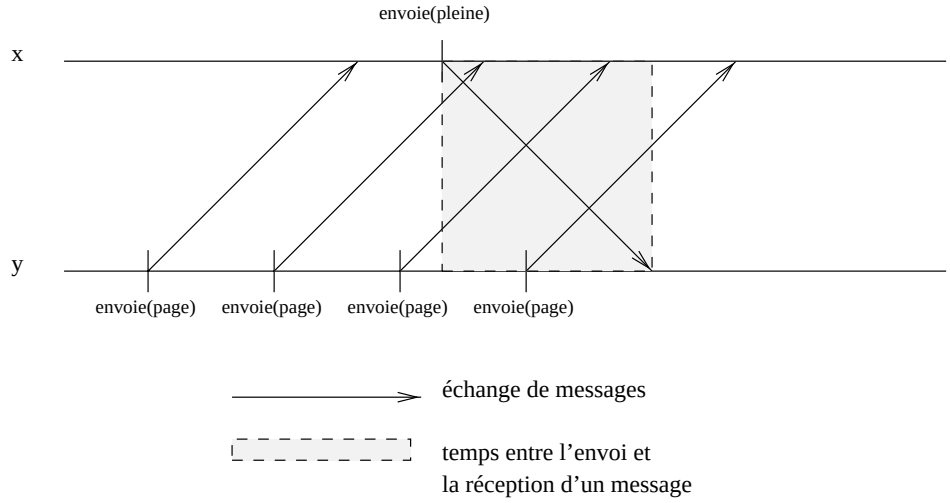


FIG. 4.4 – *L'utilité du seuil d'occupation*

Dès que le nœud y reçoit le message *pleine* du nœud x, il positionne l'indicateur correspondant à x dans sa table de charge à PLEINE et n'envoie plus de pages à ce nœud.

Néanmoins, les pages envoyées par y avant la réception du message *pleine* pourront être stockées sur x si le seuil est bien fixé en fonction du temps moyen de communication.

En général, plus les seuils de réception et de remplacement sont proches,

plus grande est la probabilité d'arrivée d'une page pour laquelle il n'y a pas de place. En revanche, un seuil d'occupation petit réduit l'occupation des mémoires distantes avec des pages empruntées et, par conséquent, augmente l'activité de transfert vers le disque.

L'efficacité de notre schéma de remplacement de pages dépend de la détermination d'un bon compromis entre les valeurs du seuil de remplacement et du seuil de réception.

Politique de mise à jour des informations

Dans notre système, l'état d'occupation de la mémoire est mesuré par le nombre de cadres de page occupés. Pour le répresenter, nous faisons une projection de l'occupation de la mémoire sur deux états: LIBRE et PLEINE. Si le nombre de cadres de page occupés est inférieur au seuil de réception, une projection est effectuée sur l'état LIBRE. Dans le cas contraire, l'état est dit PLEINE. Notre algorithme utilise donc une représentation n-états [KK92] où n est égal à 2.

La mise à jour des états peut se faire au moment du changement de l'état ou au moment de l'exécution de l'algorithme de remplacement. Dans le premier cas, l'information est disséminée sans la requête explicite du nœud récepteur. Dans le second cas, le nœud qui exécute l'algorithme de remplacement demande l'état des voisins au moment de l'exécution de la procédure `choisir_nœud`.

La deuxième approche présente deux incovénients. D'abord, le temps d'exécution de l'algorithme de remplacement est augmenté puiqu'il faut échanger au moins $2v$ messages, où v est le nombre de voisins directs. De plus, le coût de l'algorithme est principalement supporté par le nœud surchargé, c'est à lui de traiter les $2v$ messages alors que les nœuds voisins ne traitent que 2 messages.

Nous avons alors retenu la première approche. Le coût de cette approche répose sur le taux de mise à jour des états d'occupation. Comme chaque changement d'état génère v messages aux nœuds voisins, un taux important de mises à jour peut causer la congestion du réseau d'interconnexion.

La projection 2-états a été prévue pour réduire le nombre de communications inter-nœuds. Une situation limite peut encore se produire quand l'état d'occupation d'un nœud oscille fréquemment entre LIBRE et PLEINE. Pour éviter un nombre trop important de messages échangés quand une telle situation se produit, nous avons établi un intervalle de temps minimum pendant lequel aucun message d'occupation ne peut être envoyé.

La procédure de mise à jour des informations est montrée ci-dessous.

```
allouer_mémoire()
{
    cadre_page = chosir_cadre_libre();
    occupation_mémoire = occupation_mémoire + 1;
    SI (changement_état(occupation_mémoire))
    {
        SI (temps_changement_état ≥ TEMPS_MINIMUM)
            POUR TOUS voisins
                envoyer_message(voisin, état);
        SINON
            stocker(état_changé);
    }
}
```

4.1.6 Conclusion

L'algorithme de remplacement de pages proposé dans $\mathcal{DIVA}$ a servi à deux objectifs distincts. Le premier est de tirer profit du niveau intermédiaire créé par la mémoire virtuelle partagée. Ceci est réalisé à un coût faible de communication entre nœuds. Dès que les besoins de contrôle augmentent (les mémoires du voisinage sont pleines), l'algorithme se comporte comme un algorithme traditionnel de remplacement de pages.

Un autre objectif, moins évident, est d'empêcher que les pages pour lesquelles une opération d'écriture a été initiée mais n'est pas encore terminée soient choisies pour le remplacement.

Dans une machine virtuelle à modèles de cohérence multiples, nous pouvons avoir plusieurs modèles de cohérence relâchés. Une manière très courante d'implantation des modèles relâchés consiste à retarder la diffusion des modifications effectuées sur une page. Dans $\mathcal{DIVA}$, ces pages sont dans l'état instable EN_MODIFICATION et leur migration n'est pas souhaitable. Ainsi, l'algorithme de remplacement est nécessaire à notre système car il garantit qu'une telle page ne sera jamais remplacée.

4.2 Préchargement des pages

Pour la conception du module de préchargement de $\mathcal{DIVA}$ nous nous sommes intéressés aux aspects suivants: le choix de la page à précharger, la localisation

de la page préchargée et la politique de gestion des pages préchargées. Chacun de ses aspects est décrit en détail dans le reste de ce paragraphe.

4.2.1 Choix de la page à précharger

En général, c'est la page qui sera référencée dans un futur proche qui doit être préchargée. Toutefois, les méthodes automatiques proposées jusqu'à maintenant font des estimations soit très conservatives, soit inexactes. Ceci est dû en grande partie à la nature dynamique du comportement d'un programme (voir paragraphe 2.4.4).

Nous pensons que le programmeur est la personne qui connaît le mieux les caractéristiques des accès aux données réalisés par son application. Notre système lui laisse donc le choix de la page à précharger.

Le schéma de préchargement de pages est montré dans la figure 4.5.

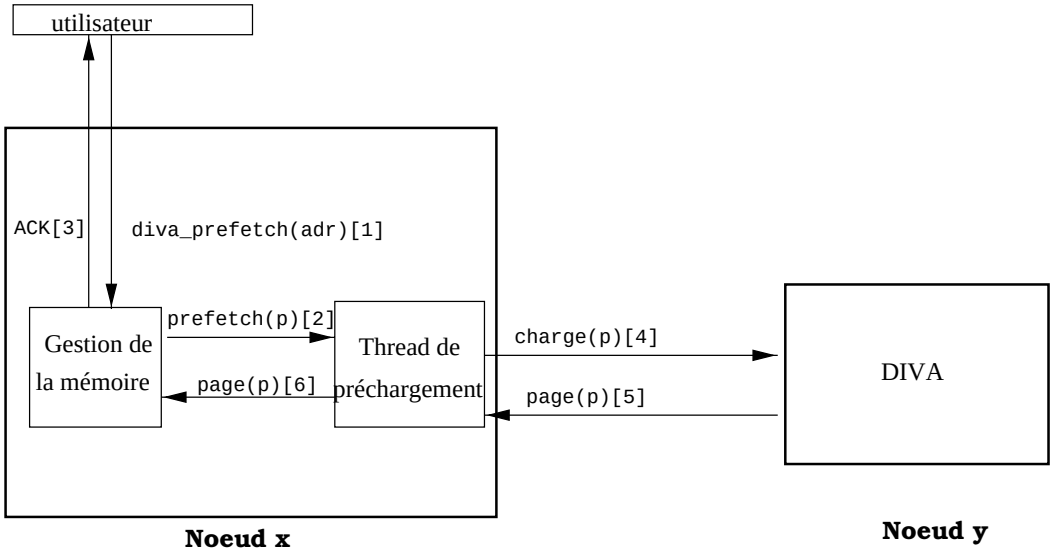


FIG. 4.5 – *Le schéma de préchargement de DIVA*

L'opération de préchargement des pages est initiée dans notre système par l'appel à la primitive `diva_prefetch(<adr>)`. L'appel à cette primitive fait qu'une requête de préchargement de la page qui contient <adr> soit mise dans une file de requêtes. L'application peut alors continuer son exécution.

L'opération de chargement d'une page requiert au moins deux messages entre nœuds: la requête de chargement de la page et l'envoi de la page demandée.

Les requêtes de préchargement pour la page p sont traitées par un flot d'exécution autonome (thread de préchargement). L'opération de chargement de p se déroule simultanément à l'exécution de l'application.

Dans ce contexte, un défaut de page pour la page p peut arriver avant que son préchargement s'effectue. Dans ce cas, la page p est rendue à l'application dès qu'elle arrive au nœud et l'opération de préchargement est annulée.

4.2.2 Localisation de la page préchargée

Si la page préchargée arrive au nœud sans y être réferencée entre-temps, il faut trouver de la place pour la stocker. En ce qui concerne la localisation de la page préchargée, deux approches peuvent être utilisées. La première approche place la page dans un cadre mémoire alloué à l'application utilisateur et la deuxième la place dans un tampon géré par le système à mémoire virtuelle partagée. Les avantages et inconvénients de ces deux approches sont discutés dans les paragraphes qui suivent.

La première approche nécessite que la page soit en quelque sorte verrouillée en mémoire pendant une période de temps Δ. Ceci sert à empêcher que les pages préchargées ne soient aussitôt choisies pour le remplacement puisqu'elles n'ont jamais été référencées. L'implantation de ce mécanisme de verrouillage est réalisée par des temporisateurs. Il faut un temporisateur matériel pour chaque page préchargée.

Pour la deuxième approche, l'utilisation de tampons à l'intérieur du système à mémoire virtuelle partagée entraîne les problèmes traditionnels de gestion de tampons. En plus, à chaque défaut de page, une recherche est d'abord menée dans ces tampons. Le temps de recherche doit être petit car il sera additionné au coût du défaut de page. De plus, la cohérence des pages contenues dans le tampon doit être assurée.

La première approche permet une implantation plus simple et plus performante du préchargement des pages. Néanmoins, l'accès à quelques mécanismes système tels que défauts de page provoqués et temporisateurs se fait nécessaire. C'est la raison pour laquelle nous n'avons pas retenu cette approche.

Les pages préchargées sont alors mises dans des tampons gérés par notre système selon la politique FIFO. Le nombre de tampons alloués au préchargement est un paramètre de DIVA et peut être défini par l'utilisateur. Pour que cette technique soit efficace, le nombre de tampons doit être petit.

4.2.3 Politique de gestion des pages préchargées

A chaque défaut de page, les tampons de préchargement sont consultés. Si la page s'y trouve, elle est aussitôt mise en mémoire et le tampon est libéré. Sinon, la page est chargée directement en mémoire.

Les tampons de préchargement sont aussi accédés à la suite des opérations de cohérence. Notre système garantit que les pages préchargées sont cohérentes avec le modèle de cohérence courant.

Les procédures qui implantent le préchargement sont les suivantes:

```
diva_prefetch(adr)
{
    page_ref = page(adr);
    stocker_requête_dans_file(page_ref);
}

traiter_requête()
{
    TANT QUE vrai
    {
      page_ref = retirer_requête_de_la_file();
      page_pré = charger_page(page_ref, gestionnaire(page_ref));
      SI (défaut_de_page(page_pré))
        rendre_la_page_à_l'application(page_pre);
      SINON
        mettre_dans_tampon(page_pré);
    }
}
```

Les tampons ont été implantés comme une simplement liste chaînée gérée selon la politique FIFO. Les opérations valides sur la liste de préchargement sont l'insertion et la suppression des pages.

Insertion de pages

Une page préchargée est toujours mise en fin de file. Ainsi, la page qui detient le plus grand temps de séjour dans un tampon est celle qui est en tête de file: $t_{page_0} \geq t_{page_1} \geq \cdots \geq t_{page_{n-1}}$ où t_{page_i} est le temps de séjour de la page i et n est le nombre maximal de tampons.

Suppression d'une page

On distingue trois cas selon la suppression est declenchée lors d'une référence, ou lorsque les tampons sont pleins ou lorsqu'une page a depassé le temps de séjour maximum.

Suppression d'une page référencée - Le transfert d'une page du tampon de préchargement vers la mémoire de l'application est effectué quand la page est réellement référencée par l'application. A ce moment là, le tampon alloué à la page est libéré. La page référencée peut occuper n'importe quelle position dans la file des pages préchargées.

Suppression d'une page (tampon plein) - La page dont le temps de séjour dans le tampon est le plus grand est celle que sera supprimée. En effet, il suffit de supprimer la page qui est en tête de liste.

Suppression d'une page (permanence $\geq$ maximum) - Le préchargement des pages additionne deux sortes de surcoût à notre système. Le premier type de surcoût est généré lors du préchargement d'une page qui ne sera jamais référencée. Ceci rajoute au système une opération inutile de chargement de page.

Le deuxième type de surcoût concerne toutes le pages préchargées. D'abord, le temps de recherche d'une page dans les tampons de préchargement est rajouté au coût du défaut de page. En plus, les opérations de cohérence sont aussi exécutées sur les pages préchargées. Le maintien des pages dans les tampons est alors coûteux au système.

Pour réduire ce coût, nous avons établi un temps maximum Δ de permanence des pages dans les tampons. Au-delà de ce temps, la page qui n'a pas encore été référencée est supprimée.

4.2.4 Evaluation de l'algorithme

Afin d'évaluer notre mécanisme de préchargement de pages, nous avons analysé le comportement d'une même application dans deux situations différentes. Pour le premier cas, une politique classique de chargement de pages à la demande est utilisée. Pour le second cas, l'application se sert du mécanisme de préchargement de pages offert par notre serveur.

L'application étudiée est une application simple qui accède aux données d'une manière exclusivement séquentielle. La chaîne de référence aux pages lors d'une

exécution quelconque de l'application est: $page_1$, $page_2$, ..., $page_n$.

Nous montrons dans la figure 4.6, le code de l'application quand la politique de chargement de pages appliquée est le chargement à la demande (*cd*) et quand la politique appliquée est le préchargement de pages (*pc*).

Algorithmes

Chargement à la demande (cd):	Prechargement (pc):

x=1;
.
.
y=4;
.
.
z=7;
	diva_prefetch(x);
	diva_prefetch(y);
	diva_prefetch(z);
	x=1;
	.
	.
	y=4;
	.
	.
	z=7;

FIG. 4.6 – *Les politiques de chargement des pages*

Afin d'initier l'opération de préchargement de la page qui contient la donnée x, l'utilisateur se sert de la primitive `diva_prefetch(x)`.

Ces deux algorithmes seront évalués selon trois critères: le temps d'exécution de l'application, le nombre de messages échangés lors de l'exécution de l'application et la mémoire nécessaire pour stocker les pages manipulées par l'application.

Temps d'exécution

Soit `t(a,pc)` le temps d'exécution de l'application a quand la politique de préchargement est utilisée et soit `t(a,cd)` le temps d'exécution de l'application a quand la politique de chargement à la demande est utilisée. L'objectif principal des mécanismes de préchargement des pages est de réduire le temps d'exécution d'une application. Ainsi, `t(a,pc)` doit être plus petit que `t(a,cd)`. Pour y arriver, le mécanisme de préchargement essaye de réduire le délai causé par l'opération de chargement de pages. Ce délai est important car chaque fois qu'une page non

résidente en mémoire est référencée, l'application reste bloquée jusqu'à la fin de l'opération de chargement.

Nous présentons par la suite des éléments pour l'évaluation du coût en temps d'exécution de notre algorithme de préchargement de pages ($\mathcal{C}_{cd}$) et de l'algorithme de chargement à la demande ($\mathcal{C}_{pc}$).

$\mathcal{C}_{cd} = pf$
- p = coût moyen d'un défaut de page à distance
- f = taux de défaut de page

$\mathcal{C}_{pc} = pm + p'h + C'$
- m = taux d'échec des accès au tampon de préchargement
- p' = coût moyen d'une réussite d'accès au tampon de préchargement
- h = taux de réussite des accès au tampon de préchargement
- C' = coût additionnel du préchargement

En général, $p' \ll p$. Le coût du préchargement est alors déterminé par m et C' [SC93]. Par la suite, nous analysons l'impact de ces deux paramètres.

Dans notre mécanisme de préchargement, c'est l'utilisateur qui détermine la page à précharger car c'est lui qui connaît le mieux la séquence des références aux données manipulées par son programme. Nous supposons alors que le taux de bonnes décisions de préchargement est important. Dans ce contexte, nous pouvons dire que, pour un tampon de préchargement de taille infinie et pour un temps maximum de séjour dans les tampons infini, m est petit.

Néanmoins, dans un système réel, les tampons de préchargement occupent physiquement la même mémoire que les cadres de pages. Une augmentation de la taille des tampons entraîne une réduction proportionnelle du nombre de cadres de pages. Un nombre petit de cadres de page augmente le taux de remplacement et, par conséquent, augmente le temps d'exécution de l'application. Ainsi, nous devons trouver un compromis entre ces deux facteurs.

Nous avons déterminé un temps maximum de séjour (Δ) dans les tampons

pour réduire le coût des opérations de cohérence sur les pages préchargées. Sans ce mécanisme, une page préchargée qui n'est jamais référencée peut rester longtemps dans les tampons de préchargement et beaucoup d'opérations inutiles de cohérence peuvent être exécutées pour cette page. L'estimation de Δ est aussi très importante pour l'efficacité de notre mécanisme de préchargement. Un Δ petit peut causer la suppression de la page du tampon de préchargement avant que la référence à cette page ne soit générée. La probabilité d'exécution d'opérations de cohérence additionnelles augemente avec un Δ trop important.

Pour resumer, nous avons essayé de maintenir petit le taux d'échecs du tampon de préchargement (m) en laissant à l'utilisateur le choix de la page à précharger. Toutefois, pour que m soit effectivement petit, il faut une bonne estimation de la taille des tampons de préchargement et du temps maximum de permanence des pages dans les tampons.

Dans notre cas, le coût C' est le coût de l'exécution de la primitive `diva_prefetch`. Ce coût comprend un appel à notre serveur. Les opérations exécutées par $\mathcal{DIVA}$ lors de l'appel à `diva_prefetch` sont très simples. Elles se résument à placer une requête de préchargement dans une file. Nous avons un thread autonome qui exécute le préchargement de pages en parallèle avec l'exécution de l'application. Nous pouvons conclure que C' est alors très faible.

Messages échangés

Dans ce paragraphe, nous évaluons le nombre de messages entre nœuds échangés lors de l'exécution d'une application pour le cas du chargement à la demande $\mathcal{M}_{cd}$ et du préchargement $\mathcal{M}_{pc}$.

$\mathcal{M}_{cd} = c_h + e + c$
- c_h = nombre de messages nécessaires pour le chargement des pages
- e = nombre de messages échangées entre les processus de l'application
- c = nombre de messages de cohérence

$\mathcal{M}_{pc} = c'_h + e + c'$
- c'_h = nombre de messages nécessaires pour le chargement des pages
- e = nombre de messages échangées entre les processus de l'application
- c' = nombre de messages de cohérence

Puisque le nombre de messages échangés par les processus qui composent

l'application est le même pour les deux cas, nous allons analyser uniquement les messages de chargement et les messages de cohérence.

Pour le chargement à la demande, le nombre de messages de chargement échangés est $c_h = m_c f$, où m_c est le nombre moyen de messages par opération de chargement et f le taux de défauts de pages.

Pour notre mécanisme de préchargement, le nombre de messages de chargement échangées est donnée par la formule suivante: $c'_h = m_c(m + h + a)$. Le premier facteur correspond aux opérations de chargement au moment d'un défaut de page. Le deuxième facteur modélise les opérations de chargement de pages effectuées au moment de l'exécution d'une primitive de préchargement qui ont été utiles, c'est à dire, les pages préchargées ont été référencées dans un moment postérieur et elles se trouvaient encore dans les tampons. Le troisième facteur $(m_c a)$ modélise les opérations de préchargement inutiles. Soit les pages préchargées ne sont jamais référencées, soit les pages préchargées sont référencées à un moment où elles ne se trouvent plus dans les tampons.

Pour une situation idéale de l'exécution d'une application sur une politique de préchargement, $m + h = f$ et $a = 0$. Dans ce cas, le nombre de messages échangés par les deux politiques est le même. Nous pouvons conclure, alors, que $c_h \leq c'_h$.

La cohérence des pages est assurée pour toutes les pages qui se trouvent sur la mémoire locale d'un nœud. Pour le cas du chargement à la demande, les pages qui se trouvent sur un nœud sont les pages référencées précédemment pour l'application. Pour le cas du préchargement, les pages qui se trouvent sur un nœud sont toujours les pages qui y ont été référencées mais aussi les pages qui y ont été préchargées. Comme les opérations de cohérence sont en général exécutées sur toutes les copies d'une page, on a $c \leq c'$.

Cette analyse nous montre que le nombre de messages de chargement échangés par notre mécanisme de préchargement est plus grand ou égal au nombre de messages de chargement échangés par une politique de chargement à la demande.

Mémoire

Le coût en mémoire de notre mécanisme de préchargement est plus élévé que le coût en mémoire de l'algorithme de chargement à la demande. Dans les deux cas, un cadre de page est réservé pour toute page référencée.

Le mécanisme de préchargement fait que quelques pages additonnelles soient aussi stockées en mémoire locale. Ceci fait que les besoins en mémoire du mécanisme de préchargement sont plus grandes que les besoins en mémoire du chargement à la demande.

4.2.5 Conclusion

Nous avons proposé ici un schéma de préchargement de pages qui se sert des annotations de l'utilisateur. D'après l'analyse de ce schéma, nous avons observé que son coût en mémoire et en nombre de messages échangés est plus élevé que celui du chargement à la demande.

Pour que notre schéma de préchargement soit efficace dans le but de réduire le temps d'exécution d'une application, il faut observer la condition suivante: les pages préchargées doivent être référencées par l'application dans un futur proche. Puisque nous considérons que les tampons de préchargement ont une taille finie et petite, un laps de temps trop important entre l'opération de préchargement et la référence à la page peut entraîner la suppression de la page des tampons avant l'occurrence de la référence.

Chapitre 5

Mise en Œuvre d'un serveur $\mathcal{DIVA}$

Afin de valider la correction et d'évaluer les performances des mécanismes proposés dans les deux chapitres précédents, deux approches sont généralement adoptées: la simulation et la réalisation d'un prototype.

La première approche consiste à analyser le comportement des mécanismes à l'aide des outils de simulation. Nous avons écarté cette approche pour deux raisons. Tout d'abord, le comportement des mécanismes proposés est tellement complexe qu'il nous semble improbable de trouver un outil de simulation capable de réproduire la plupart des caractéristiques et propriétés que nous voulons analyser. De plus, bien que les mécanismes proposés traitent des problèmes différents, ils interfèrent mutuellement. L'analyse de chaque mécanisme isolé peut donc conduire à des conclusions qui ne sont pas vraies dans un système réel.

Nous avons donc choisi la deuxième approche, qui consiste à implanter un prototype de la machine virtuelle décrite dans les chapitres précédents. Le prototype nous a servi d'abord à montrer que l'implantation des mécanismes proposés, en particulier le module de gestion de modèles, est possible et relativement simple. Ensuite, l'existence d'un prototype nous a permis d'observer le comportement des applications parallèles sur plusieurs modèles de cohérence. Ceci a permis la validation de notre hypothèse initiale selon laquelle le choix du modèle de cohérence doit prendre en compte les caractéristiques des accès aux données.

Puisque nous avons conçu $\mathcal{DIVA}$ comme une machine virtuelle du micronoyau ParX (voir paragraphe 3.3), le choix naturel d'implantation est le système d'exploitation PAROS. Néanmoins, ce système a été implanté dans une machine à base de transputers, le Supernode [Wai91]. Les transputers sont des processeurs très simples qui n'ont pas de support matériel pour la gestion de la mémoire virtuelle. Ceci rend impraticable l'utilisation de la pagination dans les systèmes

à base de transputers.

Afin de maintenir l'implantation du prototype de $\mathcal{DIVA}$ la plus proche possible de sa conception, nous avons opté pour son implantation dans un autre système d'exploitation. Ainsi, dans un premier temps, nous avons implanté $\mathcal{DIVA}$ sur une machine Intel/Paragon car c'est une machine parallèle qui offre le support matériel pour la pagination. Comme support logiciel, nous avons choisi le micronoyau Mach car plusieurs de ses caractéristiques ressemblent au micro-noyau ParX.

Dans ce chapitre, nous décrivons la conception d'un serveur qui implante les mécanismes proposés par $\mathcal{DIVA}$ sur la machine parallèle Intel Paragon™. D'abord, nous décrivons l'architecture proposée pour le serveur. Ensuite, nous présentons les principales caractéristiques de la machine, du micronoyau Mach et du système d'exploitation Paragon OSF/1. La mise en œuvre d'un serveur $\mathcal{DIVA}$ est présentée à la fin du chapitre.

5.1 L'architecture du Serveur

La fonctionalité d'une mémoire partagée peut être offerte aux utilisateurs de plusieurs manières. Le premier problème auquel nous nous sommes posé pour l'implantation des mécanismes décrits précédemment concerne le schéma d'implantation du système à mémoire virtuelle partagée.

Le paragraphe 2.5 montre que l'éventail des choix pour le schéma d'implantation d'un système à MVP est très vaste et diversifié.

Le critère qui a guidé notre choix du schéma d'implantation est celui de la portabilité. Selon ce critère, nous avons écarté l'implantation par matériel et celle au niveau du noyau système. Nous avons choisi l'implantation d'un serveur et non l'implantation sous forme de bibliothèque ou d'un environnement de programmation pour deux raisons:

1. Notre système doit offrir la mémoire virtuelle partagée à toutes les applications qui le souhaitent, indépendamment du langage de programmation utilisé.

2. Notre système doit être capable de coexister avec autres mécanismes de communication entre processus comme, par exemple, l'échange de messages.

Ayant choisi le niveau d'implantation, il faut s'intéresser à l'organisation du serveur. Un serveur est un automate qui reçoit des requêtes, exécute des services

associés aux requêtes reçues et envoie éventuellement une réponse. Les performances du serveur dépendent du temps passé dans le traitement des requêtes et l'organisation du serveur joue un rôle très important dans la réduction de ce temps.

Nous considérons par la suite trois types traditionnels d'organisation: centralisée, distribuée monolithique et distribuée multiflots.

Un serveur centralisé se trouve sur un nœud unique. Son activation est faite par des appels de procédures distantes. Les sites autres que le site serveur reçoivent les requêtes des processus utilisateur et les acheminent au serveur de mémoire virtuelle partagée. Le processus utilisateur reste bloqué jusqu'à l'arrivée de la réponse à sa requête.

Bien que l'implantation d'un tel serveur soit très simple, ses performances ne sont pas bonnes. Le serveur et les liens qui permettent d'y accéder deviennent surchargés à mesure que le nombre de processeurs et requêtes du système augmentent. Pour cette raison, une solution centralisée de ce type est à écarter.

Dans l'approche distribuée monolithique, une copie du code du serveur se trouve sur chaque processeur. Les requêtes sont d'abord adressées au serveur local, qui collabore avec les serveurs à distance pour les traiter.

Dans cette organisation, la tâche de gestion de la mémoire virtuelle partagée est distribuée parmi les processeurs du système. Le traitement des requêtes reste pourtant séquentiel: une requête n'est servie que quand le traitement de la requête précédente est terminé.

L'approche distribuée multiflots est utilisée pour apporter de la concurrence au traitement des requêtes. De façon similaire à l'approche précédente, une copie du serveur est placée sur chaque processeur. Cependant, chacune de ces copies abrite plusieurs flots d'exécution (*threads*). Ceci permet le traitement de plusieurs requêtes à la fois. La sérialisation des requêtes n'est réalisée que quand cela est nécessaire, par exemple pour les requêtes d'une même page.

L'obtention de cette concurrence a évidemment un coût associé. D'abord, il faut définir une politique pour associer les requêtes aux flots d'exécution qui les serviront. Ensuite, il faut protéger les structures de données du serveur avec des mécanismes de synchronisation. En plus de compliquer la conception du serveur, les besoins de synchronisation peuvent ralentir son exécution à cause de la contention d'accès aux ressources partagées.

Les flots multiples d'exécution sont eux-mêmes organisés selon plusieurs critères. Les deux organisations les plus répandues sont l'organisation maître/esclave et l'organisation par fonction.

Dans l'approche maître/esclave, il existe un flot d'exécution maître respon-

sable du traitement initial de la requête. C'est à ce flot de décider quel flot esclave donnera suite au traitement. Dans cette organisation, n'importe quel flot peut servir n'importe quelle requête.

Le deuxième approche organise les flots par fonction. Dans ce cas, un flot ou un sous-ensemble de flots est responsable du traitement des requêtes de fonctions spécifiques.

L'approche que nous avons retenue essaye de trouver un compromis entre la simplicité d'implantation et l'efficacité. Ainsi, le serveur $\mathcal{DIVA}$ est organisé selon l'approche distribuée multiflots et s'exécute sur tous les nœuds du système.

Il peut être activé de trois manières: par l'invocation des primitives utilisateur; par le noyau local, à la suite des défauts de page; et par les serveurs à distance [BM94b].

Le serveur $\mathcal{DIVA}$ est composé de plusieurs modules qui interagissent pour implanter les fonctionnalités offertes aux applications. Le fonctionnement interne du serveur est montré dans la figure 5.1.

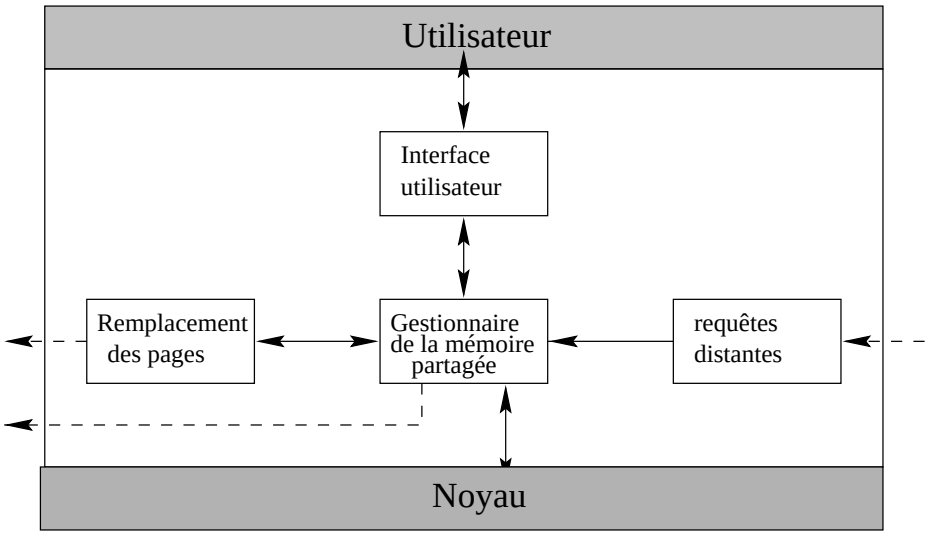


FIG. 5.1 – *La structure du serveur*

Dans chaque nœud, le serveur $\mathcal{DIVA}$ est constitué essentiellement par quatre flots d'exécution: le gestionnaire de la mémoire partagée, le service de remplacement des pages, l'interface avec l'utilisateur et le gestionnaire de requêtes distantes. La communication entre les flots d'exécution est réalisée par le partage de variables communes.

Par la suite, nous décrivons chaque module d'une manière concise.

L'interface utilisateur est le module chargé du dialogue entre le serveur et l'utilisateur. Il reçoit les appels de primitives, vérifie la syntaxe de la primitive et ensuite active le gestionnaire de la mémoire partagée.

Le gestionnaire de la mémoire partagée est le module le plus important du serveur. Il est chargé de l'implantation des modèles de cohérence de la mémoire, de la gestion des pages et réalise les mécanismes de synchronisation. Son activation est faite par le noyau local, lors d'un défaut de page; par l'interface utilisateur, lors de l'exécution d'une primitive par l'application; ou par un nœud distant, pour répondre à une requête.

Le module de remplacement des pages est activé par le gestionnaire de la mémoire partagée lorsque la mémoire locale devient surchargée. L'algorithme choisit donc une page à supprimer et informe le gestionnaire de la mémoire partagée de son identité. Si la page à supprimer a été modifiée, celle-ci est envoyée à un nœud distant.

Le gestionnaire de requêtes distantes est responsable de la réception et du traitement des requêtes des nœuds distants.

La figure 5.2 montre la syntaxe et la fonction de l'ensemble des primitives offert par $\mathcal{DIVA}$ ainsi que le module qui les implante.

Dans la suite, nous présentons chacun de ces modules. Nous y précisons notamment les primitives définies.

5.1.1 L'interface utilisateur

Le serveur $\mathcal{DIVA}$ met à disposition de l'utilisateur un ensemble de primitives qui permettent le dialogue entre l'utilisateur et le serveur au moment de l'exécution de l'application. Le fonctionnement de ces primitives a été présenté lors de la description des mécanismes originaux offerts par la machine virtuelle $\mathcal{DIVA}$ dans les chapitres 3 et 4.

Le module interface utilisateur exécute l'algorithme suivant:

```
tant que (VRAI)
  primitive = recevoir_primtive();
  si syntaxe_correcte(primitive)
    reponse = appeler_module(primitive);
    répondre_utilisateur(reponse);
  sinon
    répondre_utilisateur(ERREUR);
```

Comme on peut le voir, ce module ne s'occupe pas du traitement des primi-

SYNTAXE	DESCRIPTION	MODULE
model_set = diva_set_memory_model(model_wanted)	L'UTILISATEUR IDENTIFIE LE MODELE DE COHERENCE SOUHAITE POUR L'EXECUTION DE L'APPLICATION	Gestion de la cohérence
lock_number = diva_get_lock()	L'UTILISATEUR DEMANDE UN VERROU	Gestion de la synchro
diva_remove_lock(lock_number)	L'UTILISATEUR INFORME LE SERVEUR QUE LE VERROU NE SERA PLUS UTILISE	Gestion de la synchro
diva_lock(lock_number)	OPERATION DE VERROUILLAGE	Gestion de la cohérence Gestion de la synchro
diva_unlock(lock_number)	RELACHEMENT DE VERROU	Gestion de la cohérence Gestion de la synchro
diva_prefetch(adresse)	PRECHARGEMENT D'UNE PAGE	Gestion des pages
diva_map(region)	PROJECTION D'UNE REGION PARTAGEE	Gestion des pages
diva_unmap(region)	DEFAIT LA PROJECTION D'UNE REGION	Gestion des pages

FIG. 5.2 – *L'ensemble de primitives du serveur*

tives. Il consiste en une boucle infinie qui reçoit une primitive, vérifie sa syntaxe et appelle le module correspondant.

5.1.2 Le Gestionnaire de la mémoire partagée

Le gestionnaire de la mémoire virtuelle partagée est le module responsable de l'exécution du programme selon le modèle de cohérence mémoire courant. Il cumule les fonctions d'un gestionnaire traditionel de la mémoire virtuelle (chargement, remplacement et localisation des pages) et les fonctions d'un gestionnaire de la mémoire partagée (cohérence et synchronisation). Sa structure est montrée sur la figure 5.3.

Le module de gestion des défauts de page est le module qui s'occupe des premiers pas du traitement du défaut de page. Il est activé par le noyau Mach et demande la page au module de gestion des pages.

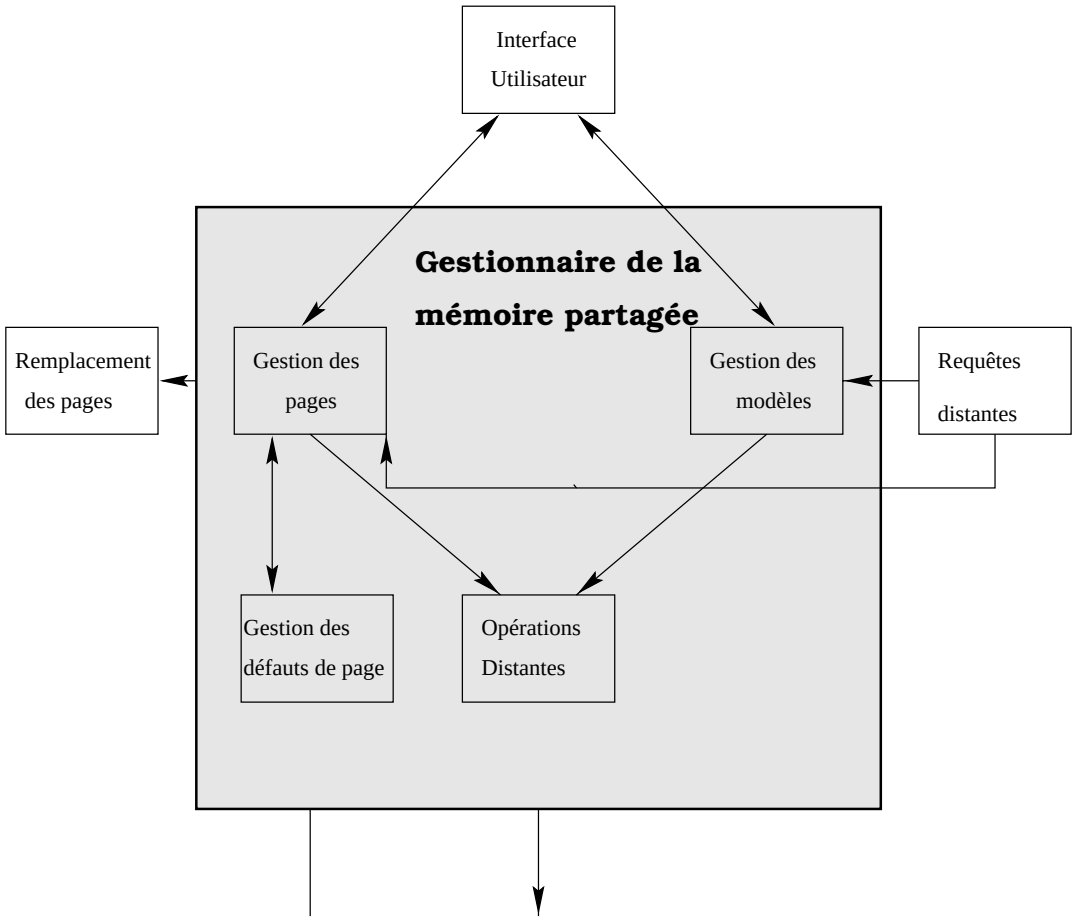


FIG. 5.3 – *La structure du gestionnaire de la mémoire virtuelle partagée*

Le module d'opérations distantes est le responsable de l'envoi de messages aux nœuds distants.

Le module de gestion des modèles implante les mécanismes décrits dans le chapitre 3.

Nous détaillons ici que le fonctionnement du module de gestion des pages.

Gestion des pages

Dans notre serveur, chaque bloc de mémoire partagée est vu comme un objet. Les objets sont divisés en pages de taille fixe. Le module de gestion des pages s'occupe de la gestion de la table des objets et de la table des pages ainsi que des opérations de chargement de pages.

Selon la nomenclature traditionnelle, chaque objet correspond à un segment. A chaque objet défini dans le système correspond une entrée dans la table d'objets.

Cette entrée contient des informations qui concernent l'objet en tant qu'entité. A chaque objet est associée une table de pages. Les champs qui composent une entrée de la table d'objets sont au nombre de cinq (cf. figure 5.4).

nom	NOM DE L'OBJET
etat	ETAT DE L'OBJET
pend	OPERATIONS BLOQUEES
verrou	VERROU DE L'OBJET
ptable	POINTEUR VERS LA TABLE DE PAGES

FIG. 5.4 – *Structure d'une entrée de la table d'objets*

L'objet doit être référencé par son <nom>. Les opérations `diva_map(<nom>)` et `diva_unmap(<nom>)` font la mise à jour du champ <etat> de l'objet (*projeté* et *libre*, respectivement). Toute modification d'une entrée de la table d'objets est précédée d'une opération de verrouillage sur <verrou>. Le <verrou> est relâché à la fin de la mise à jour. Une opération sur un objet reste bloquée sur <pend> jusqu'à ce que le verrou lui soit accordé.

page	NUMERO DE LA PAGE
adresse	ADRESSE PHYSIQUE SUR LEQUEL LA PAGE EST MAPPEE
modif	INDICATEUR DE MODIFICATIONS
etat	ETAT DE LA PAGE
pend	OPERATIONS BLOQUEES
copyset	NOEUDS QUI POSSEDENT LA PAGE
verrou	VERROU DE LA PAGE
gel	INDICATEUR DE VERROUILLAGE EN MEMOIRE

FIG. 5.5 – *Structure d'une entrée de la table des pages*

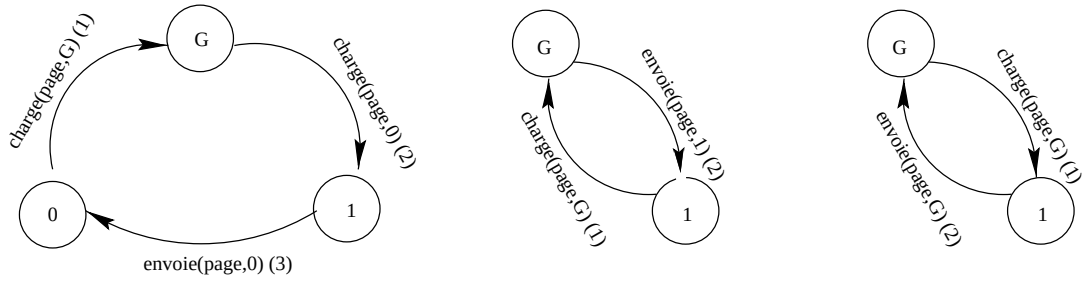
Chaque accès à l'objet est traduit en un accès à une de ses pages. Le serveur local est responsable de la gestion des pages qui se trouvent dans sa mémoire

et des pages dont il est le gestionnaire. La correspondance entre un nœud gestionnaire et une page est réalisée par l'algorithme distribué fixe décrit dans le paragraphe 2.4.1. A chaque page est associée une structure à 8 champs, montrée sur la figure 5.5.

A l'intérieur de l'objet, chaque page est référencée par son <numéro>. Chaque opération sur une page est précédée par une opération de verrouillage sur <verrou> et succédée par une opération de relâchement de ce même <verrou>. Le champ <adresse> donne la localisation physique de la page. Le champ <modif> permet de saisir si la page a été modifiée. Une page peut être "gélée" <gel> dans la mémoire. Dans ce cas, elle ne peut être choisie par l'algorithme de remplacement ni invalidée par une opération de cohérence. La liste <pend> contient les opérations d'accès à la page qui ont été bloqués par son gestionnaire. Le <copyset> est l'ensemble de nœuds qui détiennent une copie de la page. L'<état> de la page est un parmi ces sept: FREE, READ, MREAD, WRITE, BORROWED, EN_MODIFICATION et EN_TRANSIT (voir paragraphe 4.1.3).

Chargement d'une page

Une page est demandée au serveur par l'intermédiaire d'un défaut de page. A la suite d'un défaut de page, un message `charge`(<page>,G) est envoyée au gestionnaire de la page. Le gestionnaire envoie la page au processeur sur lequel le défaut de page a eu lieu, si elle se trouve sur sa mémoire locale (`envoie`(<page>,G)). Sinon, un message `charge(page, copyset(page))` est envoyé à un processeur qui appartient à l'ensemble `copyset`(<page>). La page sera transférée directement de ce processeur au processeur demandeur par le message `envoie`(<page>, <demandeur>). Pour charger une page en mémoire il faut donc trois messages, sauf dans le cas particulier où le processeur demandeur est le gestionnaire de la page ou si le gestionnaire a une copie de la page (voir figure 5.6).



(a) Cas général *(b) Le gestionnaire détient une copie* *(c) Le gestionnaire est en défaut*

FIG. 5.6 – *Le protocole de chargement des pages*

Une optimisation a été conçue pour le cas où le gestionnaire est en défaut de page et attend que la page lui soit rendue. Si, entre-temps, une requête pour la même page arrive au gestionnaire de la page, la requête est bloquée dans une liste *pend(page)*. Au moment de l'arrivée de la page dans le gestionnaire, celle-ci est envoyée à tous les nœuds qui l'ont sollicitée. Ceci permet d'éviter un message par nœud en attente dans le protocole de chargement de pages (figure 5.7).

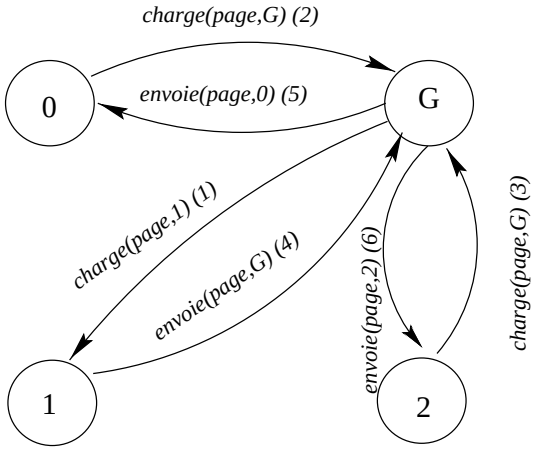


FIG. 5.7 – *Optimisation du protocole de chargement*

Le chargement d'une page se déroule toujours comme décrit. Néanmoins, le modèle de cohérence mémoire courant définit le moment où la page sera réelement rendue au processeur demandeur. Si, par exemple, le modèle de cohérence courant est la cohérence séquentielle, l'opération prendra plus de temps quand une page est sollicitée en écriture. En effet, en général, il faut invalider toutes les copies avant que l'accès en écriture soit accordé.

Le module de gestion des pages est aussi responsable de l'implantation des opérations de préchargement décrites dans le paragraphe 4.2.

5.2 La machine Paragon

Les machines Intel Paragon sont des machines parallèles sans mémoire commune composées de plusieurs nœuds connectés entre eux par un réseau d'interconnexion en grille à haut débit. La communication avec le monde extérieur est faite par l'intermédiaire des interfaces d'entrée/sortie (voir figure 5.8).

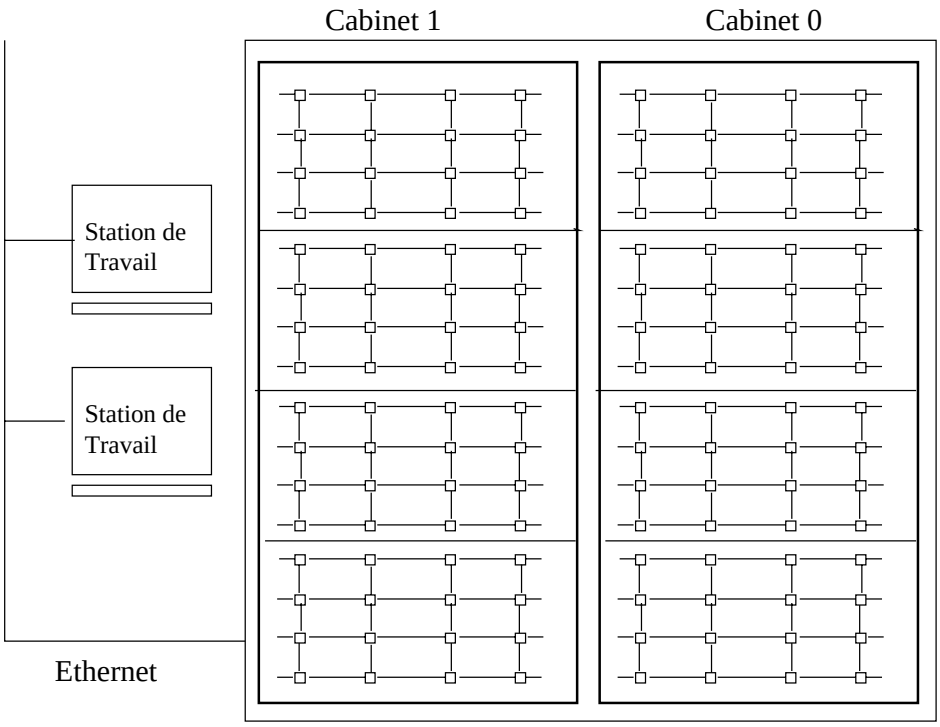


FIG. 5.8 – *La machine Intel Paragon*

Une machine Intel Paragon contient en général deux cabinets. Le cabinet 0 contient la station de diagnostic. Les autres cabinets comportent les nœuds, le réseau d'interconnexion en grille et les périphériques d'entrée/sortie. Un cabinet peut avoir jusqu'à 3 modules périphériques et 4 emplacements de cartes. Chaque emplacement de cartes a 16 slots pour les nœuds et un slot pour le moniteur de performance. Chaque module périphérique peut avoir jusqu'à 3 sous-systèmes RAID-3 ou des pilotes pour des bandes magnétiques [Int93].

La figure 5.9 illustre la structure interne d'un nœud de la machine Intel Paragon.

Chaque nœud est un multiprocesseur à mémoire partagée possédant deux ou trois microprocesseurs i860 à 50 Mhz, une mémoire locale et un dispositif de DMA. Les processeurs, la mémoire et le DMA sont connectés à un bus qui maintient la cohérence entre ces éléments. Parmi les processeurs i860, un est désigné pour le traitement des messages. Ce processeur exécute le code de traitement de messages du système d'exploitation tandis que les autres sont dédiés à l'exécution des applications parallèles.

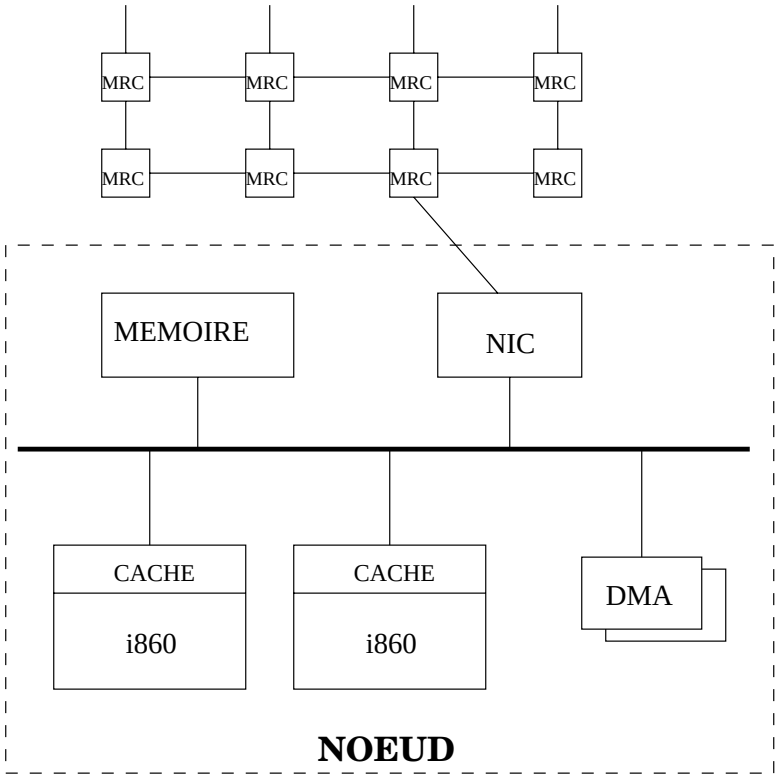


FIG. 5.9 – *La structure d'un noeud de la machine Paragon*

Chaque processeur i860 possède un cache d'instructions et un cache de données. Ces deux caches comportent des lignes de 32 octets et sont "2-way set associatives". Chaque processeur est doté d'un ensemble de tampons d'écriture qui peuvent traiter jusqu'à deux écritures simultanées.

Les nœuds sont connectés par un réseau bi-dimensionnel en grille dont le débit est 175 Mo/sec par lien dans chaque direction. Les MRC[1] implantent le routage "wormhole". La latence du MRC est de 40 nsec s'il n'y a pas de changement de dimension et 70 nsec dans le cas contraire.

Dans chaque nœud, le NIC[2] fait l'interface entre le MRC et la mémoire. Il possède deux files de 2 koctets: une pour le transfert et l'autre pour la réception des messages. En plus, plusieurs registres sont dédiés au contrôle des messages. Les dispositifs DMA font le transfert entre la mémoire et les files du NIC.

La machine Intel Paragon qui a été utilisée pour la mise en œuvre du serveur $\mathcal{DIVA}$ est celle de l'IRISA, Rennes. C'est une machine Intel Paragon XP/S A4E avec trois processeurs i860 qui font la liaison avec l'extérieur. Le système

1. MRC - Mesh Routing Chips
2. NIC - Network Interface Chip

d'entrée/sortie est composé de 3 systèmes RAID[3], chacun avec 5 disques de 1.4 Gigaoctets. La machine dispose de 56 nœuds de calcul. Chaque nœud de calcul est composé de deux processeurs i860 dont un est dédié exclusivement à la communication et d'une mémoire vive de 16 Mo. La taille de la page est 8koctets.

L'architecture logicielle des machines Intel Paragon est montrée dans la figure 5.10.

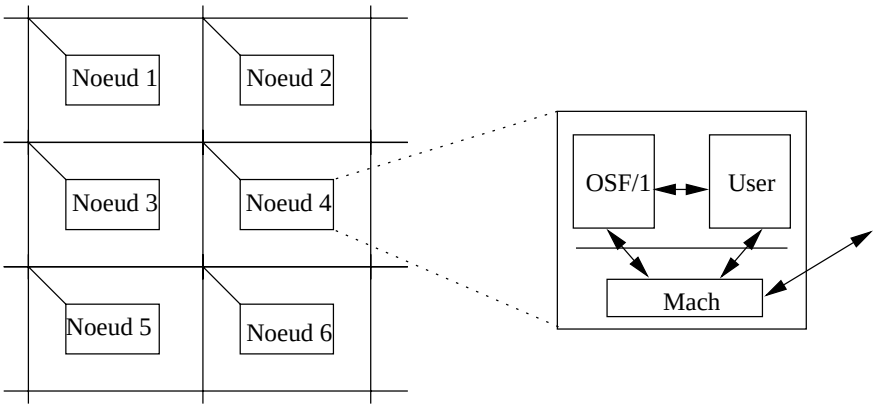


FIG. 5.10 – *Le support logiciel des machines Intel Paragon*

Le système Mach [R+88] est le micronoyau qui sert de base pour l'implantation du système d'exploitation des machines Intel Paragon. Le noyau Mach qui tourne sur la Paragon est implantation NMK13 de NORMA Mach [Tri95]. Le système Paragon OSF/1 tourne au-dessus de Mach. A l'IRISA, le système OSF/1 utilisé est l'OSF/1 version 1.0.4 (serveur 1.3, patchlevel 1.3.5). Par la suite, nous décrivons les principales caractéristiques de ces deux systèmes.

5.3 Le système Mach

Le système Mach est un système d'exploitation multiprocesseur développé à Carnegie-Mellon Unversity [R+88]. C'est un micronoyau qui offre des mécanismes de base sur lesquels plusieurs systèmes d'exploitation peuvent être bâtis. Les caractéristiques principales de Mach sont:

- support pour l'exécution de plusieurs threads dans un même espace d'adressage;
- gestion de la mémoire virtuelle indépendante de l'architecture;
- un mécanisme extensible de communication entre processus;

3. RAID - Redundant Array of Independent Disks

– intégration entre la communication et la mémoire virtuelle.

Mach est fondé sur cinq concepts: la tâche, le thread, la porte, le message et l'objet mémoire.

Une *tâche* est un environnement d'exécution. C'est l'unité d'allocation de ressources. A une tâche est associé un espace d'adressage.

Un *thread* est un flot d'exécution disposant d'un compteur de programme à l'intérieur d'une tâche. Tous les threads d'une tâche partagent l'accès à toutes les ressources allouées à la tâche.

Une *porte* est un canal de communication n vers 1. Une entité est référencée dans Mach par l'intermédiaire de sa porte. Les opérations exécutées sur les portes sont l'envoi et la réception des messages.

Un *message* est un ensemble de données utilisées dans la communication entre les threads.

Un *objet mémoire* est un ensemble de données gérées par un serveur. L'objet mémoire est projeté sur l'espace d'adressage d'une tâche.

L'action de créer une tâche, un thread ou un objet mémoire retourne au thread appelant des droits d'accès à la porte qui représente le nouvel objet.

Plusieurs mécanismes ont été prévus dans Mach pour la conception des serveurs. Un serveur Mach exécute une boucle de service qui reçoit des messages, effectue le traitement correspondant au message reçu et envoie la réponse à l'application.

La création d'un serveur requiert trois étapes [Ope92b]:

– La spécification des messages envoyés par l'application au serveur et les réponses à chaque message,

– La définition des procédures de traitement associées à chaque message,

– La définition du code d'initialisation du serveur.

Pour être connu des tâches utilisateur, le serveur doit s'enregistrer en tant que tel. Ceci est fait par l'appel à `netname_check_in`, qui réalise l'association entre le nom du serveur et la porte qui lui est attribuée. Afin de localiser un serveur, l'application doit faire appel à `netname_look_up`.

L'originalité de Mach repose surtout sur la conception de son gestionnaire de mémoire virtuelle. Le noyau Mach permet que les tâches utilisateur contrôlent des régions de l'espace d'adressage virtuel. Les objets mémoire sont des entités

Mach qui représentent ces régions. La tâche qui gère l'objet mémoire est en effet un serveur appelé *gestionnaire de mémoire*.

Le gestionnaire de mémoire interagit avec le noyau et l'utilisateur à travers le protocole pré-défini décrit dans [Ope92a]:

1. Une tâche quelconque effectue la projection d'un objet mémoire sur son espace d'adressage à travers la primitive `vm_map`. A ce moment, la tâche spécifie le gestionnaire de mémoire qui sera le responsable de la gestion de l'objet.

2. A la suite de l'exécution de `vm_map`, le noyau envoie le message `memory_object_init` au gestionnaire de mémoire correspondant. Le message contient l'adresse de la porte associée au noyau. A ce moment, le gestionnaire peut effectuer quelques initialisations. Le gestionnaire répond au noyau avec le message `memory_object_ready`.

3. La première référence effectuée par la tâche à l'objet mémoire cause un défaut de page. Ceci génère une exception qui active le noyau. Le noyau envoie un message de requête de page au gestionnaire de mémoire responsable de cet objet (`memory_object_data_request`). Le gestionnaire peut répondre au noyau en utilisant une des primitives suivantes:

   - `memory_object_data_supply`: la page sollicitée est rendue au noyau.
   - `memory_object_data_error`: la page ne peut pas être rendue à cause d'une erreur.
   - `memory_object_data_unavailable`: la page doit être générée par le noyau. En général, elle est remplie de zéros.

   Les droits d'accès sont aussi transmis au noyau par une de ces trois primitives. A la fin de l'exécution de ce protocole, la page est placée sur la mémoire de l'utilisateur et peut y être référencée.

   Le protocole de traitement du défaut de page est illustré par la figure 5.11.

4. Quand le noyau décide de remplacer une page, il l'envoie au gestionnaire de mémoire dans un message `memory_object_data_return`. Le gestionnaire de mémoire reçoit la page, il décide où la placer et l'envoie à la destination choisie.

5. Quand l'application veut référencer la page d'une manière interdite par les droits d'accès courants, le noyau envoie le message `memory_object_data_unlock` au gestionnaire mémoire. Si le gestionnaire décide d'accorder les droits d'accès sollicités, il répond avec un message `memory_object_lock_request`. Ce message doit contenir les nouveaux

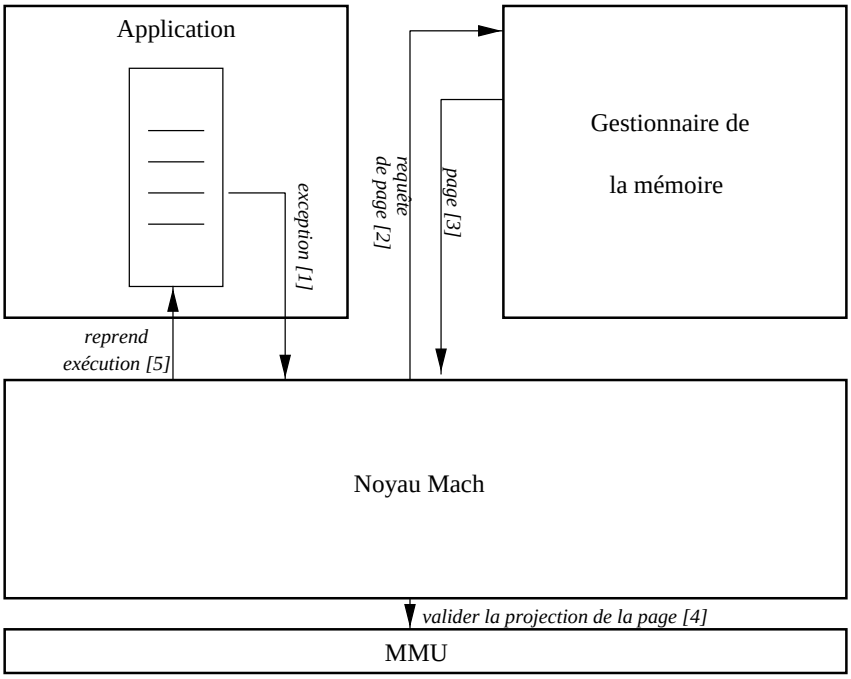


FIG. 5.11 – *Le traitement du défaut de page*

droits d'accès. Dès que le noyau met à jour les droits d'accès, il envoie le message `memory_object_lock_completed` au gestionnaire.

6. Si le gestionnaire décide de restreindre les droits d'accès à une page, il initie la conversation avec le noyau Mach en lui envoyant le message `memory_object_lock_request`. A partir de là, tout se passe comme dans le cas précédant.

7. Quand toutes les tâches suppriment les projections de l'objet mémoire faites dans leur espace d'adressage, le noyau envoie le message `memory_object_terminate` au gestionnaire. Par ce message, la communication entre le noyau et le serveur s'achève.

5.4 Le système Paragon OSF/1

Le système OSF/1 AD a été conçu par l'Institut de Recherches de l'OSF. Il offre toutes les fonctionnalités du système UNIX. Le système OSF/1 AD a servi de base pour le développement du système Paragon OSF/1 [Rab93]. Le système Paragon OSF/1 est une version étendue du système OSF/1 disposant d'un support pour le parallélisme.

Le modèle de processus de Paragon OSF/1 est fondé sur trois entités: l'application parallèle, le processus et le thread.

Une application parallèle est composée de plusieurs processus qui s'exécutent sur une partition. Une partition comporte un nombre fixe de nœuds. Sur toutes les machines Paragon, il existe deux partitions spéciales: la partition de service et la partition de calcul. La partition de service est utilisée pour exécuter des programmes séquentiels tels que compilateurs et éditeurs. La partition de calcul exécute les programmes parallèles.

Les processus Paragon OSF/1 communiquent par échange de messages. La communication peut être synchrone ou asynchrone. La communication à l'intérieur de l'OSF/1 Paragon, gérée par la bibliothèque nx, est du type n processus vers 1 processus. Le processus destinataire doit être toujours nommé par son identification. L'identification d'un processus est composée du numéro logique du nœud sur lequel il s'exécute et d'un numéro entier appelé type.

Le système de traitement de messages de l'OSF/1 Paragon est lent [SS94]. Bien que le débit supporté par matériel soit 175 Moctets/sec, le système OSF/1 envoie des messages avec un débit maximum de 35 Moctets/sec. Les latences obtenues avec l'utilisation des primitives d'échange de messages OSF/1 sont de l'ordre de 100 μs alors que latence du processeur de communication est de 70ns.

Plusieurs threads peuvent être créés à l'intérieur d'un processus. Le mécanisme de threads offert par l'OSF/1 Paragon s'inspire des threads POSIX (*pthreads*). Les ressources allouées à un processus sont partagées parmi tous les threads du processus. La bibliothèque pthreads offre des mécanismes de création et termination de threads aussi bien que des mécanismes de synchronisation. La synchronisation est réalisée à l'aide des verrous et des variables condition [Int95].

L'utilisation de flots d'exécution multiples rajoute de la complexité à l'implantation des applications parallèles. Il faut maintenant prévoir des mécanismes de contrôle d'accès aux structures de données qui sont partagées entre plusieurs threads.

L'autre cause de l'augmentation de la complexité est moins évidente que la précédente. Elle a son origine dans l'incompatibilité entre les bibliothèques déjà implantées et les threads. La plupart des bibliothèques existantes ne s'occupent pas du contrôle d'accès aux structures puisqu'elles admettent que le processus appelant n'est pas multi-flot.

Sur la Paragon, l'échange de messages, le traitement des signaux et la terminaison des processus peuvent générer des résultats inattendus quand exécutés par un processus multi-flot [Int95]. Pour résoudre ce problème, il faut soit garantir qu'un seule thread réalise des appels à la bibliothèque, soit protéger chaque appel avec des verrous.

5.5 La mise en œuvre du prototype

L'implantation du prototype de $\mathcal{DIVA}$ utilise des foncitonnalités de l'OSF/1 aussi bien que les fontionnalités de Mach.

Du point de vue du système Paragon OSF/1, $\mathcal{DIVA}$ est une application parallèle composée de n processus où n est le nombre de nœuds de la partition sur laquelle $\mathcal{DIVA}$ s'exécute. L'application utilisateur est aussi composée de processus qui s'exécutent sur la totatité ou sur un sous-ensemble des nœuds allouées à $\mathcal{DIVA}$.

Du point de vue de Mach, $\mathcal{DIVA}$ est un serveur qui implante un gestionnaire de la mémoire. L'accès à $\mathcal{DIVA}$ par les applications suit alors le protocole client/serveur défini dans [Ope92b].

5.5.1 Procédures initiales du serveur

Avant de commencer à traiter les rêquetes, notre serveur exécute les procédures d'initialisation suivantes:

1. Le serveur se déclare comme une application parallèle Paragon OSF/1 et réserve n nœuds pour son exécution.

   Ensuite, un processus est crée sur chaque nœud. Chaque processus exécute une copie de $\mathcal{DIVA}$. Nous allons décrire par la suite ce qui se passe dans chaque processus.

2. Le serveur publie son nom à travers le serveur de noms.

   Le système Mach de la Paragon est livré avec le serveur de noms standard. Bien qu'une copie de Mach s'exécute sur chaque nœud, la machine entière est vue comme un hôte unique.

   Selon l'architecture que nous avons proposée, l'utilisateur communique uniquement avec le serveur local. Ainsi, chaque copie de $\mathcal{DIVA}$ doit publier son nom localement. Pour y arriver, nous avons rajouté le numéro du nœud au nom du serveur. L'utilisateur aura donc accès direct au serveur diva$n$ où n est le numéro du nœud sur lequel le processus utilisateur s'exécute.

   La connexion au serveur $\mathcal{DIVA}$ est montrée dans la figure 5.12.

   La publication d'un serveur rend le service accessible à tous les processus qui s'exécutent sur un même nœud. Un processus obtient le droit d'accès à un service à travers la primitive `netname_look_up`.

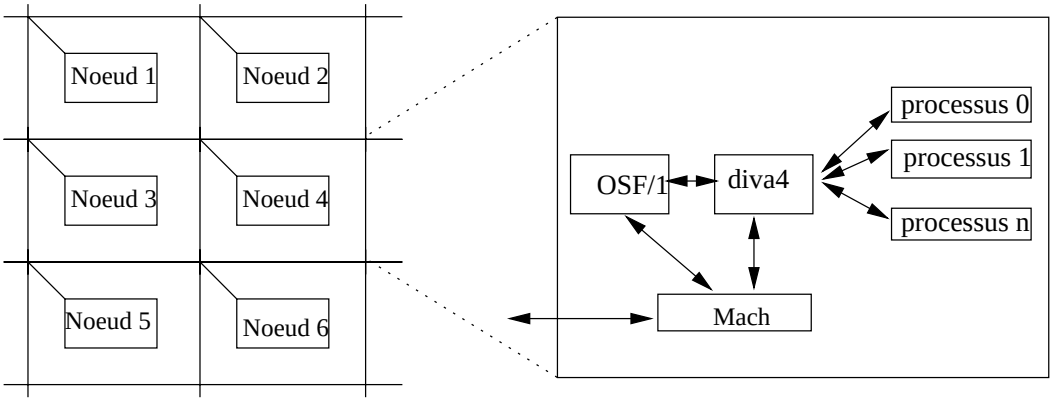


FIG. 5.12 – *La connexion au serveur*

Selon ce schema, une région mémoire peut être partagée par des processus qui composent une même application aussi bien que par des processus qui appartiennent à applications parallèles distinctes.

3. Le thread responsable de la réception des messages qui arrivent des nœuds distants est créé.

4. Le serveur reste bloqué jusqu'à l'arrivée d'une requête du noyau ou d'un appel à une primitive $\mathcal{DIVA}$. A ce moment, la seule primitive autorisée est `diva_set_memory_model` et le noyau ne peut demander que l'initialisation de l'objet partagé (`memory_object_init`). Le thread de réception de messages reste bloqué jusqu'à l'arrivée d'un message d'un nœud distant.

A la fin des procédures d'initialisation, nous avons sur un nœud la configuration montrée par la figure 5.13.

5.5.2 Procédures initiales du client

De façon similaire au serveur $\mathcal{DIVA}$, une application parallèle "usager" est composée de m processus. Un processus est créé sur chacun des m nœuds alloués à l'application parallèle.

Avant de pouvoir utiliser les fonctionnalités de $\mathcal{DIVA}$, l'application doit se connecter au serveur et ensuite effectuer la projection de la région partagée sur son espace d'adressage.

La connection au serveur est effectuée à travers la primitive `netname_look_up`. Le nom du serveur est diva$n$ où n est le nœud sur lequel le processus parallèle s'exécute.

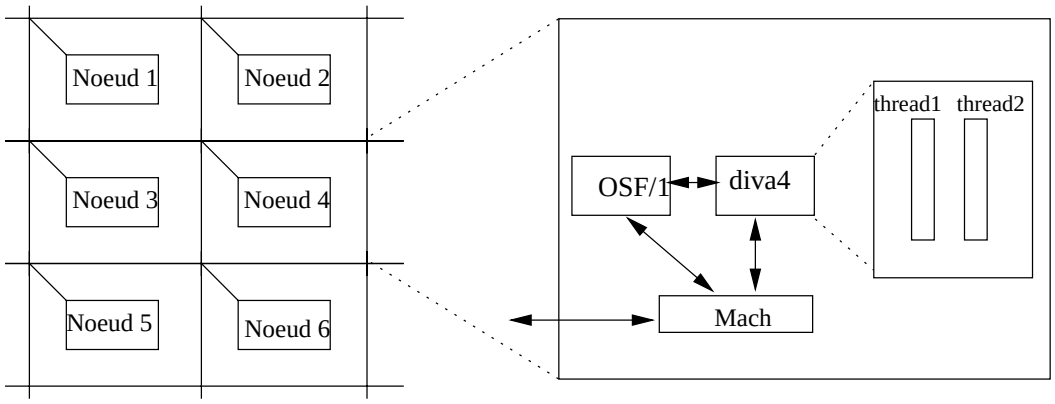


FIG. 5.13 – *L'initialisation du serveur*

La projection d'une région sur l'espace d'adressage de l'application est effectuée par l'appel à `vm_map`. L'identificateur du serveur est un des paramètres de cette primitive.

Après l'exécution de la projection, l'utilisateur peut référencer librement la région partagée. De plus, il peut faire appel à toutes les primitives $\mathcal{DIVA}$.

5.5.3 Définition de la région à partager

Sur Mach, chaque gestionnaire de la mémoire est responsable de la gestion d'une région mémoire. Quand un processus utilisateur effectue la projection d'une région mémoire sur son espace d'adressage, le gestionnaire correspondant est activé. A partir de là, toute activité de pagination effectuée sur la région projetée sera réalisée par le gestionnaire-serveur. L'accès à la région est effectué par des opérations traditionnelles d'accès à la mémoire (LOAD et STORE).

La correspondance biunivoque entre serveur et région partagée a créé la première restriction d'implantation à notre prototype. Dans sa conception, $\mathcal{DIVA}$ est capable de gérer plusieurs objets mémoire simultanément (voir paragraphe 5.1.2). Néanmoins, notre prototype ne s'occupe que d'une seule région mémoire. Malgré cette restriction, la taille de la région partagée aussi bien que le nombre de processus qui la partagent peuvent être très importants.

Le schéma de partage implanté est illustré par la figure 5.14.

La restriction d'une seule région mémoire par gestionnaire n'est pas incontournable. Cependant, nous avons jugé que l'effort nécessaire pour aller au delá

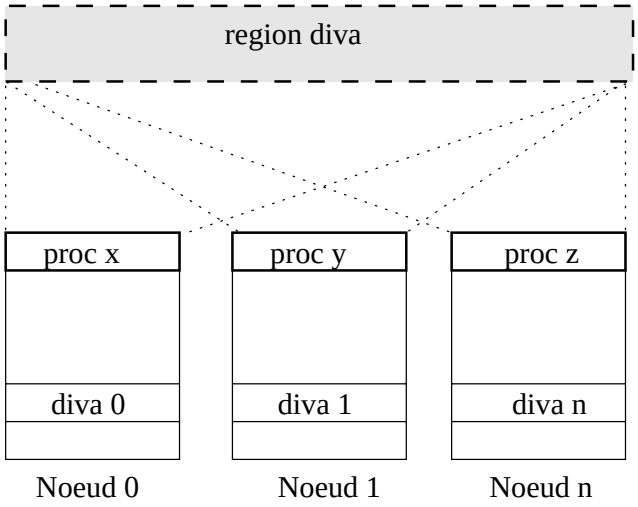


FIG. 5.14 – *Le schéma de partage du prototype de $\mathcal{DIVA}$*

nétait pas justifié dans une première version du prototype.

5.5.4 Chargement d'une page

Un défaut de page est généré au moment d'une référence à une page qui ne se trouve pas sur la mémoire locale. Le protocole exécuté lors du traitement du défaut de page est montré dans la figure 5.15.

Le noyau Mach local est activé par une exception de défaut de page dans le nœud x (1). Le noyau détermine alors le gestionnaire de la mémoire qui gère l'objet qui a causé le défaut de page. Dans notre cas, le gestionnaire de la mémoire est le serveur $\mathcal{DIVA}$. Ensuite, le noyau local envoie le message `m_o_d_request` à la copie locale de $\mathcal{DIVA}$ (2). Le serveur $\mathcal{DIVA}$ envoie à son tour un message au gestionnaire de la page qui a généré le défaut (3). Le processeur gestionnaire de la page (nœud z) est obtenu par l'expression n_{page} MOD n_{proc} où n_{page} est le numéro de la page et n_{proc} est le nombre total de processeurs alloués au serveur. Les fonctions de chargement et cohérence sont assurées par le gestionnaire de la page.

Ayant reçu le message, le gestionnaire exécute les opérations de cohérence nécessaires selon le type de la page et le modèle de cohérence courant. Les opérations de cohérence peuvent causer plusieurs communications distantes. Dans le pire des cas, deux messages sont échangés avec chacun des nœuds qui détiennent une copie de la page.

L'entrée correspondante à la page <page> dans la table de pages est mise à jour en modifiant les champs <copyset> et <état> sont modifiés.

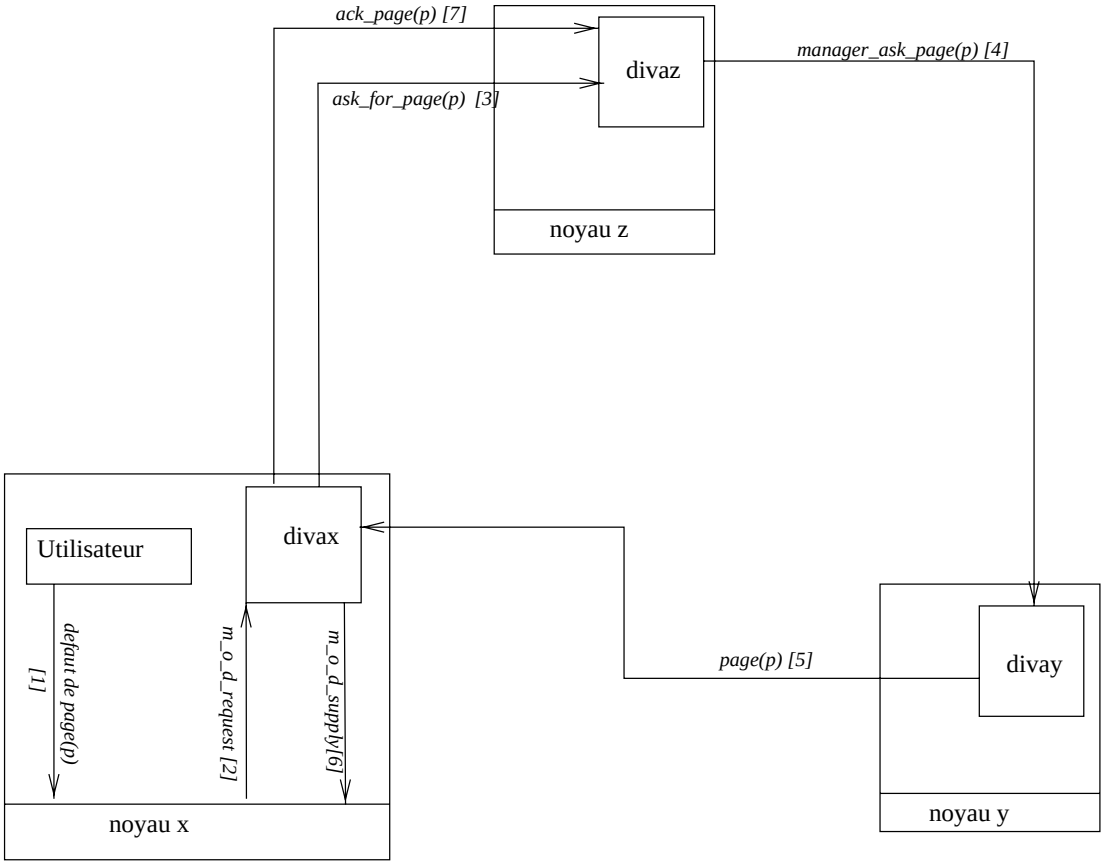


FIG. 5.15 – *Le chargement d'une page (cas général)*

Les gestionnaire demande à un nœud y tel que $y \in copyset(page)$ que la page soit rendue au serveur $\mathcal{DIVA}$ local au nœud qui a généré le défaut de page (4).

Dès que le serveur reçoit le message qui contient la page (5), il la rend au noyau Mach à travers le message `memory_object_data_supply` (6). Le protocole se termine par l'envoi d'un acquittement au gestionnaire de la page par le serveur $\mathcal{DIVA}$ (7).

Une analyse plus attentive de ce protocole permet de constater que le serveur placé sur le nœud qui détient la page (nœud y) ne communique pas avec le noyau local pour récupérer la page et la rendre au nœud qui a initié l'opération de chargement.

Un tel comportement n'est possible que si le serveur $\mathcal{DIVA}$ garde une copie de chaque page envoyée au noyau Mach. Dû à des problèmes dans la version de Mach implantée sur la machine Paragon [Sea95], c'est effectivement ce qui se passe dans $\mathcal{DIVA}$.

Cette restriction augmente beaucoup le coût en mémoire de notre serveur.

Néanmoins, elle permet qu'une page en lecture soit envoyée à un nœud distant sans que le noyau local au nœud qui detient une copie de la page soit contacté.

Le nombre de messages distants échangés par le protocole est:

$$n_{msg}(gen, p) = 4 + op_{coh}(p)$$

où gen est le protocole du cas général;

p est la page demandée;

op_{coh} est le nombre de communcations distantes nécessaires pour garantir la cohérence de la page p.

Le chargement des pages est donc une opération coûteuse dans notre serveur. Dans le but de réduire le nombre de messages échangés lors de l'exécution du protocole, nous avons alors conçu quelques optimisations du cas général qui vient d'être présenté.

Optimisation 1 - Réduction de l'ensemble de copies

Nous avons vu auparavant que les opérations de cohérence peuvent être exécutées au moment du chargement d'une page p. Ces opérations sont implantées par un protocole de cohérence qui agit en général sur l'ensemble $copyset(p)$ des nœuds qui ont une copie locale de la page.

Il est évident que le nombre de messages échangés dépend du nombre de processeurs qui appartiennent à $copyset(p)$. Afin d'établir une borne sur le nombre de messages de cohérence échangés, nous avons limité le nombre de processeurs qui peuvent posséder simultanément une copie d'une page. Au-délà de cette limite (LIM), une copie est invalidée.

Le nombre de messages distants échangés par le protocole optimisé est:

$$n_{msg}(gen_{ot}, p) = 4 + op_{coh}(p)$$

où gen_{ot} est le protocole du cas général optimisé, p est la page demandée et $op_{coh} \leq LIM$ est le nombre de communcations distantes nécessaires pour garantir la cohérence de la page p.

Le protocole gen_{ot} a été implanté dans $\mathcal{DIVA}$ pour le cas général de chargement de pages.

Optimisation 2 - La première référence à une page

Dans les cas précédents, il faut toujours décider si certaines opérations de cohérence doivent être effectuées avant que la page ne soit rendue au processeur fautif. L'analyse des modèles de cohérence de la mémoire nous a montré que la toute première référence à une page n'entraîne jamais d'opérations de cohérence. Cette observation nous a permis d'implanter l'optimisation suivante: la toute première référence à une page est entièrement traitée par son gestionnaire et la procédure qui implante le modèle de cohérence courant n'est pas appelée.

Le nombre de messages distants échangés par le protocole est:

$$n_{msg}(ot4, p) = 2$$

Le nombre de communications distantes effectuées dans ce cas est fixe et égal à 2. Les messages concernés sont ceux montrés dans la figure 5.17.

Par la suite, nous présentons des optimisations implantées pour quelques cas particuliers.

Cas 1 - Défaut de page sur le gestionnaire de la page

La figure 5.16 montre le cas particulier où le processeur "fautif" est aussi le gestionnaire de la page.

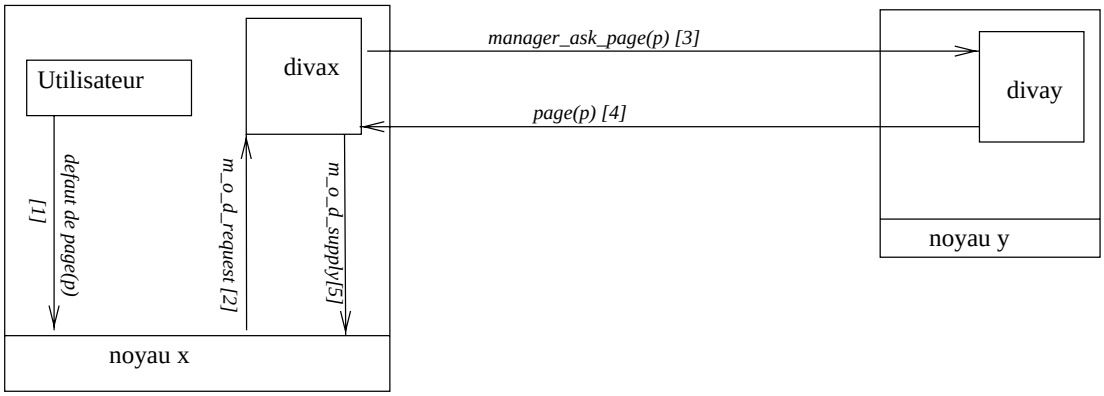


FIG. 5.16 – *Le chargement d'une page (cas 1)*

Les opérations de cohérence sont encore exécutées par le gestionnaire. Néanmoins, le processeur fautif connaît l'ensemble $copyset(p)$ des processeurs qui détiennent une copie de la page. Il demande une copie de la page directement à un de ces processeurs.

Le nombre de messages distants échangés par le protocole est:

$$n_{msg}(ot_2, p) = 2 + op_{coh}(p)$$

où ot_2 est le protocole de l'optimisation 2, p est la page demandée et $op_{coh} \leq LIM$ est le nombre de communcations distantes nécessaires pour garantir la cohérence de la page p.

Nous pouvons noter qu'il y a deux messages en moins par rapport au chargement des pages dans le cas général.

Cas 2 - Le gestionnaire détient une copie

La figure 5.17 montre le cas particulier où le gestionnaire de la page p détient une copie de p.

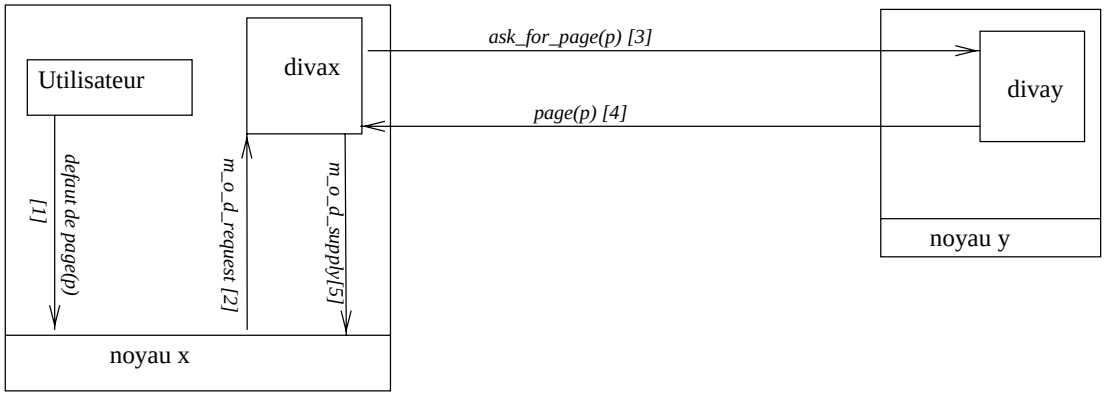


FIG. 5.17 – *Le chargement d'une page (cas 2)*

Le nombre de messages distantes échangés par le protocole est:

$$n_{msg}(ot_3, p) = 2 + op_{coh}(p)$$

où ot_3 est le protocole de l'optimisation 3, p est la page demandée et $op_{coh} \leq LIM$ est le nombre de communcations distantes nécessaires pour garantir la cohérence de la page p.

Comme dans le cas précédent, deux messages sont évités. En effet, comme le gestionnaire détient une copie de la page, il peut la rendre directement au processeur fautif.

5.6 Remplacement des pages

Sur chaque nœud, le noyau Mach exécute un démon de remplacement qui décide la page à supprimer de la mémoire locale. L'algorithme utilisé est un algorithme FIFO avec seconde chance [Dra92]. Ceci est réalisé de manière transparente au gestionnaire de la mémoire qui s'occupe de l'objet. Le gestionnaire de la mémoire (dans notre cas le serveur $\mathcal{DIVA}$) ne peut pas intervenir dans le choix de la page à remplacer.

Le noyau Mach retourne la page au gestionnaire de l'objet à travers le message `memory_object_data_return` [Ope92a]. Seules les pages modifiées sont retournées au gestionnaire. Les pages inaltérées sont simplement supprimées puisque le gestionnaire de la mémoire détient lui aussi une copie.

Comme le serveur n'intervient pas sur le choix de la page à remplacer, nous n'avons implanté que la procédure `choisir_nœud` (voir paragraphe 4.1) pour le remplacement des pages.

Les messages échangés dans l'implantation de la procédure sont montrés dans la figure 5.18.

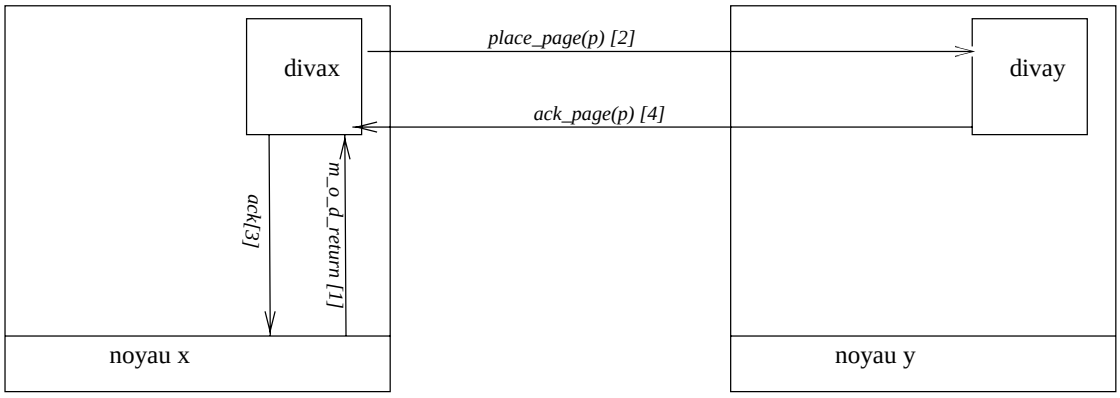


FIG. 5.18 – *Messages échangés lors du remplacement*

A la suite d'un `memory_object_data_return` associé à une opération de remplacement, la procédure `choisit_nœud` est toujours appelée. Le placement de la page modifiée sur un dispositif autre que la mémoire locale du nœud doit être fait rapidement.

Notre procédure `choisir_nœud` exécute une recherche dans un tableau de petite taille (`table_voisins`) et ensuite envoie la page à un nœud distant ou au disque. A ce moment, d'autres appels à la primitive `memory_object_data_return` peuvent être traitées. L'acquittement correspondant à la requête de placement de

page est reçu de façon asynchrone. Au moment de la réception de l'acquittement, la page est supprimée de l'espace d'adressage du serveur.

Le protocole de communication entre le gestionnaire de la mémoire et le noyau ne permet pas que le gestionnaire sollicite le placement d'une page non référencée sur la mémoire locale d'une application. Sur Mach, les pages ne sont chargées en mémoire que quand un processus les accède.

Afin de stocker les pages empruntées qui arrivent à un nœud, nous utilisons un ensemble de tampons localisés à l'intérieur du serveur. Cet ensemble de tampons, appelé `waiting_pages` stocke les pages empruntées aussi bien que les pages préchargées.

Dès qu'une page est référencée localement, elle est envoyée au noyau. Les références distantes causent la migration de la page vers le nœud qui a généré la référence.

Les tampons sont gérés selon la politique FIFO (c.f. paragraphe 4.2). Si la page à supprimer des tampons est du type emprunté, elle migre vers le nœud retourné par la procédure `choisir_nœud`.

5.7 Préchargement de pages

Dans le paragraphe 4.2, nous avons vu que l'introduction des mécanismes de préchargement de pages rajoute un coût important à notre système. Essentiellement, le temps d'exécution des opérations de cohérence et chargement est augmenté à cause de la recherche des pages en attente (tampon `waiting_pages`).

Pour réduire ce coût, nous avons implanté un mécanisme de contrôle de ces pages en attente. Basiquement, une variable indique l'existence de pages dans le tampon. S'il n'existe pas de pages stockées, la procédure correspondante à la recherche dans les tampons n'est pas activée.

Le préchargement de pages est initié par l'appel à la primitive `diva_prefetch`. A la suite du premier appel à `diva_prefetch`, deux threads sont créés. Un des threads s'occupe du traitement des requêtes déposées sur la file de requêtes et l'autre s'occupe du vidage du tampon `waiting_pages`. L'intération entre ces deux threads est faite par deux files partagées: le tampon `waiting_pages` et la file de requêtes. La figure 5.19 montre l'interaction entre les threads de préchargement.

Traitement des requêtes

Quand le gestionnaire de la mémoire partagée dépose une requête sur la file de requêtes, il met à jour la variable partagée `n_requêtes`. Le thread de traitement

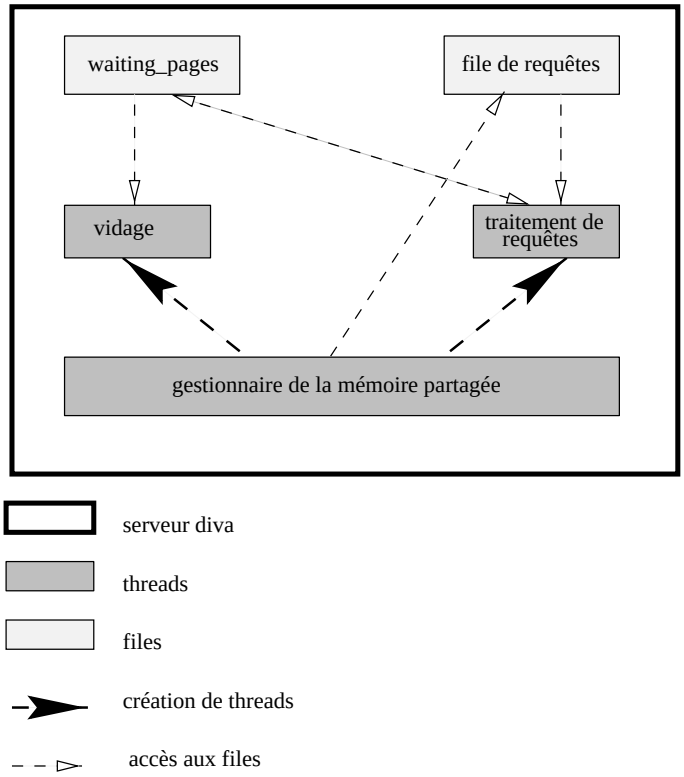


FIG. 5.19 – *Threads du préchargement*

de requêtes initie son activité lorsque la variable `n_requêtes` a une valeur plus grande que 0.

La file de requêtes est accédée selon l'ordre FIFO. Les pages à précharger sont sollicitées selon le protocole de chargement de pages décrit dans le paragraphe 5.5.4. Néanmoins, à la fin de l'exécution du protocole, la page est placée sur le tampon `waiting_pages` alors que dans le protocole de chargement la page est rendue au noyau. La situation de file pleine est aussi traitée par ce thread. Dans ce cas, une page est supprimée de la file.

Vidage du tampon

Le thread responsable du vidage du tampon `waiting_page` s'exécute périodiquement. Il effectue le parcours du tampon dans l'ordre croissant et toute page dont le temps de permanence sur le tampon est plus grand que TEMPS_PERMANENCE_MAXIMUM est supprimée. Le gestionnaire de chaque page supprimée est informé de cette opération. Quand le tampon `waiting_pages` est vide, le thread de vidage ne s'exécute pas.

5.8 Contrôle d'accès aux structures partagées

Dans chaque nœud, le serveur $\mathcal{DIVA}$ exécute plusieurs threads. Comme ces threads partagent l'espace d'adressage du serveur, il est nécessaire de prévoir des mécanismes de contrôle d'accès aux variables accédées par plusieurs threads. Les différentes structures de données manipulées par les threads sont montrées dans la figure 5.20.

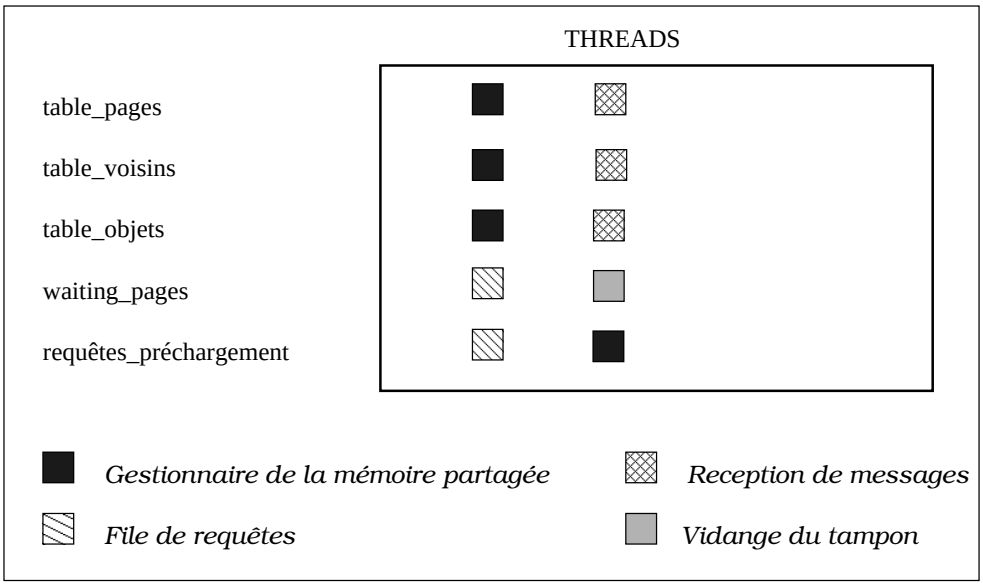


FIG. 5.20 – *Partage des structures de données entre les threads*

Afin d'assurer la synchronisation à l'intérieur de notre serveur, nous avons implanté les threads de $\mathcal{DIVA}$ comme threads POSIX (pthreads) [Int95]. Les pthreads se synchronisent à l'aide des verrous et des variables condition. Par la suite, nous présentons les mécanismes utilisés pour chaque structure partagée.

Table de pages - Dans chaque serveur, la table de pages contient une entrée par page partagée. Chaque entrée possède un verrou associé et une opération de verrouillage `mutex_lock` sur le verrou est effectuée avant tout accès à cette entrée. Les champs associées à chaque entrée de la table de pages sont montrés dans la figure 5.5.

Table d'objets - Le contrôle d'accès à la table d'objets est réalisé par objet de façon similaire à la table de pages.

Table de voisins - Dans les deux cas précédents, les tables pouvaient être écrites par deux threads. Nous avons alors implanté un mécanisme traditionnel de synchronisation (le verrou) pour interdire l'accès simultané par deux threads.

La table de voisins est accédée de manière très particulière et, par conséquent, nécessite une étude plus détaillée.

La table de voisins est mise à jour par le thread de réception de messages quand un message de changement d'état d'occupation est reçu. Chaque message contient un seul état d'occupation. Une seule entrée est donc mise à jour à la fois.

La procédure `choisir_nœud`, exécutée par le thread gestionnaire de la mémoire partagée ne fait que lire la table de voisins à la recherche d'un nœud qui est libre.

Ainsi, nous avons deux threads qui accèdent la structure de données mais un des threads ne fait que la lire et le deuxième thread met à jour une entrée à la fois. Ce type de partage ne cause pas d'interférence entre les threads et ne peut pas conduire à des résultats incorrects. Pour cette raison, aucun contrôle d'accès n'a été prévu pour la table de voisins.

Waiting_pages - Le tampon `waiting_pages` a été implanté comme une liste chaînée. L'addition d'éléments à cette liste est effectuée par le thread de traitement de requêtes. La suppression d'éléments peut être effectuée par ce même thread ou bien par le thread de vidage.

Comme la structure de données en question est une liste chaînée, le verrouillage d'éléments n'est pas suffisant pour assurer la cohérence de la liste. Nous avons alors implanté un verrou qui controle l'accès à toute la liste. Toutes les opérations sur cette liste sont ainsi sérialisées.

Rêquetes de préchargement - Comme dans le cas précédent, la file de requêtes de préchargement a été aussi implantée comme une liste chaînée. Les mêmes mécanismes de contrôle d'accès sont aussi utilisés.

Chapitre 6

Evaluation du serveur

L'objet de ce chapitre est de réaliser surtout une évaluation fonctionnelle du système $\mathcal{DIVA}$. Les mesures ont été réalisées sur le prototype de $DIVA$ décrit dans le chapitre précedent. Nous présentons d'abord l'effort de programmation dépensé dans chaque fonction du serveur. Nous faisons par la suite une évaluation du comportement d'une même application sous des modèles de cohérence différents. A la fin, nous présenterons quand même quelques performances de ce prototype de $\mathcal{DIVA}$, malgré les choix très limitatifs de l'implantation faite.

6.1 Effort de programmation

L'objectif de ce paragraphe est de mesurer l'effort de programmation nécessaire à la mise en œuvre d'un serveur avec les fonctionnalités offertes par $\mathcal{DIVA}$. A titre indicatif, le programme qui implante $\mathcal{DIVA}$ a 4000 lignes de code C environ et la taille du fichier exécutable est de 365 Koctets.

Le graphique que nous présentons dans la figure 6.1 montre le pourcentage du nombre de lignes de code qui a été consacré à chaque fonction particulière.

Nous pouvons noter ici que plus de la moitié du code de $\mathcal{DIVA}$ est consacrée aux fonctions de cohérence et chargement des pages. La programmation de ces deux fonctions présente aussi un grand degré de difficulté. Ceci est dû en grande partie aux protocoles de cohérence de la mémoire et aux protocoles de cohérence du cache.

Nous avons implanté deux modèles de cohérence: la cohérence séquentielle et la cohérence à la libération. Du total du code consacré à la cohérence, nous utilisons environ 2/3 pour implanter la cohérence séquentielle et le reste pour implanter la cohérence à la libération.

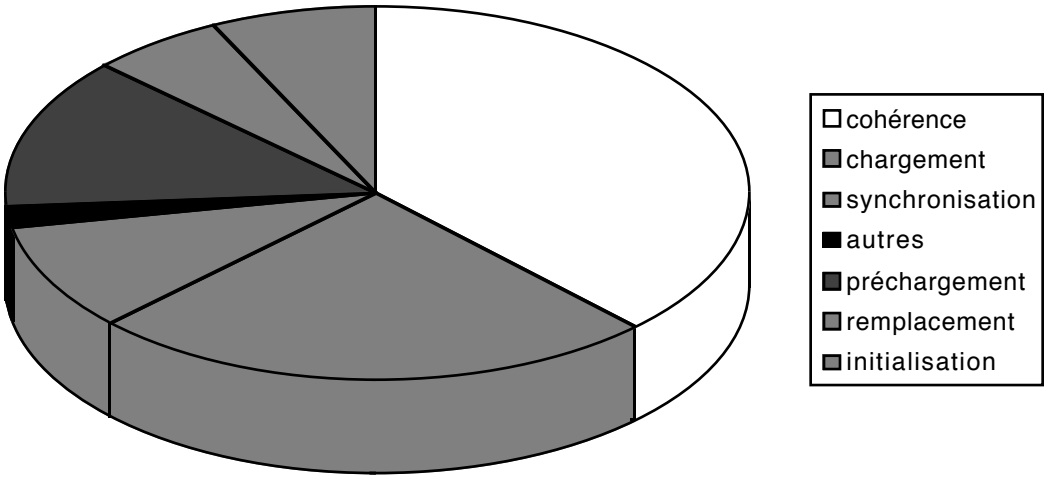


FIG. 6.1 – *L'effort de programmation*

Ces données nous ont surpris car nous avions pensé que la cohérence séquentielle nécessitait moins d'effort de programmation. L'analyse du code a montré que l'implantation du protocole d'invalidation MRSW entraîne des échanges de messages entre nœuds dans les cas des défauts de page de chargement ainsi que dans le cas des défauts de page de protection. L'invalidation des pages génère un grand nombre de types de messages différents.

L'échange de messages entre nœuds sur la Paragon exige d'abord que le message soit préparé. Ensuite, le message est envoyé et le nœud origine attend l'accusé de réception en mode asynchrone. La réception des messages est faite par un thread qui exécute le traitement associé au type du message. L'échange de messages est toujours associé à l'exécution d'un protocole. Ceci entraîne des performances mauvaises et non significatives. Par exemple, dans le noyau ParX, l'échange de messages n'est pas associée à l'exécution d'un protocole.

La cohérence à la libération a été implantée selon le mécanisme de "diffs" proposé dans le système Munin [Car93]. Au moment de l'acquisition d'un verrou, toutes les pages sont protégées contre l'écriture. Les défauts de page de protection sont traités exclusivement par le serveur local. Le traitement consiste à faire une copie de la page en défaut et la stocker dans une file. Au moment de la libération, les modifications effectuées sur les pages sont envoyées aux gestionnaires correspondants.

Les opérations de chargement de pages sont coûteuses sur n'importe quel système à mémoire virtuelle partagée. Elles exigent des échanges de messages entre nœuds et, dans le cas spécifique du système Mach, le protocole décrit dans le paragraphe 5.3 doit être exécuté.

La plus grande partie de l'effort de programmation due à l'implantation du préchargement vient de deux sources. En premier, il faut créer un thread par requête de préchargement (voir paragraphe 4.2). Comme sur la Paragon, l'existence de plus de six threads par processus parallèle peut conduire à des résultats inattendus [Int95], nous avons limité à deux le nombre de threads qui s'occupent de l'opération de préchargement. Ce contrôle a alors exigé un effort de programmation considérable. L'autre source d'effort de programmation est la gestion des tampons alloués aux pages préchargées.

En revanche, comme le remplacement de pages n'exécute que le mécanisme du choix du nœud vers lequel la page à supprimer doit migrer, l'effort de programmation est alors petit.

Bien que la synchronisation soit implantée par quatre appels au serveur, l'effort nécessaire a été faible. Ceci est partiellement dû à l'incapacité d'un serveur Mach à bloquer l'exécution d'un processus parallèle. La seule façon de le faire consiste à retarder l'envoi de la réponse à un appel au serveur. Ceci nécessite qu'un ou plusieurs threads du serveur soient allouées exclusivement au traitement de l'opération de verrouillage. Comme il existe une restriction sur le nombre de threads qui peuvent s'exécuter simultanément sur un même processus, nous avons opté pour une autre solution. La primitive `diva_lock` retourne VRAI à l'utilisateur si le verrou a été obtenu et FAUX dans le cas contraire. Ceci rend très simple la programmation de l'opération de verrouillage car il n'est pas nécessaire de gérer la file de processus en attente. En revanche, la programmation de l'application devient plus complexe puisqu'il faut rajouter la boucle d'acquisition de verrou.

6.2 Multiplication de matrices

L'analyse du comportement des applications parallèles par rapport aux modèles de cohérence de la mémoire est fondamentale si nous voulons trouver le bon compromis entre la simplicité de programmation et les hautes performances. Néanmoins, il faut faire quelques commentaires à propos de la manière dont l'analyse des applications est réalisée en général.

Dans la plupart des cas, c'est le chercheur qui a conçu le système à mémoire virtuelle partagée qui programme les applications analysées. Etant au courant de

toutes les caractéristiques de son système, il n'est pas bien placé pour juger la simplicité de programmation. Les performances obtenues seront aussi très bonnes car le chercheur peut utiliser la connaissance qu'il possède de son système pour programmer l'application cible.

Afin d'éviter ce type de situation, nous avons étudié une application très simple - la multiplication de matrices carrées n x n. Bien qu'il existe des algorithmes très performants et complèxes pour réaliser la multiplication de matrices [Pan80] [A+95], nous avons opté pour implanter un algorithme très simple et intuitif. Notre objectif ici n'est pas de proposer des implantations performantes de la multiplication de matrices. Nous nous intéressons plutôt à l'impact causé par le changement du modèle de cohérence sur l'exécution d'une application parallèle.

L'algorithme séquentiel utilisé pour la multiplication de matrices ainsi que sa version parallèle sont montrés dans la figure 6.2.

```
multiplication_matrices()

{
 int a[n][n], b[n][n], c[n][n];

 int i, j, k;

 for(i=0; i<n; i++)

  for(j=0; j<n; j++)

   for(k=0; k<n; k++)

    c[i,j] = c[i,j] + a[i,k] * b[k,j];

}
```

(a) Algorithme Séquentiel

```
calcul_par(dim1, dim2)
{
   int i, j, k, s[n,n];
   for(i=0; i<n; i++)
    for(j=dim1; j<dim2; j++)
    {
     s[i,j] = 0;
     for(k=0; k<n;k++)
       s[i,j] = s[i,j] + a[i,k]*b[k,j];
    }
    for(i=0; i<n; i++)
     for(j=dim1; j<dim2; j++)
       c[i,j] = s[i,j];
}

multiplication_matrices_parallèle(proc)
{
 int a[n][n], b[n][n], c[n][n];
 int i;
 forall(i=0; i<proc; i++)
  calcul_par(i*(n/proc),(i+1)*(n/proc));
}
```

(b) Algorithme Parallèle

FIG. 6.2 – *L'algorithme de multiplication de matrices*

Nous voulons calculer $C = A * B$, où A et B sont des matrices pleines[1]

1. Une matrice est dite pleine si la plupart de ses éléments sont différents de zero

carrées n x n. La version parallèle de la multiplication de matrices que nous avons utilisé ressemble beaucoup à l'algorithme séquentiel original. Nous avons effectué le découpage par colonne, où chacun des p processeurs du système doit calculer n/p colonnes. La figure 6.3 illustre le découpage. Pour simplifier, nous présentons dans la figure le cas particulier où $n = p$. Chaque processeur calcule une colonne.

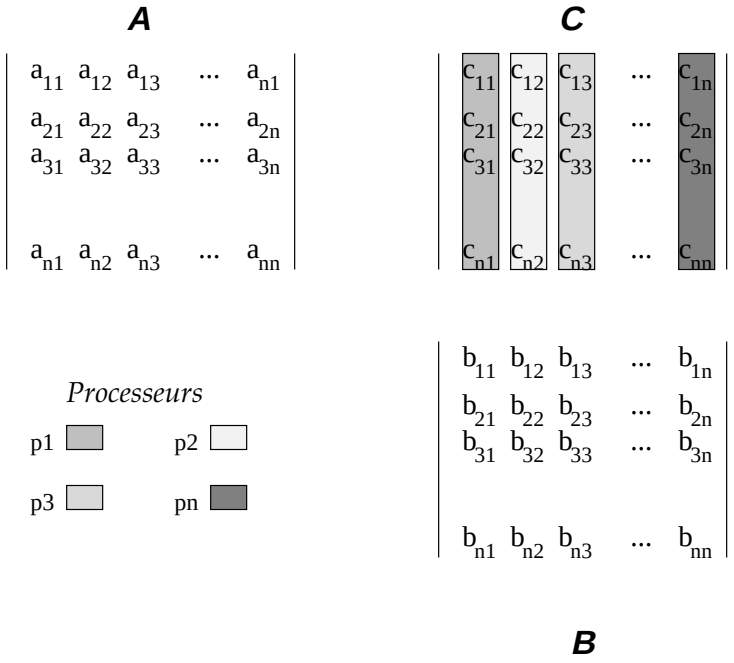


FIG. 6.3 – *Le découpage de la multiplication de matrices*

Dans cet exemple, chaque matrice A, B, C est placée sur une page différente. Les matrices A et B sont initialisées au début de l'exécution de l'application. Les résultats intermédiaires sont stockés dans un tampon. A la fin du calcul, le contenu du tampon est écrit dans la matrice C (c.f. figure 6.2).

L'analyse de cette application très simple nous a pourtant fourni des résultats fort intéressants et a fait ressortir le besoin d'un système qui supporte plusieurs modèles de cohérence.

Nous avons exécuté le même programme sur deux implantations de la cohérence séquentielle (*SC+inv* et *SC+temp*) et une implantation de la cohérence à la libération (*RC*). Les effets du préchargement de pages sont étudiés pour la cohérence séquentielle (*SC+pré*) et pour la cohérence à la libération (*RC+pré*). Les résultats, en nombres de défauts de page, sont montrés dans la figure 6.4, pour la multiplication de 2 matrices 16*16 sur 2, 4, 8 et 16 processeurs.

Les résultats présentés dans la figure 6.4 seront analysés pour chaque implantation de modèle de cohérence dans les paragraphes qui suivent.

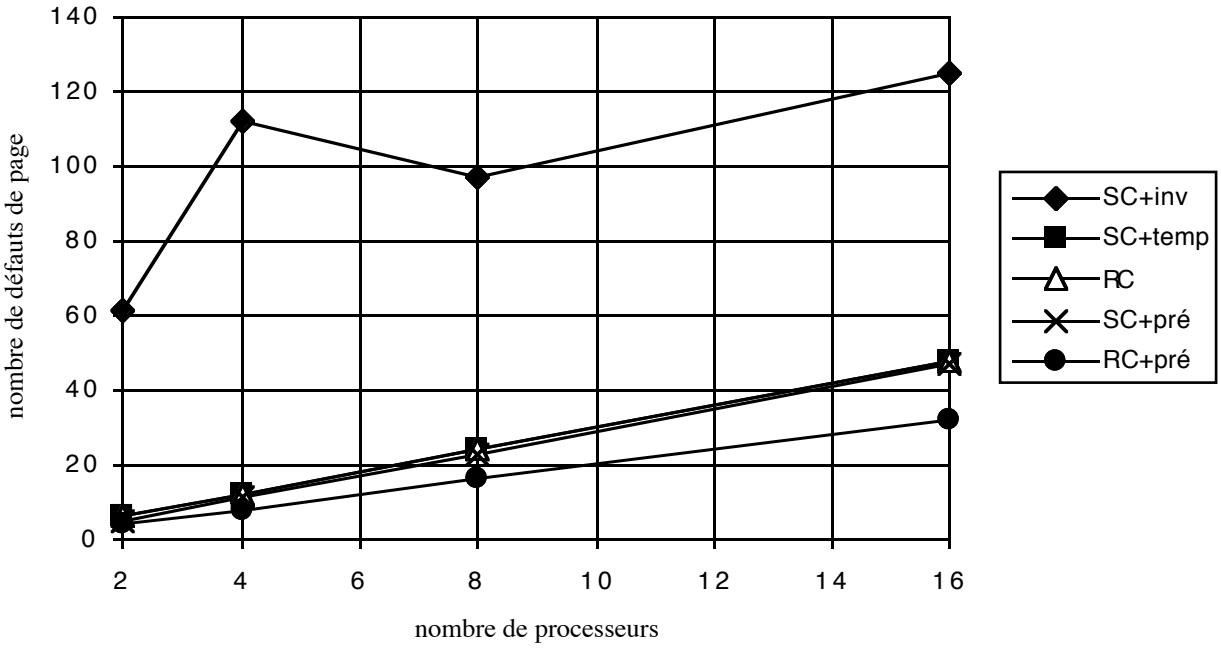


FIG. 6.4 – *La multiplication de matrices 16*16 selon 2 modèles de cohérence*

6.2.1 Cohérence séquentielle avec invalidation

Dans un premier temps, nous avons implanté la cohérence séquentielle avec un protocole d'invalidation MRSW. L'implantation que nous avons réalisée est inspirée de celle proposée par Li dans [Li88]. Au moment de la première lecture d'un élément de la matrice A ou B, un défaut de page est généré. Une opération de chargement se met alors en route. Les prochaines références aux données de A ou B se déroulent localement car la matrice est déjà chargée en mémoire. Dans notre exemple, chaque matrice est placée exactement sur une page.

Pour les lectures des matrices A et B, nous avons un nombre de défauts de page constant et égal à deux par processeur.

Bien que, pour le cas de la multiplication des matrices, il n'existe pas de vrai partage de données en écriture car chaque processeur écrit exclusivement sur quelques colonnes de la matrice résultat, nous avons observé un phénomène intense de faux partage.

Avec le protocole MRSW, chaque fois qu'un défaut de page de protection en écriture est généré, les copies de la page sont invalidées avant que l'accès soit accordé au nœud qui a généré le défaut. Dans le cas de la multiplication de matrices, la matrice C est écrite par tous les processeurs. Alors, il existe une seule copie de la page qui contient cette matrice qui passe successivement entre

les nœuds (effet ping-pong).

Les invalidations fréquentes sont la cause du nombre extrêmement élevé de défauts de page présenté sur la figure 6.4.

Dans le cas des matrices de petite taille, le faux partage est réduit car dans notre implantation, le traitement des défauts d'une même page est fait toujours par le même nœud (gestionnaire distribué fixe) selon l'ordre FIFO. Ceci fait que le début de la phase de calcul soit décalé entre les processeurs. Cette situation est illustrée dans la figure 6.5.

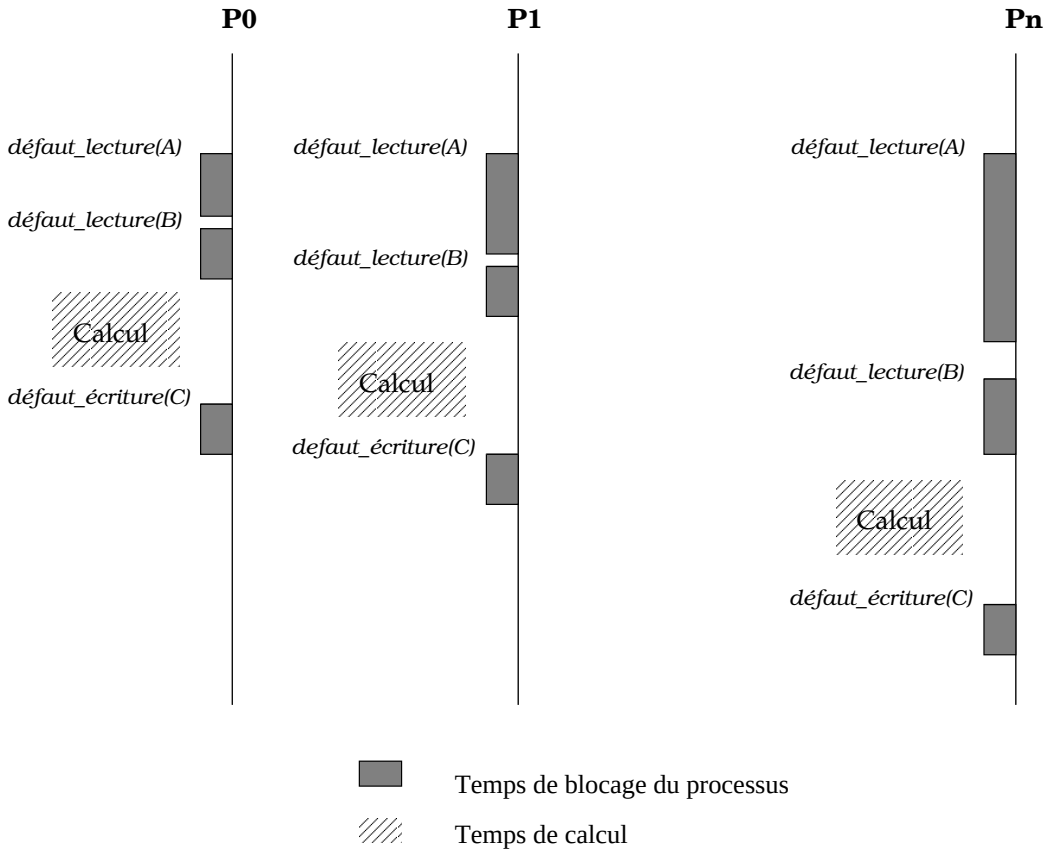


FIG. 6.5 – *Le décalage des phases de lecture et d'écriture*

Dans le cas où nous avons un décalage suffisant entre les phases de calcul des processeurs, il n'existe pas de coincidence temporelle entre les phases d'écriture. Ainsi, le nombre de défauts de page générés est de $2p$ défauts de lecture + p défauts en écriture. Le nombre total de défauts de page est alors $3p$ pour le cas idéal.

Hélas, ce cas idéal ne se produit que quand nous avons un nombre important

de processeurs et peu de calcul à faire. Dans les autres cas, la superposition des phases d'écriture est observée fréquemment et le nombre de défauts de page croît sensiblement.

Dans le pire des cas, chaque opération d'écriture va générer un défaut de page en écriture, ce que nous donne $n * n$ défauts de page en écriture. Si nous faisons la multiplication des matrices 512 X 512 sur 8 processeurs, nous aurons $n * n + 2p$ défauts de page, soit 262160 défauts de page!

Heureusement, l'occurrence d'une telle situation est fort improbable et en réalité le processeur exécute plusieurs écritures avant que la page ne lui soit réclamée. L'effet du faux partage reste pourtant visible et fait croître beaucoup le nombre des défauts de page de l'application.

6.2.2 Cohérence séquentielle avec temporisation

Le phénomène du faux partage a aussi été observé dans le système Mirage. Toujours en gardant le modèle de cohérence séquentielle, Fleish et Popek [FP89] ont proposé un mécanisme de temporisation pour réduire le faux partage. Un temporisateur garantit que la page reste au moins pendant un intervalle de temps Δ sur chaque site. Nous avons implanté ce mécanisme dans $\mathcal{DIVA}$ sous la forme d'un nouveau modèle de cohérence *SC+temp*.

L'implantation de ce modèle a été très simple puisque nous avions déjà implanté la cohérence séquentielle avec invalidation. Il nous a suffi de rajouter un mécanisme pour retarder les invalidations d'une période de temps Δ. L'application que nous avons exécutée est la même que celle du paragraphe précédent. Il a suffi de changer l'appel `diva_set_memory_model (SC)` pour `diva_set_memory_model (SC+temp)`.

Nous avons observé que, si le temps de permanence de la page sur le nœud est assez grand, le faux partage n'est plus observé. Maintenant, la page passe d'un nœud à un autre et les colonnes de la matrice C attribués au nœud sont écrites d'une seule fois. Cette situation est illustré par la figure 6.6. Cette figure répresente le déplacement de la matrice C entre les noueds. Nous pouvons noter que dans l'instant $n\Delta + c'''$ la matrice C a été entièrement écrite. A ce moment lá, l'exécution du programme s'achève.

Dans la figure, c', c'' et c''' sont les temps de migration de la page d'un nœud à autre. Ce temps est variable et dépend de la distance entre les nœuds.

Nous pouvons noter que chaque nœud a subi un seul défaut de page pour écrire la matrice C. Avec le mécanisme de temporisation, nous sommes alors capables de reproduire le cas idéal en nombre de défauts de page.

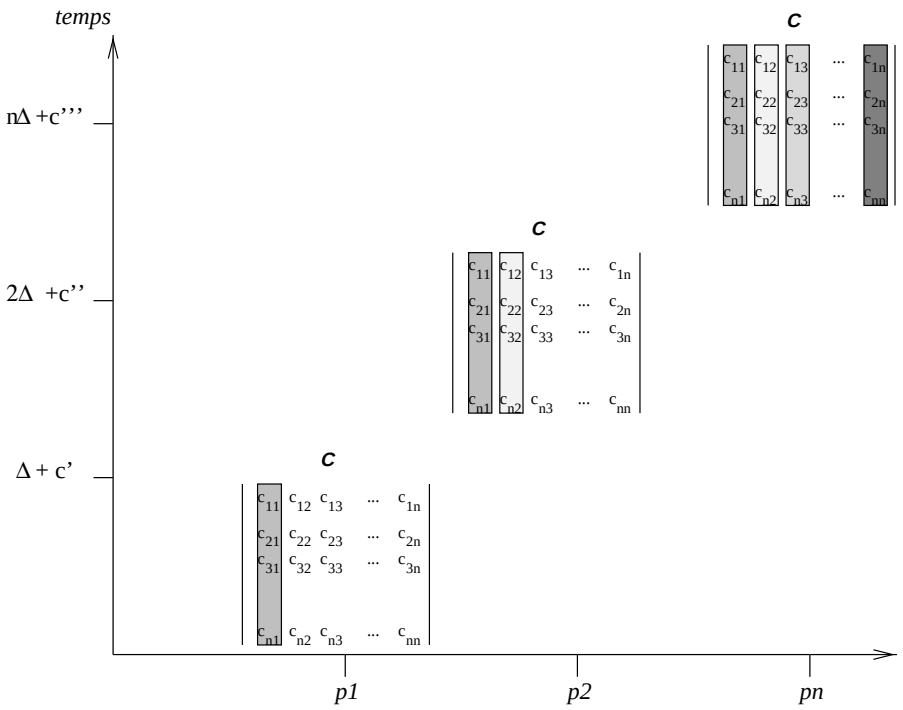


FIG. 6.6 – *Le déplacement de la matrice C entre les nœuds (SC+temp)*

Cependant, l'estimation de la bonne valeur pour Δ a été une tâche difficile. Nous avons exécuté l'algorithme plusieurs fois avec des valeurs différentes de Δ jusqu'à obtenir la situation présentée dans la figure 6.6. De plus, la bonne valeur de Δ dépend de la taille de la matrice et du nombre de processeurs alloués au calcul.

Pour obtenir les données présentés sur la figure 6.4, nous avons fixé Δ suffisamment grand et fait varier le nombre de processeurs. A titre illustratif, nous avons utilisé $\Delta = 0,5ms$.

Bien qu'en nombre de défauts de page le résultat obtenu soit idéal, les temps d'exécution de l'application ne sont pas bons. Dans certains cas, la page reste verrouillée sur un processeur longtemps après la fin de l'écriture du processeur sur la page.

6.2.3 Cohérence à la libération

Pour pouvoir exécuter l'algorithme de multiplication de matrices au-dessus de la cohérence à la libération, il a été nécessaire d'abord de spécifier le nouveau modèle de cohérence. Comme la cohérence à la libération est le modèle de cohérence par défaut de $\mathcal{DIVA}$, il nous a suffi d'enlever l'appel à la primitive

`diva_set_memory_model`($SC + temp$).

Malgré l'inexistence du vrai partage dans les accès à la matrice C, l'ajout des appels aux primitives `diva_lock` et `diva_unlock` a été fait pour déterminer le moment de propagation des modifications.

Au moment de l'exécution de la primitive `diva_unlock`, les modifications effectuées sur la matrice C sont envoyées à tous les processeurs qui détiennent une copie de cette matrice. Nous arrivons ainsi à la situation illustrée par la figure 6.7.

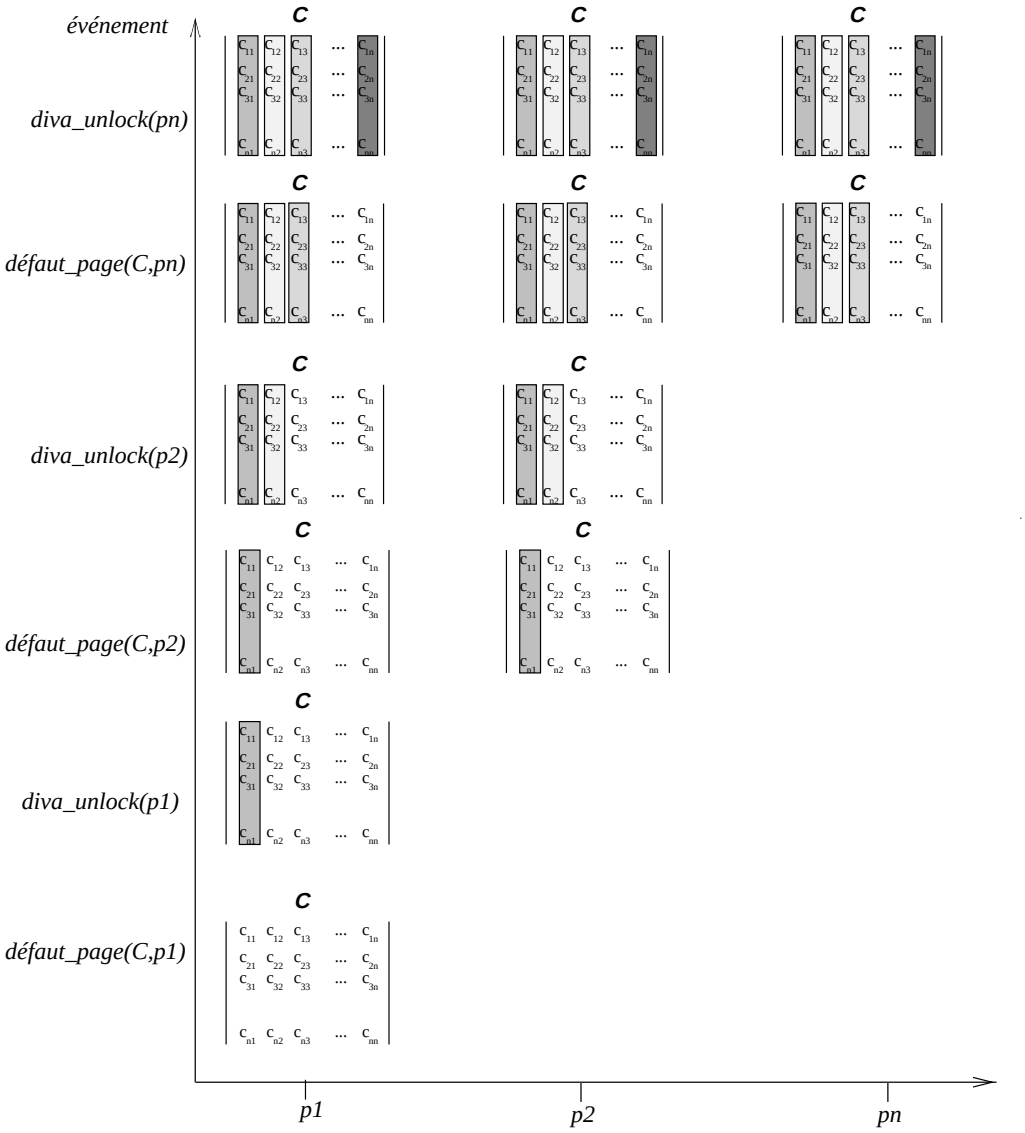


FIG. 6.7 – *Le déplacement de la matrice C (cohérence à la libération)*

De façon similaire à la figure 6.6, cette figure montre l'état de modification

de la matrice C ainsi que son déplacement entre les nœuds. L'exécution du programme s'achève quand le n_ème processeur exécute la primitive `diva_unlock`.

Dans le cas de $\mathcal{DIVA}$, la cohérence à la libération a été implantée selon le schéma proposé par Carter dans [Car93]. Le protocole de cohérence utilisé est celui de la mise à jour.

Dans le cas précédent, une seule copie de la matrice C existe à la fin de l'exécution de l'algorithme. Pour la cohérence à la libération, nous pouvons noter dans la figure 6.7 qu'à la fin de l'exécution chaque processeur possède une copie à jour de la matrice C. Ceci arrive car les modifications effectuées sur C sont propagées au moment de la libération à tous les processeurs qui possèdent une copie de la page sur sa mémoire locale.

En général, dans le cas de la multiplication de matrices, il suffit qu'un seul processeur détienne la matrice C à la fin de l'exécution pour afficher le résultat. Les copies supplémentaires sur les autres nœuds sont alors superflues.

Néanmoins, il se peut que la matrice C soit utilisée par tous les processeurs dans une prochaine phase de calcul. Dans ce cas, la situation présentée par la figure 6.7 est fort souhaitable. De toute façon, les mises à jour intermédiaires de C restent encore superflues. Cet effet peut être corrigé par l'implantation de la cohérence paresseuse à la libération, décrite dans le paragraphe 2.2.9.

Contrairement au cas illustré dans le paragraphe précédent, la situation où les accès à une page sont rétardés même quand aucun nœud ne l'accède ne se produit plus. Le contrôle des accès en écriture sur la matrice C est maintenant fait à travers les primitives d'obtention et relâchement de verrous. Le coût additionnel de cet algorithme provient essentiellement des mises à jour des copies.

6.2.4 Cohérence séquentielle avec préchargement

Les résultats étudiés dans ce paragraphe sont ceux obtenus par l'exécution du modèle *SC+temp* combiné avec le mécanisme de préchargement.

Afin d'étudier les effets du préchargement de pages sur l'exécution d'une application parallèle, nous nous sommes servis de la primitive de préchargement disponible dans $\mathcal{DIVA}$ pour exécuter la multiplication de matrices. Comme les matrices A et B sont lues au début de l'exécution de chaque processus parallèle, nous avons exécuté le préchargement uniquement pour la matrice C.

La seule modification faite dans le code de l'application a été l'insertion de la primitive `diva_prefetch` au début de l'exécution de chaque processus parallèle. Nous avons alors exécuté p opérations de préchargement de la matrice C, où p est le nombre de processeurs alloués à la multiplication.

Dans les résultats présentés, au moment de la première référence à C, nous

avons p copies de C dans le système, une par processeur. Comme le modèle *SC+temp* exécute un protocole d'invalidation, la seule copie qui sera gardée est celle du processeur qui a généré la première référence à C. Toutes les autres pages préchargées seront invalidées. La réduction en nombre de défauts de page obtenue est alors égale à un.

La figure 6.8 montre le déplacement de la matrice C pour le cas de la cohérence séquentielle avec préchargement. Nous montrons sur la figure le cas où la première référence à la matrice C est effectuée par le processeur p_1. L'opération de préchargement est terminée par rapport à tous les processeurs avant la première référence à C.

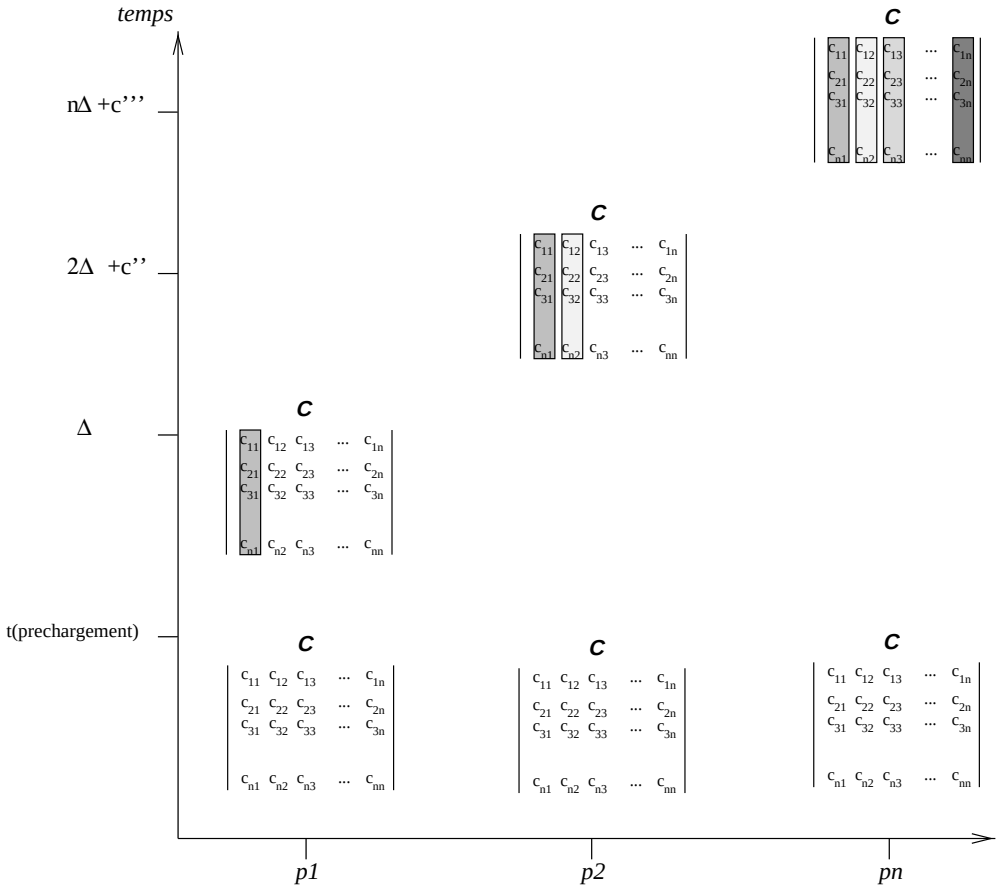


FIG. 6.8 – *Le déplacement de la matrice C entre les nœuds (SC_pré)*

Nous pouvons conclure que le préchargement de pages n'est pas très utile sur la cohérence *SC+temp* quand il existe une grande probabilité de modification de la page préchargée avant qu'une référence à elle ne soit générée. La mise à jour d'une page partagée sur la cohérence *SC+temp* active le mécanisme d'invalidation de copies. La page préchargée va ainsi être invalidée avant d'être référencée. Dans ce cas, la plupart des opérations de préchargement sont alors inutiles.

6.2.5 Cohérence à la libération avec préchargement

Pour bien évaluer le comportement du mécanisme de préchargement, nous l'avons aussi utilisé sous un modèle de cohérence relâché, la cohérence à la libération. De façon similaire au cas précédent, il nous a suffi de rajouter l'appel `diva_prefetch` au code de l'application.

Les résultats montrés sont ceux obtenus quand toutes les copies de C sont déjà en mémoire au moment de la première écriture à C.

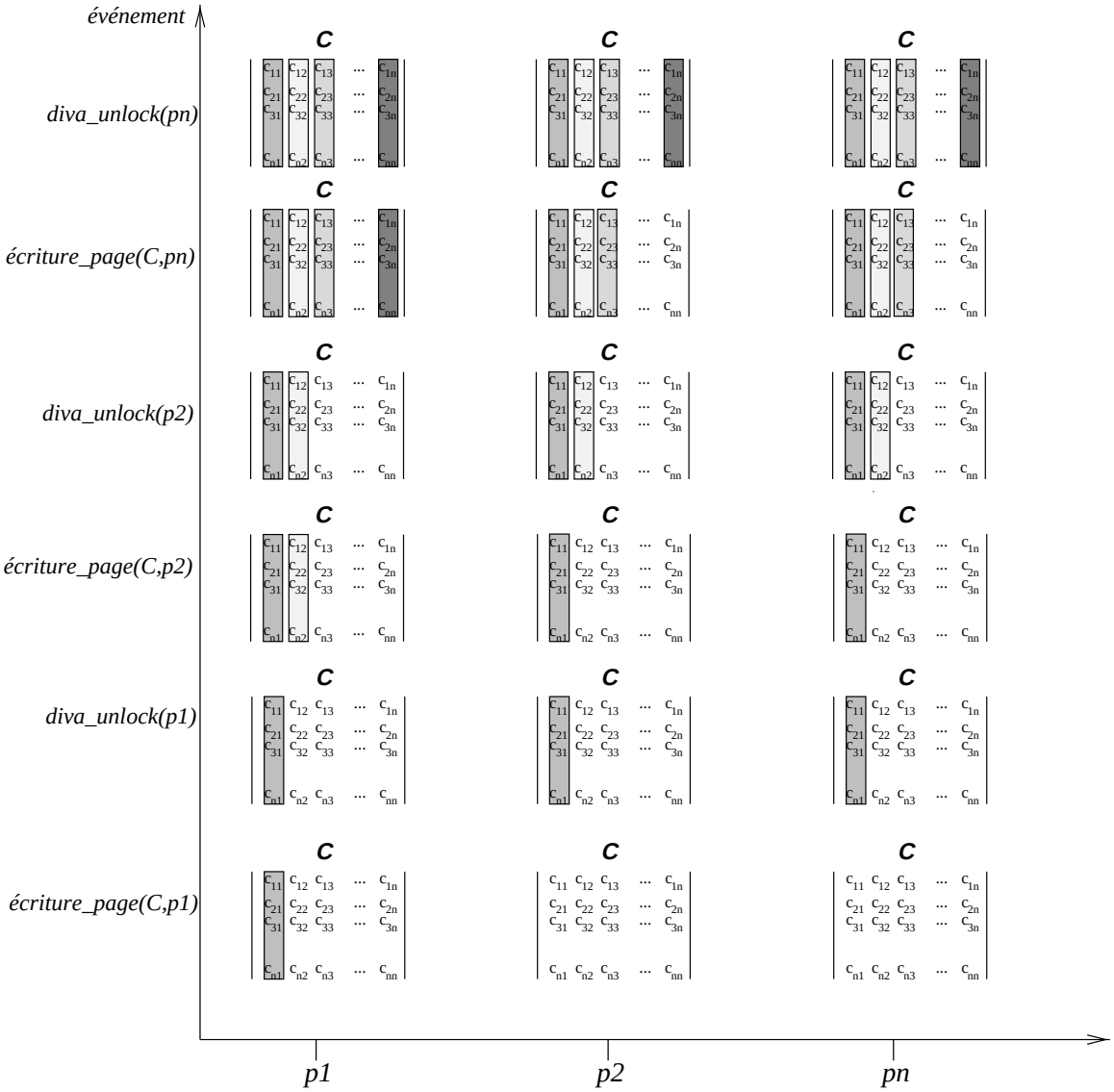


FIG. 6.9 – *Le déplacement de la matrice C entre les nœuds (RC_pré)*

Sur la cohérence à la libération, nous arrivons en effet à réduire le nombre

de défauts de page en utilisant le mécanisme de préchargement (voir figure 6.4). Néanmoins, le prix à payer est trop élevé. L'augmentation du nombre de copies de la matrice C entraîne une augmentation proportionnelle des mises à jour chaque fois qu'un processeur libère un verrou. Cette situation est mieux illustrée par la figure 6.9.

Maintenant, avant la première écriture sur la matrice C, nous avons déjà n copies de C dans le système. Chaque opération de libération de verrou cause alors $n-1$ opérations de mise à jour des copies de C.

Dans ce contexte, nous pouvons noter que l'opération de préchargement ne doit pas être utilisée pour les données fréquemment modifiées et gérées par un protocole de mise à jour.

6.2.6 Analyse comparative des modèles

La figure 6.10 montre le nombre d'opérations effectuées pour l'exécution de la multiplication de matrices sur plusieurs modèles et protocoles de cohérence.

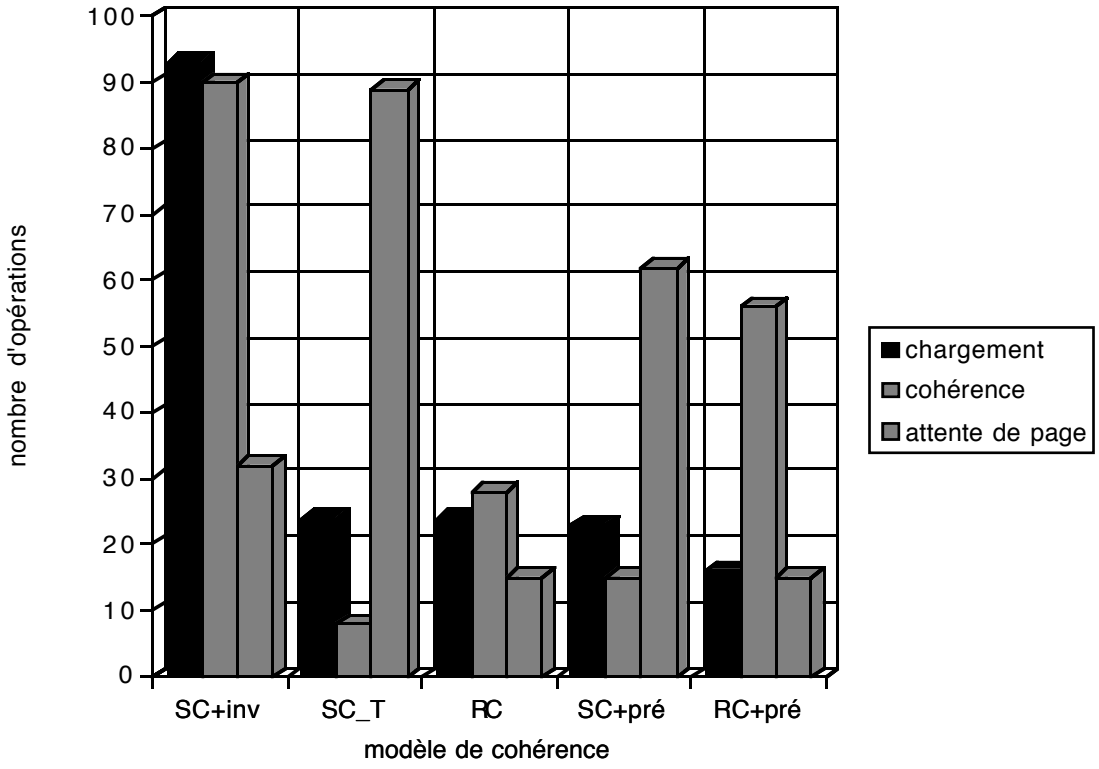


FIG. 6.10 – *Comparaison entre les modèles*

Les opérations considérées sont les opérations de chargement à la demande,

les opérations de cohérence et le temps inactif. Le temps inactif montré est le temps où le processeur attend son tour pour avoir accès à la page. Cette attente peut se produire en trois situations: le processeur attend le temps minimum Δ de permanence de la page sur un autre nœud, le processeur attend l'obtention d'un verrou et le processeur est un parmi les plusieurs processeurs qui veulent écrire sur la même page.

Pour la cohérence séquentielle avec le protocole d'invalidation (*SC+inv*), nous avons observé le plus grand surcoût. Le faux partage a fait croître le nombre d'opérations de cohérence. Comme les opérations de cohérence exécutées sont des invalidations, le nombre d'opérations de chargement de page croît de manière proportionnelle au nombre d'opérations de cohérence.

Quand la multiplication de matrices s'exécute sous le modèle *SC+temp*, nous notons une réduction importante du nombre d'opérations de cohérence par rapport au modèle $SC + inv$. Néanmoins, c'est clair sur cette histogramme que la réduction des opérations de cohérence a été obtenue par une augmentation significative du temps inactif des processus. Ceci est causé par l'utilisation du mécanisme de temporisation.

Sur la cohérence à la libération (*RC*), nous avons implanté un protocole de mise à jour. C'est la raison pour laquelle nous observons un nombre plus important d'opérations de cohérence pour le modèle *RC* que pour le modèle *SC+temp*.

L'utilisation du mécanisme de préchargement allié au modèle de cohérence *SC+temp* a causée une petite réduction du nombre d'opérations de chargement à la demande. En revanche, le nombre d'opérations de cohérence a augmenté par rapport à *SC+temp* à cause des invalidations des pages préchargées.

Bien que, pour la cohérence à la libération (*RC+pré*), l'utilisation du préchargement ait causé une réduction non négligeable du nombre d'opérations de chargement à la demande, une augmentation proportionnelle du nombre d'opérations de mise à jour a été produite.

Parmi les modèles étudiés, nous pouvons conclure que la cohérence à la libération (*RC*) présente le meilleur rapport entre la simplicité de programmation et les hautes performances. Il est notre sentiment que l'utilisation de la cohérence parresseuse à la libération nous donnerait des résultats encore plus satisfaisants. Les mesures restent à faire.

6.3 Performances brutes

L'objectif de ce paragraphe est de présenter quelques coûts intrinsèques mesurés sur le prototype du serveur $\mathcal{DIVA}$ implanté sur la machine Paragon décrite

dans le paragraphe 5.2. Les coûts présentés ici doivent être considérés plutôt d'une manière comparative car le prototype est encore en phase d'implantation et de tests. De plus, nous avons deux sortes de problèmes associés au noyau Mach. Le premier, c'est le protocole de communication entre le noyau et le serveur. D'après l'analyse réalisée par Condict et al. dans [C+93] les performances de ce protocole ne sont pas bonnes. D'autre part, l'implantation de Mach sur la machine Paragon présente des problèmes de performances et des erreurs de programmation qui prendront longtemps pour être résolus [Sea95]. Ainsi, les temps d'exécution qui y sont montrés n'ont pas été obtenus dans une situation idéale. Cependant, ceci peut alors servir comme un indicateur de coût comparatif entre les opérations.

Le tableau présenté dans la figure 6.11 montre le temps (en milliseconds) nécessaire au traitement de plusieurs types de défaut de page sur un modèle de cohérence séquentielle avec un protocole d'invalidation. Les temps ont été obtenus par l'exécution de la primitive `getclock` qui donne le temps en nanoseconds d'un horloge du système. Chaque situation de défaut de page a été mésurée 100 fois et les temps présentés ici sont les temps moyens obtenus. Le temps mésuré est le temps réel entre la réception du message de défaut de page envoyé par le noyau Mach et l'envoi de la page qui a causé le défaut au noyau.

type	**nœud**	**état de la page**	**temps(ms)**
READ ou WRITE	gestionnaire	FREE	1,52
READ ou WRITE	autre	FREE	3,08
protection	gestionnaire	READ(gest)	1,93
protection	autre	READ(autre)	3,87
READ ou WRITE	gestionnaire	READ(1 copie)	5,24
WRITE	gestionnaire	READ(10 copies)	9,29

FIG. 6.11 – *Les temps de traitement des défauts de page*

La première ligne présente le temps nécessaire au traitement d'un défaut de page sur le nœud gestionnaire de la page quand celle-ci se trouve dans l'état FREE. Cette situation est simple à traiter car les opérations exécutées sont exclusivement locales. Le gestionnaire met à jour l'entrée de la table de pages qui correspond à la page et rend une page remplie de zeros (*zero-filled*) au noyau Mach selon l'interface décrite dans le paragraphe 5.3. Ce type de défaut de page peut servir de base dans une analyse comparative de coûts.

La deuxième ligne correspond à un défaut de page sur une page libre mais dans un nœud différent du nœud gestionnaire. Ceci exige un échange de messages entre nœuds (2 messages). C'est la raison pour laquelle le temps mesuré ici est

plus grand que celui du cas précédent.

La troisième ligne correspond à un défaut de page de protection dans le nœud gestionnaire quand ce nœud détient la seule copie de la page dans le système. Un défaut de page de protection se produit par exemple quand un accès en écriture est généré pour une page accédée en lecture. Cette situation est aussi très simple et les opérations effectuées sont exclusivement locales. L'état de la page est modifié et l'accès en écriture est accordé. La différence entre le temps mésuré dans cette situation et celui mésuré dans la première situation vient du fait que les protocoles exécutés lors d'un défaut de page de protection et d'un défaut de page de chargement ne sont pas les mêmes. Hormis l'exécution de protocoles différents, la quatrième ligne correspond à la même situation que la deuxième.

L'avant dernière ligne correspond au cas où le gestionnaire est en défaut en lecture ou en écriture d'une page dont la seule copie est accédée en lecture sur un nœud distant. Dans ce cas, il n'existe pas de différence entre les temps mésurés pour les deux types de défaut de page (lecture ou écriture) car il faut dans les deux cas récupérer une copie de la page du processus utilisateur qui la référence.

La dernière ligne montre le coût associé à l'opération d'invalidation d'une page possédant 10 copies. Dans ce cas, 20 messages sont échangés avant que l'accès en écriture soit accordé.

Les temps obtenus pour le traitement du défaut de page sont, à première vue, très élevés. Ce résultat nous a beaucoup surpris.

Dans un premier temps, nous avions mis en cause la qualité du code du prototype de $\mathcal{DIVA}$. Nous avons alors comparé avec des systèmes similaires qui ont été implantés au-dessus de Mach. Dans la littérature, nous avons choisi trois systèmes:

- *l'interface Unix pour la mémoire partagée conçue par Tevanian et al. à CMU.* Les mesures présentées dans [T+87] ont été réalisées sur un MicroVAX II. Le temps plus petit, 1 ms, correspond à un défaut de page de protection.

- *le serveur de MVP proposé par Forin et al. à CMU.* Les mesures montrées dans [FBYR88] ont été réalisées sur un IBM RT-APC. Le temps d'un défaut de page de protection est de 1,1 ms.

- *MYOAN, le serveur de MVP conçu par Cabillic et al. à Rennes I.* Les mesures presentées dans [CPP94] ont été réalisées sur la même machine Intel Paragon sur laquelle $\mathcal{DIVA}$ a été implantée. Le temps le plus petit présenté est de 0,95ms pour le défaut de page "zero-fill" dans le nœud gestionnaire de la page.

Les performances obtenues par ces trois systèmes sont aussi de l'ordre de la milliseconde et, par conséquent, sont comparables à nos résultats.

Il nous semble alors évident que le surcoût présenté dans le traitement du défaut de page vient essentiellement du noyau Mach. Nous avons alors analysé le chemin parcouru dans le traitement d'un défaut de page pour le cas le plus simple, en l'occurrence, le défaut de page de "zero-fill" sur le nœud gestionnaire. Ce cas correspond à la première ligne de la figure 6.11.

Le paragraphe 5.3 décrit les opérations exécutées lors du traitement du défaut de page sur Mach. Nous pouvons voir sur la figure 5.11 que chaque traitement de défaut de page sur Mach comprend au moins deux messages locaux (du noyau au gestionnaire et vice-versa) et deux changements de contexte entre processus utilisateur (entre l'application et le gestionnaire et vice-versa). Dû a leur nature et complexité, les coûts associés à l'échange de messages et au changement de contexte sont élevés.

Les opérations effectuées par $\mathcal{DIVA}$ dans le cas étudié sont très simples. Elles se résument à allouer de la place pour une page de 8koctets, remplir la page de zeros, mettre à jour l'entrée correspondante de la table de pages et envoyer la page au noyau Mach.

D'après cette analyse, c'est clair que des systèmes tels que Mach ne sont pas adaptés aux machines parallèles de grande taille. Bien que l'utilisation du support que ces systèmes offrent soit très commode pour l'utilisateur, le prix à payer en performances est trop élevé.

Cette constatation renforce l'approche adoptée au sein de notre équipe qui consiste à construire des mécanismes génériques et corrects sur les couches très basses du noyau ParX (voir paragraphe 3.3). Sur ce noyau minimal, différentes interfaces peuvent être offertes aux applications parallèles. L'application choisit donc le support minimum adapté à ses besoins. Ainsi, nous sommes capables d'offrir un support logiciel adapté aux applications toujours en gardant les performances.

Pour le cas de $\mathcal{DIVA}$, nous avons uniquement besoin d'un mécanisme correct de routage de messages et d'un support matériel pour la détection des défauts de page. Comme ParX a été initialement implanté sur une machine à base de transputers, où le support pour la gestion de la mémoire virtuelle est inexistant, il nous a été impossible d'implanter $\mathcal{DIVA}$ sur ParX. Nous attendons alors le portage de ParX sur une architecture qui offre le support matériel pour la mémoire virtuelle pour faire à notre tour le portage de $\mathcal{DIVA}$ sur ParX. L'implantation de $\mathcal{DIVA}$ sur ParX doit présenter des résultats nettement meilleurs que ceux présentés dans sa version originale.

6.4 Conclusion

L'exemple de la multiplication des matrices sert à illustrer l'impact causé par le modèle de cohérence sur les performances d'une application parallèle. Nous avons intentionnellement choisi une application très simple où le passage entre l'algorithme séquentiel et l'algorithme parallèle est presque immédiat.

Il a été montré que, même pour une telle application, les performances sont fortement dégradées quand une implantation "naïve" du modèle de cohérence, en l'occurence la cohérence séquentielle, est utilisée.

Une autre implantation du même modèle (la cohérence séquentielle avec temporisation) nous donne des performances plus acceptables pour la même application. Néanmoins, même cette implantation n'est pas capable d'exprimer tout le parallélisme de l'algorithme car elle touche aux limites imposés par le modèle de cohérence lui-même.

Nous avons alors changé le modèle de cohérence et utilisé la cohérence à la libération. Notre implantation utilise un protocole de mise à jour. L'utilisation de ce protocole conduit à plusieurs opérations superflues de mise à jour de pages. Une autre implantation de la cohérence à la libération, la cohérence paresseuse, peut corriger ce problème.

Notre experience avec la multiplication des matrices nous permet de tirer les deux conclusions suivantes:

1. Le choix du modèle de cohérence a un rapport direct avec les performances d'une application parallèle qui s'exécute sur un système à mémoire virtuelle partagée.

2. Une implantation efficace du modèle de cohérence choisi est primordiale pour que le modèle soit bien exprimé.

Notre experience avec $\mathcal{DIVA}$ comme support pour l'exécution de modèles de cohérence multiples a été extremement satisfaisante. Outre sa fonction de gestionnaire de mémoire partagée, $\mathcal{DIVA}$ est un outil très puissant pour l'analyse du comportement des applications parallèles par rapport aux modèles de cohérence. Le changement du modèle de cohérence ne cause que des modifications mineures dans le code de l'application. En général, il suffit de changer la primitive de spécification du modèle de cohérence car la syntaxe des primitives de synchronisation reste inalterée.

La programmation d'une nouvelle implantation d'un modèle de cohérence déjà existant est aussi une tâche simple sur $\mathcal{DIVA}$. L'implantation du modèle

$SC + temp$ à partir du modèle SC ne nous a pris que quelques heures de programmation.

L'analyse du comportement de la multiplication de matrices sous une politique de préchargement nous a permis aussi de tirer des conclusions très intéressantes. Nous avons vu que le coût du préchargement est très élévé quand les opérations de cohérence sont exécutées fréquemment pour les donnés préchargées avant que la référence à eux ne soit générée.

Bien que l'analyse du comportement de la multiplication de matrices ne soit pas favorable au préchargement, nous croyons que d'autres types d'application bénéficieront de ce mécanisme. En particulier, l'utilisation du préchargement pour les données qui sont rarement ou jamais modifiées peut causer une réduction importante du temps d'exécution de l'application.

Chapitre 7

Conclusion et Perspectives

Conclusion

Le comportement des programmes qui admettent le partage des données entre plusieurs processus dépend de l'ordre dans lequel les accès aux données partagées sont perçus par les processus. Dans les machines monoprocesseur et même dans les machines parallèles à mémoire commune, nous arrivons avec une relative facilité à donner aux processus qui composent le programme la même vision de l'ordre des accès aux données partagées. Ceci peut être fait grâce à l'unicité de la mémoire physique.

Dans les machines parallèles sans mémoire commune ou dans les machines qui comportent plusieurs caches, la vision identique de l'ordre des accès aux données partagées peut encore être offerte aux processus. Hélas, cette vision unique est obtenue par une grande dégradation des performances. Cette dégradation atteint des niveaux difficilement acceptables et met en cause l'utilisation des données partagées dans un environnement parallèle à mémoire distribuée.

Cette thèse est une étude sur le comportement de la mémoire partagée pour les applications parallèles qui s'exécutent sur une machine sans mémoire commune. Dans ce cadre, nous avons étudié plusieurs modèles de la mémoire, ou modèles de cohérence de la mémoire. Ces modèles définisent l'ensemble des ordres des accès à la mémoire partagée qui peuvent être perçus par chaque processus parallèle. En général, les modèles qui posent beaucoup de restrictions dans l'ordre des accès donnent lieu à des implantations peu performantes. En contrepartie, les modèles qui admettent un ensemble important d'ordres des accès peuvent produire des résultats avec lesquels il est difficile de raisonner.

Dans le choix du modèle de cohérence de la mémoire le plus adapté, les caractéristiques des accès aux données présentées par les applications parallèles jouent

un rôle fondamental. De façon similaire, l'attente du programmeur à propos des résultats qui peuvent être produits par son programme doit être considéré dans le choix du modèle. Il est évident alors que l'application doit être libre pour choisir le support de cohérence le plus adapté à ses besoins et attentes.

L'objectif de cette thèse était de montrer la relation étroite entre le choix d'un modèle de cohérence de la mémoire et les performances des applications parallèles. Pour mettre en évidence cette relation, nous avons proposé un système à mémoire virtuelle partagée qui permet le choix par l'application entre plusieurs modèles de cohérence de la mémoire.

Très vite, nous avons remarqué que la flexibilité apportée par le choix entre différents modèles de cohérence n'était pas suffisante. Il faudrait aussi offrir à l'utilisateur une manière de définir des modèles autres que ceux existants dans notre système. Ainsi, nous avons abouti à un mécanisme très flexible où l'ajout de nouveaux modèles de cohérence est aussi possible.

Nous avons conçu un système à mémoire virtuelle partagée, appelé $\mathcal{DIVA}$, qui offre le support pour l'exécution de modèles de cohérence de la mémoire multiples. La conception de ce système a fait ressortir deux problèmes qui sont rarement traités dans le domaine de la mémoire virtuelle partagée.

Le premier problème consiste à définir où placer une page qui doit être supprimée de la mémoire locale. L'approche traditionnelle consiste à écrire cette page sur le disque. Nous avons alors essayé cette approche, ce qui a généré un trafic important de données quand l'architecture parallèle cible disposait d'un système d'entrée/sortie centralisé. Afin de réduire le trafic dans le réseau d'interconnexion, nous avons proposé un algorithme de remplacement qui place les pages à supprimer dans la mémoire d'un nœud distant. Nous avons aussi observé que l'opération de chargement de pages est un des facteurs principaux de l'augmentation du temps d'exécution d'une application parallèle. Afin de réduire l'impact du chargement de pages sur le temps d'exécution de l'application, nous avons proposé un mécanisme de préchargement de pages. Ces deux mécanismes prennent toujours en compte l'existence de modèles de cohérence multiples. Ils ont aussi servi de support au module de gestion de modèles.

Le système à mémoire virtuelle partagée résultant est un système où les problèmes de gestion du partage et de gestion de la mémoire virtuelle sont traités.

L'impossibilité de l'implantation de $\mathcal{DIVA}$ sur un support logiciel plus adapté aux environnements parallèles (e.g ParX) nous a amenés à choisir le système Mach pour la mise en œuvre d'un prototype. Les mauvaises performances présentées par le prototype sont dues en grande partie aux nombreuses couches logicielles et protocoles implantés par le système Mach. Une première conclusion de notre travail est alors que des systèmes tels que Mach, bien que portables et structurés, sont inadéquats aux machines parallèles de grande taille.

L'analyse d'une application parallèle simple a fait ressortir les différences entre les performances obtenues quand celle-ci s'exécute sur différents modèles de cohérence de la mémoire. Ceci vérifie notre hypothèse initiale: le choix des contraintes de cohérence les plus adaptées à une application dépend de l'application.

Il nous semble alors que les modèles de cohérence multiples sont une des caractéristiques fondamentales à être incorporées aux systèmes à mémoire virtuelle partagée futurs pour offrir aux applications parallèles un modèle de programmation à la fois simple et performant.

Néanmoins, en évaluant des applications parallèles, nous avons constaté qu'il existe en fait deux niveaux de choix en ce qui concerne les modèles de cohérence de la mémoire. D'abord, il faut choisir le modèle de cohérence le plus adapté à l'application parallèle. Il nous semble plus réaliste que ce choix soit fait en deux temps. Dans un premier temps une analyse de l'application permet de selectionner quelques modèles "probablement" adaptés à l'application et l'experimentation permet de faire le choix définitif. Dans cette deuxième étape, un outil tel que $\mathcal{DIVA}$ est fort utile.

Ayant choisi le modèle de cohérence, il faut choisir une implantation efficace de ce modèle. Une implantation trop restrictive peut détruire la grande partie des avantages apportés par le modèle. Dans cette étape aussi, l'expérimentation est fortement souhaitable et les mécanismes offerts par $\mathcal{DIVA}$ sont d'une extrême utilité.

Perspectives

Le présent travail de recherche ouvre un grand nombre de perspectives. La première et la plus immédiate est la migration de $\mathcal{DIVA}$ vers une machine sur laquelle le noyau ParX s'exécute. Cette migration permettra d'effectuer des mesures réelles des performances des mécanismes offerts par notre système. Les mesures obtenues dans l'actuelle version de $\mathcal{DIVA}$ sont dominées par des coûts introduits par le support d'exécution choisi, en l'occurrence le système Mach.

Il nous faut dans une prochaine étape réaliser des évaluations sur un ensemble significatif et varié d'applications réelles avec un grand degré de parallélisme. Il est aussi nécessaire d'évaluer à fond le comportement des mécanismes proposés pour le remplacement et le préchargement des pages par rapport aux différents modèles de cohérence de la mémoire.

En ce qui concerne l'évolution de $\mathcal{DIVA}$, il faut étudier la possibilité d'implanter un mécanisme de gestion de modèles de cohérence à deux niveaux. Dans sa présente version, $\mathcal{DIVA}$ ne traite qu'un niveau de définition. Deux implantations différentes d'un même modèle sont définies comme deux modèles différents.

Une implantation à deux niveaux permettrait une réduction considérable de l'effort de programmation nécessaire pour réaliser des nouvelles implantations d'un même modèle de cohérence à $\mathcal{DIVA}$.

Un autre axe de recherche est celui de la conversion automatique entre la spécification du modèle de cohérence de la mémoire et les méthodes de description de modèles acceptées par $\mathcal{DIVA}$. Bien qu'une telle conversion semble improbable, nous croyons qu'il est fort possible de réaliser une conversion semi-automatique où une grande partie de la conversion des restrictions en méthodes de cohérence soit réalisée automatiquement. L'utilisateur serait alors responsable de compléter quelques détails spécifiques à son modèle qui n'ont pas été traités.

Les outils d'analyse des performances des applications parallèles peuvent aussi utiliser les mécanismes proposés par notre système pour évaluer le comportement d'une application exécutée sur plusieurs modèles de cohérence. Ce type d'analyse peut séparer les problèmes de performance qui sont inhérents à l'application et ceux qui sont causés par une association mauvaise entre application parallèle et modèle de cohérence.

Annexe A

Etude de cas: la cohérence séquentielle

L'objectif de ce paragraphe est d'illustrer sur un exemple comment un modèle de cohérence de la mémoire peut être intégré à notre système. Nous avons choisi d'étudier la cohérence séquentielle à cause de sa relative simplicité d'implantation.

Le modèle de cohérence séquentielle (SC) a été défini dans le chapitre 2.2. Dans [SD87], Scheurich et Dubois ont donné une condition suffisante pour l'implantation correcte de ce modèle. Cette condition est satisfaite si tous les processeurs initient les accès à la mémoire partagée selon l'ordre du programme et une opération n'est initiée que quand l'opération précédente est terminée.

Pour l'implantation de la cohérence séquentielle, nous utilisons un protocole MRSW d'invalidation.

Les ensembles $\mathcal{O}$ et $\mathcal{P}$ définis sont:

$\mathcal{O} = \{r, w\}$ et $\mathcal{P} = \{methode\_r, methode\_w\}$

Les principales structures de données manipulées par les protocoles sont:

1. **Table de page** - plus spécifiquement, les champs manipulés sont l'état de la page et son <copyset>.

2. **Liste des pages en défaut** - cette liste contient les pages qui ont subi un défaut de page dont le traitement est en cours.

3. **Liste d'opérations inachevées** - la page pour laquelle une requête de chargement a été envoyée (CHARGE_PAGE_SC) est stockée dans cette liste pour son gestionnaire jusquà la fin de l'opération de chargement.

Les méthodes associées aux défauts de page en lecture et écriture sont décrits dans les paragraphes qui suivent.

methode_r L'exécution de la `methode_r` est initiée à la suite d'un défaut de page en lecture pour la page p. La procédure `lecture_sc()` est exécutée sur le processeur qui a subi le défaut de page. Les messages échangés par le protocole sont au nombre de six: CHARGE_PAGE_R_SC, ACK_PAGE_SC, PAGE_SC, SEND_PROT_PAGE_SC, CHARGE_PAGE_DISK_SC, SEND_PAGE_SC.

Les messages échangés lors de l'exécution du protocole sont montrés sur la figure A.1.

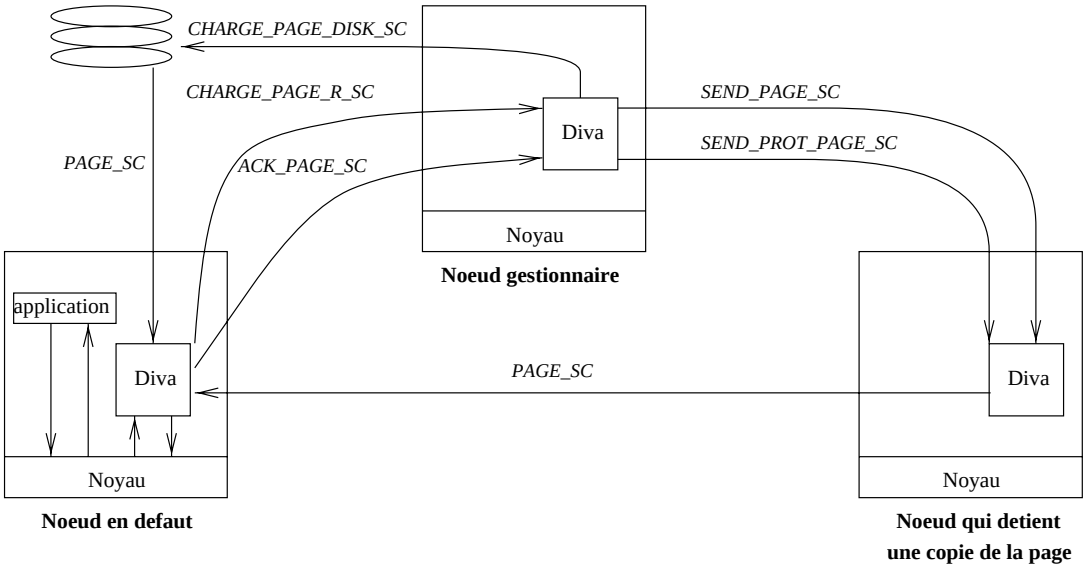


FIG. A.1 – *Le protocole de lecture sur la cohérence séquentielle*

Nous décrivons ci-dessous, de manière simplifiée, les actions prises par la procédure `lecture_sc()` et pour la réception des messages associés.

`lecture_sc()`
- ◇ obtient le numéro de la page qui a causée le défaut;
- ◇ envoie le message CHARGE_PAGE_R_SC au gestionnaire de la page;
- ◇ l'application reste bloquée en attendant la page;
- ◇ le numéro de la page est mis dans une liste de pages en défaut;

Les actions prises lors de la réception d'un message du protocole sont:

CHARGE_PAGE_R_SC:
* message reçu par le gestionnaire de la page
  ◇ case `etat`(page)

    - FREE:
      - la valeur READ est affectée à l'état de la page;
      - le processeur qui a envoyé le message est additionné au *copyset* de la page;
      - le message CHARGE_PAGE_DISQUE_SC est envoyé au disque;

    - READ:
      - la valeur MREAD est affectée à l'état de la page;
    - MREAD:
      - le processeur qui a envoyé le message est additionné au *copyset* de la page;
      - le message SEND_PAGE_SC est envoyé à un processeur du *copyset(page)*;
      - le processeur demandeur est additonné au *copyset* de la page;

    - WRITE:
      - la valeur MREAD est affectée à l'état de la page;
      - le message SEND_PROT_PAGE_SC est envoyé à un processeur du *copyset(page)*;
      - le processeur demandeur est additonné au *copyset* de la page;

  ◇ la page est mise dans une file d'opérations inachevées.

SEND_PAGE_SC:
* message reçu par un noeud qui détient une copie de la page.
  ◇ la page est envoyée au noeud demandeur;

SEND_PROT_PAGE_SC:
* message reçu par un noeud qui détient une copie de la page.
  ◇ la page est protégée contre l'écriture;
  ◇ la page est envoyée au noeud demandeur;

ACK_PAGE_SC:
* message reçu par le gestionnaire.
  ◇ la page est rétirée de la file d'opérations inachevées;

PAGE_SC:
* message reçu par le noeud en défaut.
  ◇ dès que la page arrive au noeud demandeur, elle est rendue au noyau;
  ◇ la page est rétirée de la liste de pages en défaut;
  ◇ le message ACK_PAGE_SC est envoyé au gestionnaire;

CHARGE_PAGE_DISQUE_SC:
* message reçu par le système d'entrée/sortie
  ◇ la page est envoyée du disque au noeud qui a généré le défaut;

methode_w L'exécution de la `methode_w` est initiée à la suite d'un défaut de page en écriture pour la page p. La procédure `ecriture_sc()` est exécutée sur le processeur qui a subi le défaut de page. Les messages échangés par le protocole sont au nombre de sept: CHARGE_PAGE_W_SC, INV_PAGE_SC, INV_SEND_PAGE_SC, ACK_INV_SC, ACK_PAGE_SC, PAGE_SC CHARGE_PAGE_DISK_SC.

Les messages échangés lors de l'exécution de la méthode sont montrés sur la figure A.2.

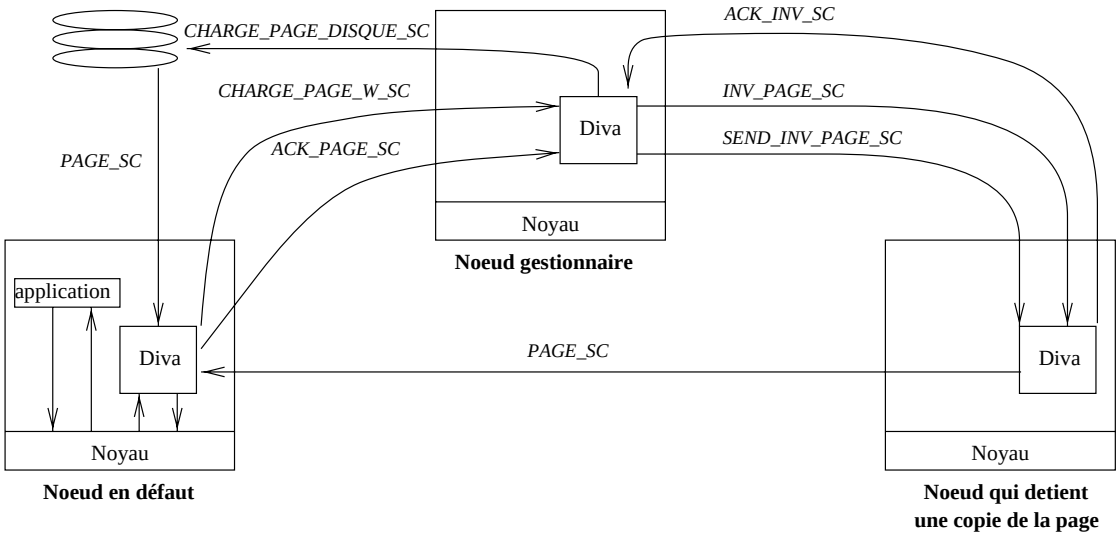


FIG. A.2 – *Le protocole d'écriture sur la cohérence séquentielle*

Nous décrivons ci-dessous, de manière simplifiée, les actions prises par la procédure `ecriture_sc()` et pour la réception des messages associés.

`ecriture_sc()`
- ◇ obtient le numéro de la page qui a causée le défaut;
- ◇ envoie le message CHARGE_PAGE_W_SC au gestionnaire de la page;
- ◇ reste bloqué en attendant la page;
- ◇ le numéro de la page est mis dans une liste de pages en défaut;

Les actions prises lors de la réception d'un message du protocole sont:

CHARGE_PAGE_W_SC:
*message reçu par le gestionnaire.
◇ case `etat`(page)

- FREE:
  - la valeur WRITE est affectée à l'état de la page;
  - le processeur qui a envoyé le message est additionné au *copyset* de la page;
  - le message CHARGE_PAGE_DISQUE_SC est envoyé au disque;
- READ:
- MREAD:
- WRITE:
  - envoie le message SEND_INV_PAGE_SC à un processeur qui détient la page;
  - a tous les autres processeurs de *copyset(page)* envoie le message INV_PAGE_SC;
  - attend l'acquittement de tous ces messages (ACK_INV_SC);
  - la valeur WRITE est affectée à l'état de la page;
  - le processeur demandeur est additonné au *copyset* de la page;

◇ la page est mise dans une file d'opérations inachevées.

SEND_INV_PAGE_SC:
*message reçu par un noeud qui détient une copie de la page.
◇ la page est envoyée au noeud demandeur à travers le message PAGE_SC;
◇ la copie locale de la page est invalidée;
◇ le message ACK_INV_SC est envoyé au gestionnaire;

INV_PAGE_SC:
*message reçu par un noeud qui détient une copie de la page.
◇ la copie locale de la page est invalidée;
◇ le message ACK_INV_SC est envoyé au gestionnaire;

ACK_INV_SC:
*message reçu par le gestionnaire.
◇ le processeur qui a envoyé le message est rétiré du *copyset* de la page;

Les messages PAGE_SC, CHARGE_PAGE_DISQUE_SC et ACK_PAGE_SC se comportent exactement comme dans le cas de défaut de page en écriture.

Le fichier de configuration utilisé pour l'intégration de la cohérence séquentielle à $\mathcal{DIVA}$ est montré dans la figure A.3.

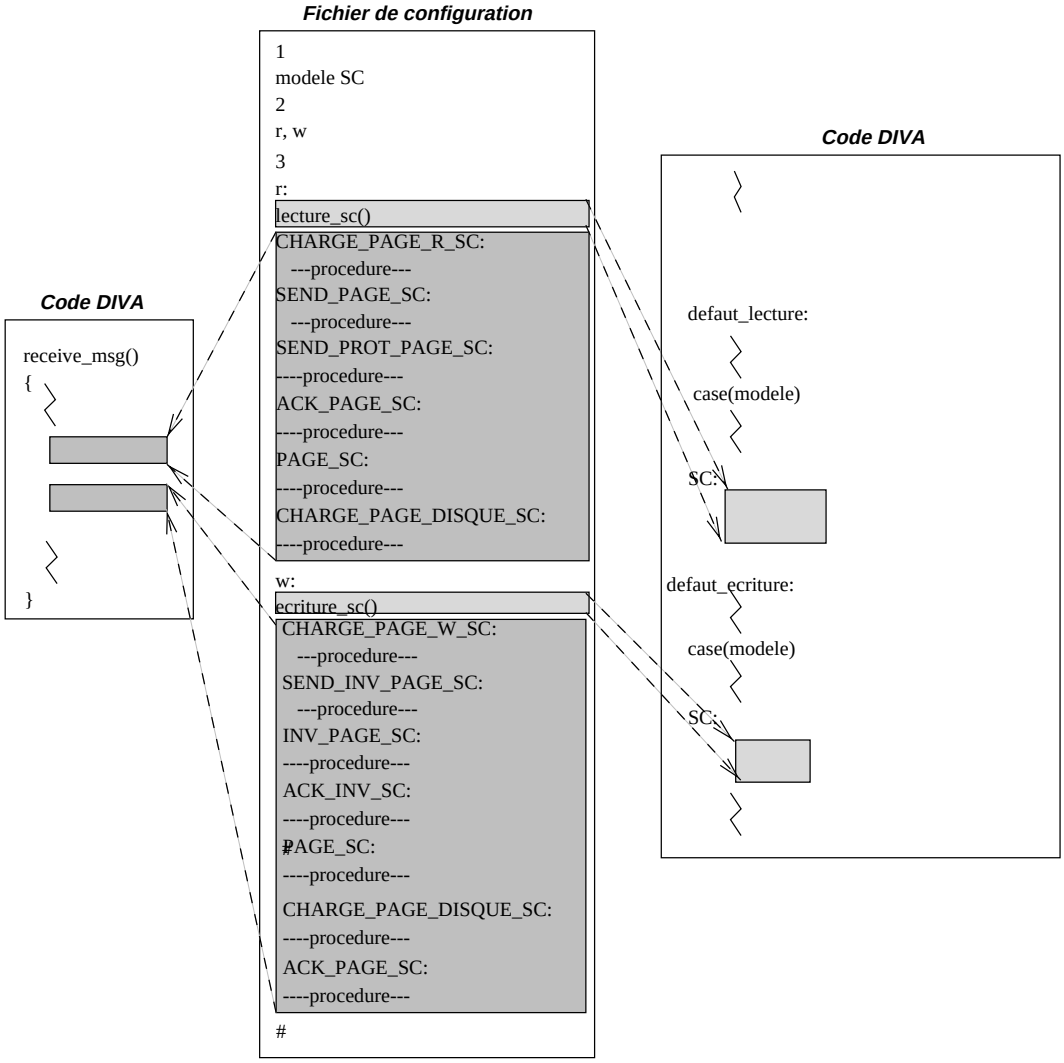


FIG. A.3 – *L'implantation de la cohérence séquentielle*

Bibliographie

[A+92] M. Ahamad et al. The power of processor consistency. Technical Report GIT-CC-92/34, Georgia Institute of Technology, 1992.

[A+95] R. C. Agarwal et al. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, pages 152–160, 1995.

[ABM93] Y. Afek, G. Brown, and M. Merrit. Lazy caching. *ACM Transactions on Programming Languages and Systems*, pages 182–205, 1993.

[Adv93] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, University of Wisconsin-Madison, 1993.

[AH90] S. Adve and M. Hill. Weak ordering: a new definition. In *17th Annual International Symposium on Computer Architecture*, pages 2–11, 1990.

[AHJ90] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed shared memory. Technical Report GIT-CC-90/49, Georgia Institute of Technology, 1990.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamim/Cummings Publishing Company, Inc., 1991.

[BCZ91] J. K. Bennet, J. C. Carter, and W. Zwaenepoel. *Lecture Notes on Computer Science 563*, chapter Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency, pages 56–60. Springer-Verlag, 1991.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, pages 78–101, 1966.

[BH90] L. Borrmann and M. Heidieckerhoff. A coherency model for virtually shared memory. In *1990 International Conference on Parallel Processing*, pages 252–257, 1990.

[BKT92] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaun. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, 1992.

[BM94a] A. Balaniuk and T. Muntean. Adaptive page replacement in the diva shared virtual memory parallel server. In *Journées de Jeunes Chercheurs en Architectures de Machines et Systèmes*, pages 223–232, December 1994.

[BM94b] A. Balaniuk and T. Muntean. Programming with shared data in parallel loosely coupled machines: The shared virtual memory approach. In *IEEE/USP International Workshop on High Performance Computing*, pages 129–142, March 1994.

[BM96] A. Balaniuk and T. Muntean. Diva: un serveur de memoire virtuelle partagee pour les machines paralleles. In *Journées de Recherche sur la Mémoire Partagée Répartie*, May 1996.

[BR90] R. Bisiani and M. Ravishankar. Plus: A distributed shared memory system. In *17th Annual International Symposium on Computer Architecture*, pages 115–124, 1990.

[BZ91a] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91170, Carnegie Mellon University, 1991.

[BZ91b] B.N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming for distributed memory multiprocessors. Technical Report CMU-CS-91170, Carnegie-Mellon University, 1991.

[BZS93a] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *COMPCOM*, pages 34–42, 1993.

[BZS93b] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *COMPCON*, 1993.

[C+93] M. Condict et al. Optimising performance of mach-based systems by server co-location: a detailed design. Technical Report server_colocation, OSF Research Institute, 1993.

[Car93] J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. PhD thesis, Rice University, September 1993.

[Cas95] H. Castro. *Les Entrés/Sorties dans les Architectures Massivement Parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, 1995.

[CF89] A. L. Cox and R. J. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor: Experiences with platinum. In *12th ACM Symposium on Operating Systems Principles*, pages 32–43, 1989.

[CF90] P. M. Clancey and J. M. Francioni. Distribution of pages in a distributed virtual memory. In *1990 International Conference on Parallel Processing*, pages 258–263, 1990.

[CG92] D. Comer and J. Griffioen. Efficient order-dependent communication in a distributed virtual memory environment. In *Usenix Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 249–262, 1992.

[CMP95] G. Cabillic, G. Muller, and I. Puaut. The performance of consistent checkpointing in distributed shared memory systems. Technical Report PI-924, IRISA, 1995.

[CMW93] H. Castro, A. Elleuch T. Muntean, and P. Waille. Generic microkernel architecture for the paros parallel operating system. In *World Transputer Congress - Transputer Application and Systems*, pages 19–28, 1993.

[CPP94] G. Cabillic, T. Priol, and I. Puaut. Myoan: An implementation of the koan shared virtual memory on the intel paragon. Technical Report 812, IRISA, 1994.

[D+91] M. Dubois et al. Delayed consistency and its effects on the miss rate of parallel programs. Technical Report CENG-92-11, CENG, 1991.

[Den70] P. Denning. Virtual memory. *IEEE Computer Surveys*, pages 153–189, 1970.

[Dij65] E. J. Dijkstra. *Programming Languages*, chapter Co-operating Sequential Processes, page 11. Londres: Academic Press, 1965.

[DM93] R. Despons and T. Muntean. Constructing correct protocols for a diffusion virtual machine in message passing parallel architectures. In *World Transputer Congress - Transputer Application and Systems*, pages 111–119, 1993.

[Dra92] R. Draves. Page replacement and reference bit emulation in mach. Carnegie-Mellon University, 1992.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[DVJ90] S. Gjessing D. V. James, A. T. Laundrie. Scalable coherent interface. *IEEE Computer*, pages 74–77, 1990.

[EHH92] A. Landin E. Hagerstein and S. Haridi. Ddm - a cache-only memory architecture. *IEEE Computer*, pages 44–54, 1992.

[ESP91] ESPRIT Project 2528 - SUPERNODE II. *SNOS Kernel Specifications*, 1991.

[FBYR88] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation and performance evaluation of a distributed shared memory server for mach. Technical Report CMU-CS-88-165, Carnegie Mellon University, 1988.

[FP89] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *14th ACM Symposium on Operating Systems Principles*, pages 211–221, 1989.

[G+90] K. Ghararchorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th Annual International Symposium on Computer Architecture*, pages 15–25, 1990.

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *4th International Conference on Architectural Support for Programming Languages and Systems*, pages 245–257, April 1991.

[Gia93] S. Giancone. Un modèle de programmation parallèle mixte basé sur l'échange de messages et les données partagées. Technical Report Mémoire CNAM, INPG, 1993.

[Goo89] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[HA90] P. Hutto and M. Ahmad. Slow memory: Weakening consistency to enhance concurrency on distributed shared memories. In *10th international Conference on Distributed Computing Systems*, pages 302–309, 1990.

[HB84] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing*. MacGraw-Hill, 1984.

[HERV93] G. Heiser, K. Elphinstone, S. Russel, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SCSE-9314, University of New South Wales, 1993.

[Hol89] M. A. Holliday. Page size and migration daemons in local/remote architectures. In *ASPLOS-III*, pages 104–112, 1989.

[HS92] A. Heddaya and H. Sinha. An overview of mermera: a system formalism for non-coherent distributed parallel memory. Technical Report BU-CS-92-009, Boston University, 1992.

[HS93] A. Heddaya and H. Sinha. An implementation of mermera: a shared memory system that mixes coherence with non-coherence. Technical Report BU-CS-93-006, Boston University, 1993.

[HSMB91] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, pages 415–422, 1991.

[Int93] Intel Corporation. *Paragon System Administrator's Guide*, 1993.

[Int95] Intel Corporation. *Paragon System User's Guide*, 1995.

[JA93] R. John and M. Ahamad. Causal memory: Implementation, programming support and experiences. Technical Report GIT-CC-93/10, Georgia Institute of Technology, 1993.

[JH93] J.Huck and J. Hays. Architectural support for translation table management in large address space systems. In *20th Annual international Symposium on Computer Architectures*, pages 39–50, 1993.

[JJ92] N. C. Juul and E. Jul. *Lecture Notes on Computer Science 637*, chapter Comprehensive and Robust Garbage Collection in a Distributed System, pages 103–115. Springer-Verlag, 1992.

[Ka94] J. Kuskin and al. The stanford flash multiprocessor. In *21st Annual International Symposium on Computer Architecture*, pages 302–313, 1994.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *19th Annual Symposium on Computer Architecture*, pages 13–21, 1992.

[KDCZ93] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Threadmarks: Distributed shared memory on standard workstations and operating systems. Technical Report COMP TR93-214, Rice University, 1993.

[KFJ94] P. T. Koch, R. Fowler, and E. Jul. Message-driven relaxed consistency in a software distributed shared memory. In *First Symposium on OSDI*, pages 75–85, November 1994.

[KJe91] S. J. Eggers T. E. KJeremiassen. Eliminating false sharing. In *1991 International Conference on Parallel Processing*, pages 377–381, 1991.

[KK92] O. Kremien and J. Kramer. Mehodical analysis of adaptive load sharing algorithms. *IEEE Transactions on Parallel and Distibuted Systems*, pages 747–760, 1992.

[KNA93] P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. Technical Report GIT-CC-93/04, Georgia Institute of Technology, 1993.

[Lah93] Z. Lahjomri. *Conception et Réalisation d'un Mécanisme de Mémoire Virtuelle Partagée sur un Machine Multiprocesseur à Mémoire Distribuée*. PhD thesis, Université de Rennes I, September 1993.

[Lam78] L. Lamport. Time, clocks and ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, pages 690–691, 1979.

[Lam86] L. Lamport. *Distributed Computing*, chapter On Interprocess Communication; Part I: Basic Formalism, pages 77–85. 1986.

[Lan91] Y. Langue. *PARX: Architecture de Noyau de Système d'Exploitation Parallèle*. PhD thesis, Institut National Polytechnique de Grenoble, December 1991.

[LE91] R. P. LaRowe and C. S. Ellis. Placement policies for numa multiprocessors. *Journal of Parallel and Distributed Computing*, pages 112–129, 1991.

[LEH92] R. P. LaRowe, C. S. Ellis, and M. A. Holliday. Evaluation on numa memory management through modeling and measurements. *IEEE Transactions on Parallel and Distributed Systems*, pages 686–701, 1992.

[Les93] B. P. Lester. *The Art of Parallel Programming*. Prentice Hall International Editions, 1993.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Architectures*. PhD thesis, Yale University, 1986.

[Li88] K. Li. Ivy: a shared virtual memory system for parallel processing. In *1988 International Conference on Parallel Processing*, pages 94–101, 1988.

[Lil93] D. J. Lilja. Cache coherence in large-scale shared memory multiprocessors: Issues and comparisons. *ACM Computer Surveys*, 25(3):303–335, 1993.

[LKBT92] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaun. Comparison of two paradigms for distributed shared memory. *Software - Practice and Experience*, 22(11):985–1010, November 1992.

[LLJ+93] D. Lenosky, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[LP92] Z. Lahjomri and T. Priol. *Lecture Notes on Computer Science 634*, chapter KOAN: a Shared Virtual Memory for the iPSC/2 Hypercube, pages 442–452. Springer Verlag, September 1992.

[LS88] R. Lipton and J. Sandberg. Pram: a scalable shared memory. Technical Report 180-88, Princeton University, 1988.

[LS89] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *1989 International Conference on Parallel Processing*, pages 125–132, 1989.

[M+89] T. Muntean et al. Parx: a parallel operating system for transputer-based machines. In *Occam Users Group 10 - Applying transputer based parallel machines*, pages 115–141, 1989.

[Moh93] A. Mohindra. *Issues in the Design of Distributed Shared memory Systems*. PhD thesis, Georgia Institute of Technology, 1993.

[Mos93] D. Mosberger. Memory consistency models. *Operating Systems Reviews*, pages 18–26, 1993.

[MSG92] B. Mukherjee, K. Schwan, and P. Gopinath. A survey of multiprocessor operating system kernels. Technical Report GIT-CC-92/05, Georgia Institute of Technology, 1992.

[NA91] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, pages 57–61, 1991.

[NL91] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, 1991.

[Ope92a] Open Software Foundation and Carnegie-Mellon University. *Mach 3 Kernel Principles*, 1992.

[Ope92b] Open Software Foundation and Carnegie-Mellon University. *Mach 3 Server Writer's Guide*, 1992.

[Pan80] V. Pan. New fast algorithms for matrix operations. *SIAM Journal on Computing*, pages 321–342, 1980.

[R+88] R. Rashid et al. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, pages 896–907, 1988.

[Rab93] F. Rabii. The process management architecture of osf/1 ad version 2. OSF Research Institute, September 1993.

[Roc90] B. Rochat. Design and implementation of a multi-cache system on a loosely coupled processor. In *5th Distributed Memory Computing Conference*, pages 676–681, 1990.

[RS95] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. Technical Report PI-968, IRISA, 1995.

[SC93] I. Song and Y. Cho. Page prefetching based on fault history. In *Mach III Symposium*, pages 203–213, April 1993.

[SD87] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *14th Annual International Symposium on Computer Architecture*, pages 234–243, 1987.

[Sea95] Steve Sears. Personnal communication. OSF Organization, May 1995.

[SS94] S. Saini and H. D. Simon. Applications performance under osf/1 ad and sunmos on the intel paragon xp/s-15. Technical Report 1063-9535, NASA Ames Research Center, 1994.

[Sun93] Sun Microsystems Inc. *The SPARC Architecture Manual*, 1993.

[T+87] A. Tevanian et al. A unix interface for shared memory and memory mapped files under mach. Technical report, Carnegie Mellon University, 1987.

[Tal93] E. G. Talbi. *Allocation de Processus sur les architectures Parallèles à mémoire distribuée.* PhD thesis, Institut National Polytechnique de Grenoble, Mai 1993.

[Tan87] A. Tanenbaunn. *Les Systemes D'Exploitation.* Inter Editions - Paris, 1987.

[Tan92] A. Tanenbaunn. *Modern Operating Systems.* Prentice-Hall International Editions, 1992.

[Tri95] Stefan Tritscher. Personnal communication. European Supercomputer Development Center - Intel Corporation, May 1995.

[TSF90] M. Tam, J. Smith, and D. J. Farber. A taxonomy-based comparison of several distributed shared memory systems. *Operating Systems Review*, 14(3):40–66, July 1990.

[US92] R. Unrau and M. Stumm. Hierachical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, University of Toronto, 1992.

[W+93] A. W. Wlison et al. Update propagation in the galacticanet distributed shared memory architecture. Technical Report CHPC TR 93-007, Worchester University, 1993.

[Wai91] P. Waille. *La Communication dans les multiprocesseurs à connectique programmable: réconfiguration et routage.* PhD thesis, Institut National Polytechnique de Grenoble, 1991.

[ZB92] R. N. Zucker and J. L. Baer. A performance study of memory consistency models. In *19th Annual International Symposium on Computer Architecture*, pages 2–12, 1992.

[ZSLW92] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, pages 540–554, 1992.