N° d'ordre : 54

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir

le Titre de Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de la formation doctorale d'Informatique de Lyon

par **Laurent LEFEVRE**

Conception et Mise en Œuvre d'un Environnement de Programmation Parallèle fondé sur un Système de Mémoire Distribuée Virtuellement Partagée

Le système DOSMOS

Soutenue le 8 Janvier 1997

Après avis de : PRIOL Thierry
ZWAENEPOEL Willy

Devant la commission d'examen formée de :

BRUNIE Lionel
COSNARD Michel
LUDWIG Thomas
PRIOL Thierry
ROBERT Yves
THÉRON Eric

Remerciements

Je tiens à exprimer ici ma reconnaissance envers les membres du jury.

- A Thierry Priol et Willy Zwaenepoel qui ont accepté d'être rapporteurs de cette thèse. Je les remercie pour leur grande patience à la relecture du manuscrit ainsi que pour les remarques qu'ils ont formulées.
- Merci à Thomas Ludwig qui a accepté, sans hésitation, de faire le voyage depuis Munich pour faire partie du jury.
- Je remercie aussi Eric Théron de la société Matra Cap Systèmes d'être membre de ce jury.
- Que Michel Cosnard, président du jury soit ici remercié.
- A Yves Robert, qui depuis son exil américain, m'a néanmoins soutenu et conseillé.
- A Lionel Brunie, un sacré directeur de thèse, pour son aide dans toutes mes démarches, sa bonne humeur et sa transparence. Je me souviens parfaitement de notre première rencontre en Juin 1993. Lui, jeune docteur en informatique médicale et moi qui venait de finir mon stage de DEA sur de l'imagerie. C'est là que l'on s'est dit qu'on était parfait pour faire du système !!!
- Je remercie aussi toutes les personnes qui ont contribué de près ou de loin à cette thèse : Olivier Reymann, Harald Kosch, Jérôme Bolliet, Sébastien Tixier et Stéphane Vernat.

Remerciements personnels

- A la mémoire de mon grand-père Juan, qui aurait peut-être bien dit un ou deux *"Me cago en dios !"* en voyant le déroulement d'une thèse mais qui aurait été fier de son petit-fils.
- A mes parents qui croient encore qu'une thèse se résume à des cours à la fac, des conférences aux quatre coins du monde et à surfer sur internet, mais en est-ce vraiment loin??
- A ma soeur Magali, co-turne d'occasion, à qui j'ai apporté au moins une certitude dans la vie; elle ne fera jamais d'informatique.
- Enfin à mes pôtes de thèse qui m'ont suivi aux quatre coins du monde et qui m'ont soutenu dans mes recherches par leurs candides questions ("Au fait tu bosses sur quoi?") : Didier, Jean-Marc, Cyril, Denis, Pierre, Richard, Thierry, Arnaud...

Je ne remercie pas l'armée Française, pour sa formidable perspicacité à utiliser les appelés au mieux de leur formation et qui m'a affecté à un bureau avec comme mission de trier du courrier pendant 10 mois.

"La Théorie, c'est quand on comprend tout
et que rien ne fonctionne.

La Pratique, c'est quand tout marche
et que l'on ne sait pas pourquoi.

Ici, on a réuni les deux.
Tout fonctionne et on sait pourquoi."

Extrait dérivé d'un proverbe militaire.
Base Aérienne d'Ambérieu en Bugey
Juillet 1996

Table des matières

Table des figures

CHAPITRE 1

Introduction - Motivations

Affirmer que le parallélisme permet d'atteindre de hautes performances, n'est plus suffisant aujourd'hui pour plaider son développement. Bien entendu, les utilisateurs restent toujours attirés par la puissance et l'extensibilité offertes par les nouvelles architectures. Mais la difficulté de programmation des machines parallèles, frein, sans doute, le plus important de leur diffusion, demeure très importante. En effet, si les performances des applications parallèles sont souvent représentées par la classique courbe d'accélération qui met en avant les performances (*speedup*) de l'application en fonction du nombre de processus, il existe, malheureusement, une troisième dimension, souvent cachée, que l'on peut appeler l'effort de programmation.

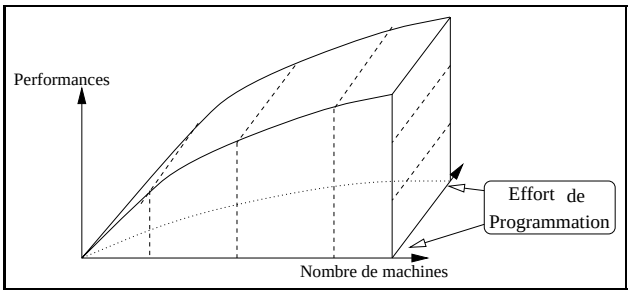


FIG. 1.1 - *Une application performante, mais à quel prix?*

Cet effort de programmation intrinsèquement important résulte de la conjonction de nombreux facteurs. Tout d'abord, il ne faut pas le cacher, la parallélisation d'une application est souvent difficile. Elle fait intervenir de nouvelles approches algorithmiques des problème très différentes de l'algorithmie séquentielle. Le problème est renforcé par la difficulté d'adéquation des modèles de programmation parallèle aux besoins des applications et des utilisateurs et par le manque de standardisation des environnements de programmation parallèle. Cette difficulté de programmation a limité fortement le développement du parallélisme en particulier en informatique industrielle.

1.1 Modèles de programmation parallèle

Pour paralléliser une application, un utilisateur dispose de trois grands modèles de programmation : l'échange de messages, la parallélisation par les données et la mémoire partagée. Analysons succinctement ces trois modèles.

1.2 L' échange de messages

Le développement des architectures massivement parallèles MIMD[1] et la standardisation et la généralisation de bibliothèques de communications ont favorisé une large expansion de la programmation parallèle à base d'échanges de messages. Souvent considérée comme la seule solution pour obtenir une parallélisation performante des applications, la programmation parallèle à échange de messages demeure difficile et réservée à une classe de programmeurs expérimentés. En effet, programmer en échanges de messages impose de gérer des structures de données complexes et de définir et d'optimiser l'ensemble des protocoles de communication inter-processus.

Depuis peu, sont apparues quelques bibliothèques standards en particulier PVM [Str90] et MPI [WY94]. Même si celles-ci proposent un modèle de programmation à base d'échanges de messages unifié et portable sur différentes plate-formes elles ne réduisent pas la difficulté intrinsèque à ce mode de programmation.

1.3 Le parallélisme de données

Il existe de nombreux langages à parallélisme de données dont certains ont été portés sur un grand nombre de machines. Du premier langage apparu, API [Ive62] à HPF [KLS+94], en passant par C* [Thi90], tous ont été conçus dans l'optique du parallélisme. De nombreuses extensions du langage FORTRAN ont, en particulier, été proposées par les constructeurs de machines parallèles, car Fortran est largement utilisé pour la conception d'applications numériques. Dans ce contexte, HPF [For93], High Performance Fortran, est une norme définie par un groupe de constructeurs et de chercheurs dans le but de fournir des compilateurs efficaces dédiés au parallélisme. L'utilisation d'HPF est fondée sur un certain nombre de directives de parallélisation. Ces directives permettent de compiler des programmes HPF aussi bien sur des machines SIMD que sur des machines MIMD. Elles concernent les fonctions d'alignement et de distribution des tableaux, la gestion de pointeurs, l'affectation et l'allocation dynamique de tableaux... Malheureusement, les compilateurs HPF, ne peuvent pas toujours garantir de bonnes performances [Dio96]. Une grande partie des manipulations complexes souvent indispensables à une parallélisation efficace reste, en effet, à la charge du programmeur (par le biais d'annotations astucieuses).

1.4 La Mémoire partagée

Même s'ils proposent un mode de programmation beaucoup plus intuitif et proche de la programmation séquentielle que les deux modèles précédents; les architectures et systèmes de mémoire distribuée virtuellement partagée souffrent de deux principales limitations. Les approches matérielles pâtissent de leur non-portabilité. Les systèmes logiciels souffrent de performances, parfois, décevantes et d'un manque d'environnements de développement efficaces. Ainsi seuls les systèmes LINDA et TreadMarks bénéficient d'une réelle diffusion.

1. M.I.M.D. : Multiple Instruction Multiple Data.

1.5 Vers un nouveau mode de programmation parallèle

Notre propos n'est pas de remettre en cause le parallélisme de données. En dépit d'une certaine difficulté d'utilisation, il s'adapte à merveille à des applications régulières. Nous ne remettons pas non plus en cause l'échange de messages ou la mémoire partagée qui, on l'a vu, présentent un certain nombre d'avantages et d'inconvénients. Mais notre démarche s'appuie sur la constatation qu'une large classe d'applications et d'utilisateurs ont été dédaignés par les concepteurs d'environnement de programmation.

Jusqu'à très récemment, l'accent, tant d'un point de vue architectural qu'algorithmique, a été essentiellement placé sur la performance des applications. Il s'agissait de définir des architectures puissantes et des algorithmes sur-optimisés en vue d'atteindre la meilleure performance possible même au prix de coûts de développement élevés. Cette approche est, sans doute, l'une des causes du moindre développement du parallélisme, eu égard aux espoirs que l'on plaçait dans ce nouveau mode de programmation. Les raisons de cet échec relatif tiennent dans l'inadéquation de cette approche "performance d'abord" aux réalités de la demande des utilisateurs. On peut distinguer, en effet, deux grandes catégories d'utilisateurs du parallélisme :

- L'utilisateur novice et l'industriel ont des demandes souvent proches en matière de programmation parallèle. Leur première contrainte n'est pas forcément l'atteinte de performances optimales. Ils recherchent plutôt une qualité de service qui leur permette de garantir un coût de programmation raisonnable ainsi qu'une bonne portabilité et maintenabilité de leurs logiciels (tout en conservant des performances correctes !). L'époque où la parallélisation d'un algorithme nécessitait de longs mois de modélisation et de développement est aujourd'hui révolue.

- Notre propos s'adresse aussi à *l'expert en parallélisation*. Celui-ci n'utilise souvent généralement qu'un seul modèle de programmation et qu'il s'efforce perpétuellement d'adapter aux spécificités applicatives auxquelles il doit faire face. Il est clair que l'apport d'un modèle de programmation plus général et plus puissant pourrait lui apporter une plus grande souplesse d'utilisation.

Ainsi pour ces deux types d'utilisateurs, la parallélisation d'une application "coûte cher" que ce soit en terme de développement ou de maintenance logicielle. ... Ils préféreraient, sans doute, implémenter un code, certes moins efficace, mais beaucoup plus rapidement développé (donc beaucoup moins cher) sur une machine-cible légèrement plus performante.

De la même manière, l'échec de certains constructeurs de machines parallèles généraliste tient pour une grande part à une analyse erronée du marché. Le segment des acheteurs potentiels de machines parallèles est, en effet, extrêmement réduit. Très peu d'organismes dans le monde ont le besoin et la capacité d'intégrer un groupe de développeurs d'applications parallèles. Pour les autres, le parallélisme reste un luxe hors de portée. Être "moins cher" suppose une minimisation des coûts d'investissements matériels mais aussi surtout des coûts de développement. Concernant le premier point, l'utilisation de composantes génériques et l'émergence de réseaux locaux haut-débit

(ATM, Myrinet...) ouvrent des potentialités de développement très prometteuses. Le deuxième point concerne l'objet de cette thèse.

Notre but est de tirer parti à la fois des avantages de la programmation à échange de messages et de la programmation fondée sur un environnement de programmation parallèle intégré qui essaie de répondre aux exigences suivantes :

- facilité et intuitivité de la programmation ;
- transparence du parallélisme ;
- portabilité et hétérogénéité: compatibilité avec une grande variété de plates-formes cibles, des réseaux de stations aux machines massivement parallèles ;
- compatibilité avec d'autres modèles de programmation ;
- efficacité et extensibilité des applications ;
- programmation parallèle structurée ;
- environnement d'aide à la programmation.

Les travaux décrits dans ce document s'appuient sur l'introduction de concepts et choix d'implémentation :

- mise en oeuvre d'un système de MDVP à base d'objets : en effet, ce type de systèmes offre un modèle de programmation à la fois intuitif (transparence des communications) et portable ;
- choix de PVM comme librairie de communication sous-jacente : PVM est, en effet, un standard de fait, porté sur la quasi-totalité des architectures parallèles et installé sur la plupart des réseaux de stations de travail. Il permet, en outre, de travailler sur des architectures hétérogènes;
- structuration des applications en groupes de processus : le système de Mémoire Distribuée Virtuellement Partagée (MDVP) DOSMOS permet (suggère !) de structurer les applications parallèles en tâches mises en oeuvre par des groupes de processus. Cette fonctionnalité, originale dans le cadre des MDVP, permet :
  - d'offrir de bonnes possibilités d'extensibilité aux applications, grâce à une minimisation des coûts de gestion de la mémoire virtuellement partagée et, ainsi, de proposer un modèle de programmation compatible avec des architectures massivement parallèles ;
  - de modéliser finement des applications structurellement modulaires : exemples : applications hétérogènes (faisant intervenir, par exemple, du traitement du signal et du calcul matriciel), applications fondées sur une redondance logicielle...
  - de s'adapter au plus près à la structure des architectures en grappes de processus (Machines SGI, Matra CAPITAN...) qui reçoivent aujourd'hui, un intérêt croissant (en particulier dans le cadre d'applications de bases de données parallèles) ;

  - d'envisager, à terme, le développement d'applications sur des réseaux à grande échelle (interconnexion à grande distance de réseaux locaux).

- compatibilité avec PVM : permettre le mélange de code DOSMOS et de code PVM est intéressant à plusieurs titres : optimisation de portions de code spécifiques, intégration de code de MDVP dans des applications PVM existantes dans le cadre de procédures de ré-engineering, utilisation de bibliothèques de calculs optimisées ;

- développement d'un environnement de programmation parallèle graphique : installation (des applications, du système), aide à la programmation parallèle, analyse d'exécution et optimisation des applications.

En résumé, notre objectif est de proposer un environnement de programmation parallèle fondé sur un modèle de programmation intuitif et capable de s'adapter à une gamme d'applications et d'architectures la plus large possible. Cela, on le verra, ne dispense cependant pas le programmeur de "penser parallèle" ! Paralléliser un code existant peut, certes, être réalisé avec DOSMOS de façon quasi-immédiate. Mais les risques de déception, en termes de performances, sont grands. L'efficacité d'une application passera, dans tous les cas, par une bonne analyse de la structure de l'application en tant qu'application *parallèle*. Dans ce cadre, l'environnement DOSMOS propose, d'une part, un modèle de programmation souple et intuitif, d'autre part, une variété d'outils d'aide à la programmation parallèle.

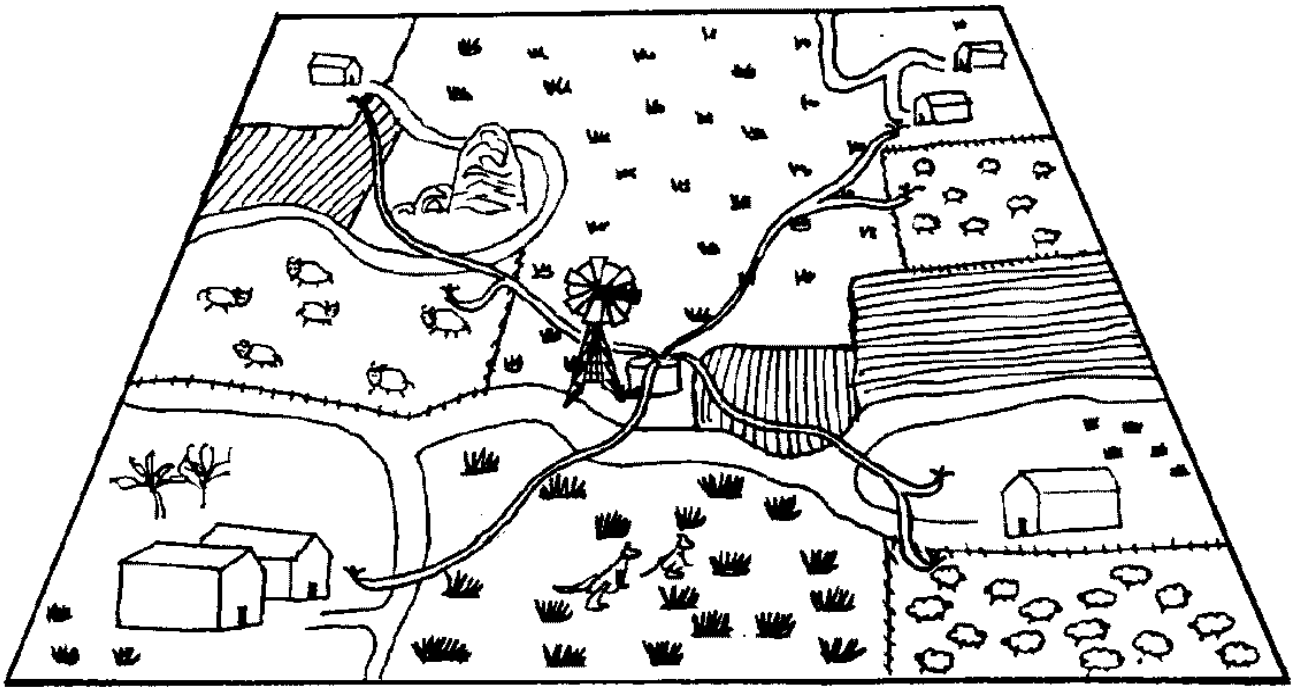
Cet environnement a été placé entre les "mains" d'un panel utilisateurs non-habitués aux systèmes de MDVP voire sans aucune expérience de la programmation parallèle. Le retour d'informations, très positif, a orienté la structure et les fonctionnalités des outils proposés.

Ce document est structuré en sept chapitres. Après un rapide exemple introductif rappelant que le concept d'accès à des ressources partagées n'est pas un concept nouveau, nous décrivons les concepts à la base des systèmes de Mémoire Distribuée Virtuellement Partagée (chapitre 3). Le quatrième chapitre propose un état de l'art de ce domaine. Puis, nous introduisons les concepts originaux à la base de notre modèle (chapitre 5) et décrivons le système DOSMOS qui s'appuie sur ce modèle (chapitre 6). L'environnement de programmation parallèle associé au système DOSMOS est décrit chapitre 7. Le chapitre 8 présente les expérimentations que nous avons réalisées. Suit une discussion sur l'ensemble de ces travaux et sur les perspectives qui en découlent. Le chapitre 10 conclue ce document.

CHAPITRE 2

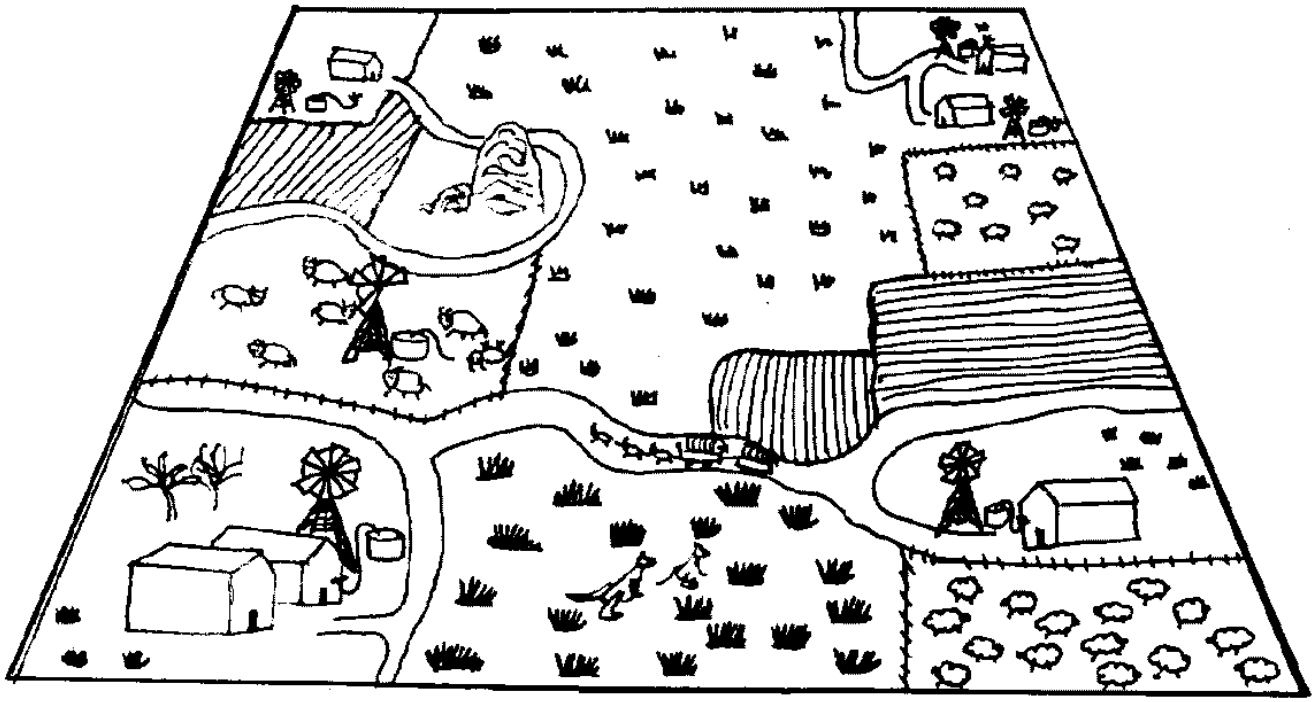
Du bush australien à la M.D.V.P.

"Pendant l'été 1870, les fermiers de la plaine d'Alice Springs dans le centre de l'Australie étaient désemparés.



Une sécheresse d'une rare intensité sévissait dans la région; elle décimait les troupeaux et détruisait les récoltes. Un seul point d'eau dans toute la plaine, alimenté par une énorme éolienne, était disponible pour le ravitaillement des plantations. Mais les canalisations étaient vétustes, le débit très faible et les fermiers craignaient qu'une défaillance de l'unique éolienne ne condamne toutes les propriétés de la région. La sécheresse persistant, on dut installer un roulement entre les fermiers, la quantité d'eau disponible ne permettant pas d'irriguer l'ensemble des plantations en même temps. De cette manière, tous les fermiers recevaient la même quantité d'eau. Mais cette eau était insuffisante; le bétail mourait et les plantations étaient perdues.

C'est pendant ce terrible été, que les bushmen de la région décidèrent de régler leur problème de ravitaillement en eau.

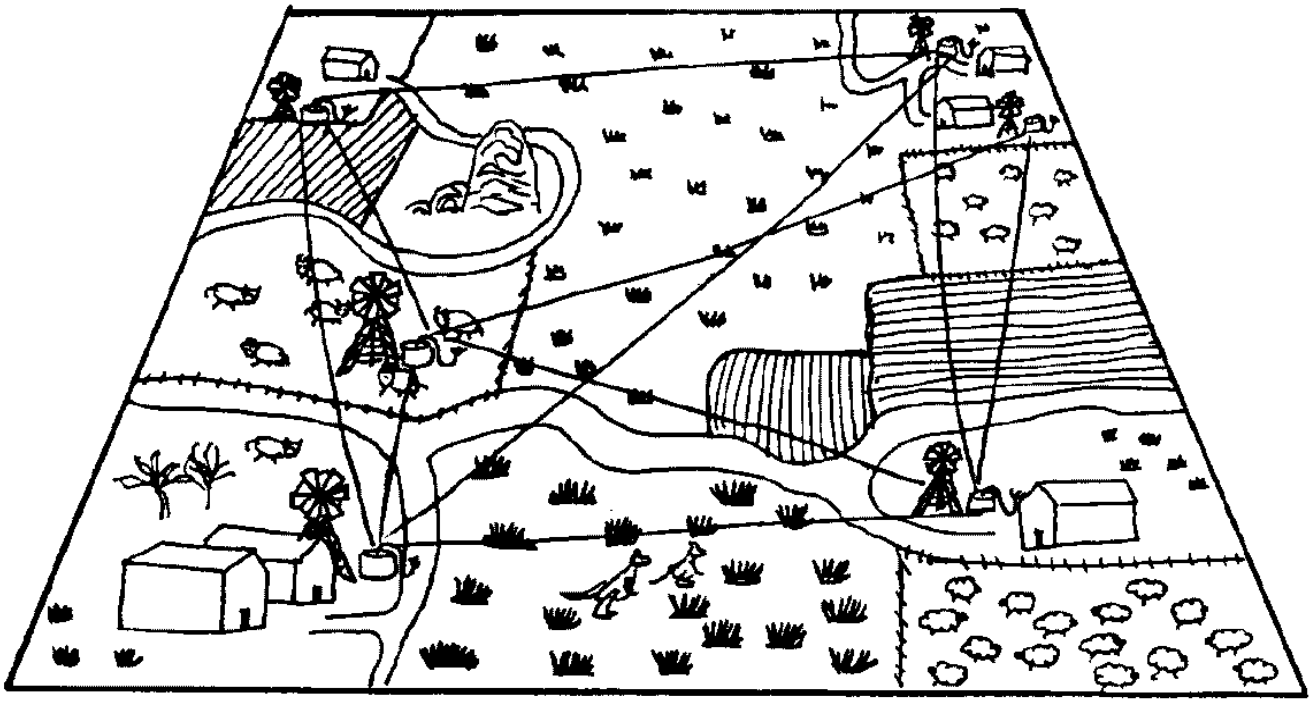


La vieille éolienne fût abandonnée et chaque domaine fermier effectua son propre forage phréatique alimenté par une éolienne de taille, certes plus modeste, mais située sur la propriété. On évitait ainsi un long acheminement de l'eau dans les gigantesques canalisations nécessaires à la traversée de ces immenses régions arides. Pendant les premiers temps, tout se passa bien, chaque bushman s'occupant de la maintenance de sa propre éolienne.

L'été suivant, une nouvelle vague de sécheresse s'abattit uniquement sur la partie sud d'Alice Springs. Ainsi, alors que des propriétés ne pouvaient subvenir en leur besoin en eau, d'autres, sous-utilisaient, au contraire, leurs propres installations devenues trop productives. Pendant un premier temps, les fermiers s'entraidèrent. Les propriétés qui disposaient de beaucoup d'eau organisèrent alors de gigantesques convois afin de transporter le précieux liquide jusqu'aux fermiers malheureux.

Cependant, cette organisation coûteuse se révéla insuffisante et de nombreuses propriétés du sud de la plaine perdirent du bétail et une partie de récoltes. On était revenu à la même situation que pendant l'été 1870...

C'est ainsi, qu'à la fin de l'été, les fermiers du bush décidèrent de mettre en commun leur expérience en matière d'irrigation. Ils voulaient utiliser leur propre source personnelle mais en même temps pouvoir disposer des sources voisines si le besoin s'en faisait sentir.



Les fermiers résolurent le problème en reliant toutes les sources locales entre elles par l'intermédiaire de simples canalisations. Ils prévoyèrent aussi des canalisations de secours, ainsi, si une éolienne tombait en panne, les autres sources pouvaient continuer à alimenter la propriété en difficulté.

Les propriétés étaient correctement alimentées en eau, les troupeaux et les récoltes étaient sauvés. L'avenir des propriétés fût ainsi garanti pour de nombreuses années."

Ainsi, sans le savoir, ces fermiers australiens ont mis en avant l'ensemble des problèmes que l'on peut rencontrer en parallélisme en ce qui concerne le mode d'accès au données. Ce sont les fondements des systèmes de Mémoire Distribuée Virtuellement Partagée.

Mais nous aurons l'occasion d'en reparler...

CHAPITRE 3

Systèmes de Mémoire Distribuée Virtuellement Partagée : Concepts de base

3.1 Pourquoi partager virtuellement une mémoire distribuée ?

Deux classes d'architectures s'opposent dans le monde des machines parallèles MIMD. La plus ancienne est celle des machines à mémoire partagée dans lesquelles tous les processeurs accèdent à une seule mémoire physique. Malheureusement, en dépit de quelques tentatives ([CG91]) la plupart de ces machines restent limitées en nombre de processeurs en raison de la centralisation des accès mémoire. Par contre leur programmation est facile et toutes les phases de communications sont transparentes pour l'utilisateur. A l'instar des fermiers australiens qui accédaient à leur unique éolienne, les processus manipulent qu'un seul espace mémoire. Chaque processus accède aux données à tour de rôle, d'où le risque de goulots d'étranglement.

A l'opposé, les systèmes massivement parallèles, dits "shared nothing", ne partagent rien: la mémoire et les disques rattachés à un processeur ne sont accessibles directement par aucune autre unité de traitement. On parle d'architectures à mémoire distribuée. Ces machines peuvent contenir un grand nombre de processeurs et permettre ainsi d'obtenir de bonnes performances. Leur handicap réside dans la difficulté de les utiliser efficacement : toutes les communications doivent être programmées (et optimisées !) par l'utilisateur. C'est la raison pour laquelle ce marché est resté longtemps limité au monde scientifique, gros consommateur de puissance de calcul et peu rebuté par les difficultés liées à la programmation parallèle [Com96]. On peut faire un certain rapprochement entre l'utilisation de ces architectures et la deuxième méthode proposée par les fermiers australiens dans laquelle ils disposent tous d'un forage local et irriguent ainsi facilement leur propre domaine. Par contre, dès qu'ils ont besoin d'accéder aux forages de leurs voisins, cela devient compliqué. Ils doivent prévenir le voisin qu'ils ont besoin d'eau, puis préparer un convoi avec des bidons à vide pour recueillir le liquide. Il leur faut alors attendre le retour du chariot avant d'irriguer leur domaine !

Dans ce contexte, il y a exactement dix ans, en 1986, Kai Li publie les premiers travaux concernant un nouveau système de programmation parallèle : une mémoire virtuellement partagée sur une architecture parallèle à mémoire distribuée [Li86].

C'est un compromis entre les deux architectures existantes, fonctionnellement, ces systèmes essaient de combiner la facilité de programmation des machines à mémoire partagée avec l'efficacité des machines à mémoire distribuée.

Un système de mémoire distribuée virtuellement partagée peut être vu de deux manières différentes.

- Niveau architectural : On s'appuie sur des architectures à mémoire distribuée en raison de leur potentiel en termes d'efficacité et d'extensibilité :
- Niveau logiciel : on donne l'impression à l'utilisateur qu'il travaille avec une (importante) mémoire partagée entre tous les processeurs. Il n'a ainsi plus à se préoccuper de la localisation de ses données ni des communications entre les processeurs.

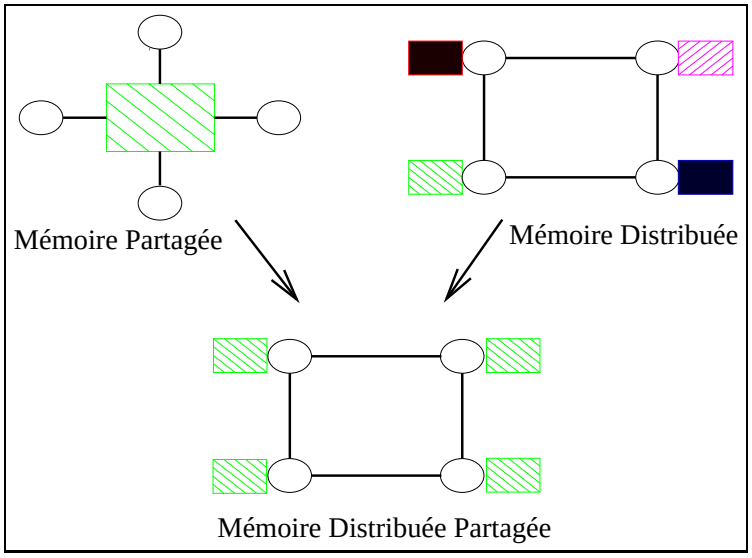


FIG. 3.1 - *Aboutir à une Mémoire Distribuée Virtuellement Partagée*

La figure 3.1 propose une image illustrant le concept de Mémoire Distribuée Virtuellement Partagée qui se veut une fusion des architectures à Mémoire Partagée avec les architectures à Mémoire Distribuée. Une Mémoire Distribuée Partagée propose donc une vue unifiée de différentes zones de mémoires distribuées entre les processeurs. Ainsi tels les fermiers mettant en commun leurs éoliennes par l'intermédiaire de simples canalisations, les processus accèdent à des zones mémoire distantes sans prendre en compte la localisation des données ni les moyens mis en oeuvre pour y accéder. Les processus "consomment" de la mémoire sans savoir si elle leur appartient réellement.

En dix ans, de nombreux systèmes ont été proposés qui, sans se départir des idées initiales proposées par Li, améliorent les modèles de Mémoire Distribuée Virtuellement Partagée. Deux grandes classes de systèmes à Mémoire Distribuée Virtuellement Partagée coexistent :

- **Systèmes à base de pages** : la mémoire de tous les processeurs est virtuellement réunie dans un seul espace d'adressage découpé en pages de taille fixée. Ce sont ces pages que le

système "partage" lors de l'exécution d'une application. Dans ces systèmes, la granularité est donc fixe (taille de la page) et ne prend pas en compte la taille des données traitées.

- **Systèmes à base d'objets** : Dans ce modèle, les données partagées sont des objets contenant des valeurs et éventuellement des fonctions d'accès (à l'instar de la programmation orientée-objet). La granularité des données partagées n'est pas fixée : elle dépend de la taille de chaque objet. L'utilisateur peut ainsi adapter ses données à ses applications en choisissant une taille appropriée.

Les paragraphes suivantes essaient de donner une vue aussi exhaustive que possible des travaux menés au cours des dix dernières années. Dans un premier temps, nous discuterons des concepts de base nécessaires à la mise en oeuvre d'une Mémoire Distribuée Virtuellement Partagée en termes de modèle, cohérence, protocoles, implémentation. Puis nous nous intéresserons aux principales réalisations qui ont été proposées en essayant de mettre en relief leurs spécificités, leurs apports et leurs limitations.

3.2 Mémoire virtuelle, mémoire partagée, mémoire répartie...

Il est, tout d'abord, sans doute nécessaire de bien fixer la terminologie employée par la suite. En effet, en une décennie, les systèmes ayant beaucoup évolué, le lecteur d'articles traitant de Mémoire Distribuée Virtuellement Partagée peut souvent être dérouté par la multitude des termes employés. Nous proposons ici un consensus sur les différentes définitions :

Définition 3.1 *Mémoire Virtuelle Partagée (MVP ou Shared Virtual Memory) : ce terme utilisé lors de la mise en œuvre des premiers systèmes désigne les systèmes qui partagent des pages de données.*

Définition 3.2 *Mémoire Distribuée Partagée (MDP ou Distributed Shared Memory) : utilisé à l'émergence des premiers systèmes à base d'objets afin de les différencier des systèmes à base de pages.*

Définition 3.3 *Mémoire Partagée Aléatoire (MPA ou Randomized Shared Memory) : proposée par [Hel92, EJ93], elle peut être considérée comme une amélioration des MVP. La répartition de l'espace d'adressage unique se fait ici en fonction de lois probabilistes proposant un placement optimisé sur les processeurs.*

Définition 3.4 *Mémoire Partagée Répartie (MPR ou Network Shared Memory [OMW+92]) : ce terme est plutôt dédié aux systèmes implémentés sur des architectures distribuées ou réparties afin de les différencier des systèmes exclusivement parallèles.*

Définition 3.5 *Mémoire Distribuée Virtuellement Partagée (***MDVP***) : ce terme générique, utilisé depuis peu, permet de désigner des systèmes partageant logiquement de la mémoire sans faire de distinction entre les architectures d'implémentation ni les caractéristiques logicielles des systèmes.*

Par la suite, nous emploierons essentiellement le terme de Mémoire Distribuée Virtuellement Partagée (MDVP) qui nous semble le mieux convenir pour décrire et comparer les systèmes quels que soient leur implémentation ou les types de données qu'ils partagent.

3.3 De la duplication pour de meilleures performances

Au milieu des années 80, les concepteurs de systèmes parallèles améliorèrent les machines à mémoire partagée en associant à chaque processeur une zone mémoire permettant de stocker temporairement des données : un cache local.

Les premières techniques de gestion de caches consistaient, très simplement, à interdire à une donnée d'être présente dans plusieurs caches à la fois. Le principe d'unicité de la donnée en mémoire était préservé. Ainsi les données migraient à la demande entre les différents processeurs. De nombreux systèmes ont été implémentés en utilisant cette technique [ABLN85, JLHB88, CAL+89]. Ces systèmes évitaient ainsi les problèmes de cohérence des données mais les gains en performances apportés étaient faibles. Les applications parallèles ne pouvaient alors tirer pleinement partie des machines à mémoire partagée.

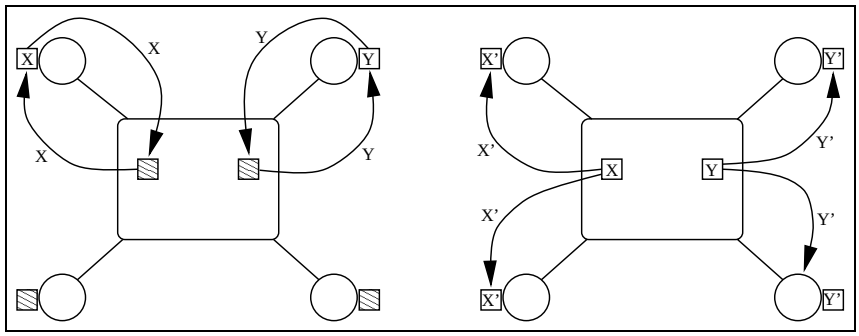


FIG. 3.2 - *Migration des données (à gauche) : chaque processeur qui accède à X (ou Y) verrouille cette donnée et la relâche à la fin de ses manipulations. Duplication des données dans les caches locaux (à droite) : les processeurs récupèrent une copie de la donnée dans leur cache, ce qui permet des accès en parallèle*

Pour atteindre de meilleures performances, les caches locaux doivent pouvoir servir d'espace mémoire dédié à la recopie des données. Ainsi, à un instant donné, les données peuvent être dupliquées dans différents caches. Les accès aux données sont ainsi améliorés, les processeurs profitant de la présence des copies en mémoire locale. Par contre, des problèmes de cohérence peuvent apparaître lorsque les copies ne sont plus valides ou que plusieurs processeurs modifient une copie de la donnée en même temps. Le système doit alors assurer la cohérence des données partagées au moindre coût (Fig.3.2) Cela a abouti au développement des modèles de cohérence de cache, puis, par la suite, aux modèles de cohérence des données partagées.

3.4 De la cohérence de cache à la cohérence de données partagées

De nombreuses techniques ont été développées en vue d'assurer une bonne gestion des caches mémoire [Str90, Lil93]. Ce sont ces techniques qui ont été adaptées par la suite pour la cohérence des données partagées.

La cohérence d'une mémoire partagée fait référence à la sémantique des accès concurrents aux objets partagés. Les deux types d'accès disponibles sont la lecture et l'écriture d'une donnée. Les principales questions auxquelles doivent répondre les sémantiques de cohérence sont ainsi : Quelle est la valeur retournée par la lecture d'une donnée partagée? A-t-on la possibilité d'effectuer des écritures concurrentes? Comment les traiter?...

Définition 3.6 *Une mémoire partagée est cohérente si tous les accès aux données partagées vérifient la sémantique spécifiée.*

Pour une MDVP, le choix de la sémantique de cohérence est crucial puisqu'il va déterminer *in fine* le type des applications que le système sera capable de modéliser. Ainsi, il n'est pas surprenant de constater qu'un nombre considérable de critères de cohérence ont été proposés et étudiés.

On sépare traditionnellement les modèles de cohérence en deux grandes familles : les cohérences fortes et les cohérences faibles ou relâchées. Les critères de cohérence forte se réfèrent à des exécutions réparties fondées sur une relation temporelle globale entre les différents processus. A l'inverse, les modèles de cohérence faible considèrent la sémantique des applications telles qu'elles sont décrites par le programmeur. En d'autres termes, une cohérence forte ordonnance l'exécution de toutes les opérations de lecture/écriture alors qu'une cohérence relâchée s'intéresse à certaines opérations critiques. Deux définitions font ainsi référence dans la littérature [BCZ90]:

Définition 3.7 *Une mémoire partagée est fortement cohérente si la valeur retournée par une opération de lecture est la même que la valeur écrite par la plus récente opération d'écriture à la même adresse.*

Définition 3.8 *Une mémoire partagée est faiblement cohérente si la valeur retournée par une opération de lecture est celle écrite par une opération d'écriture au même emplacement. Cette écriture peut avoir immédiatement précédé l'opération de lecture dans un ordonnancement légal de l'exécution des processus.*

Ces deux définitions reposent sur une notion de temps global qui permet d'ordonnancer les accès (ex: le temps causal de Lamport [Lam78]).

Définition 3.9 *Soient A et B deux accès à une même donnée partagée (lecture ou écriture). On dit que A précède B (et on note $A -> B$) si et seulement si l'exécution de A est complètement terminée avant que B ne commence.*

Dans les paragraphes suivantes, nous allons présenter rapidement les principaux modèles de cohérence disponibles pour l'implémentation d'une MDVP, de la cohérence la plus forte à la plus relâchée.

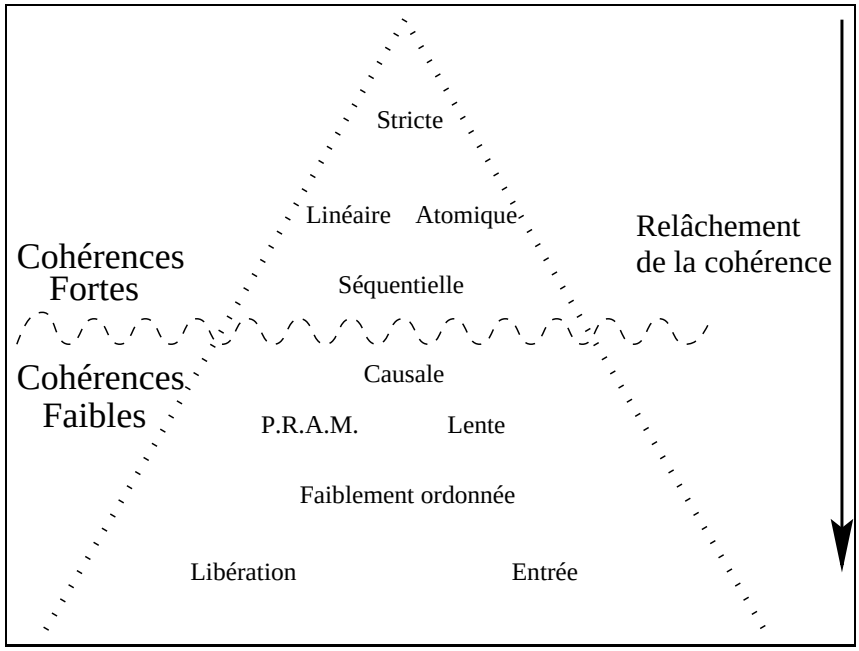


FIG. 3.3 - *Ensemble des modèles de cohérence: de la plus restrictive (forte) à la plus relâchée (faible).*

3.4.1 Les cohérences fortes

– Cohérence stricte

L'approche la plus naturelle est, sans doute, de considérer qu'une mémoire est cohérente si et seulement si la valeur retournée par une opération de lecture est la même que la valeur écrite par la plus récente opération d'écriture à la même adresse. La cohérence stricte repose sur le fait qu'une même donnée ne peut exister qu'en une occurrence dans tout le système d'information. Ainsi, toutes les opérations sont totalement ordonnées. Un tel modèle implique une perte importante de performance puisqu'il empêche le recouvrement des accès mémoire.

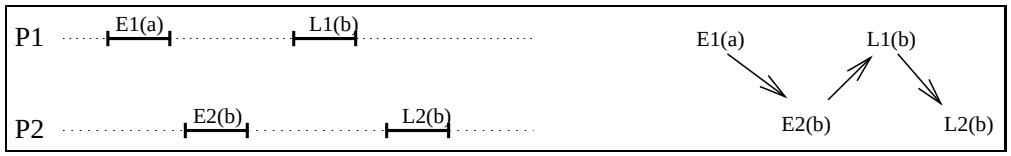


FIG. 3.4 - *Cohérence stricte: Partage d'une même donnée par deux processus.*

Notation. L'exécution temps-réel des processus en ce qui concerne les opérations de Lecture et Écriture sur la même donnée partagée est visible sur le côté gauche de la figure 3.4. La notation *Ex(y)* représente l'écriture de la valeur y dans la donnée partagée. Le paramètre x

sert à numéroter les opérations de lecture/écriture, il désigne le numéro du processus sur lequel a eu lieu l'accès. D'une manière similaire *Lx(y)* représente une opération de lecture générée par le processus x. Cette lecture de la variable partagée renvoie la valeur y. Les notations Pi représentent les processus sur lesquels ont lieu les accès. Le côté droit de la figure propose l'ordonnancement des accès de telle manière qu'il respecte la sémantique de cohérence stricte

Comme on peut le voir dans la figure 3.4 les données partagées ne sont pas dupliquées ce qui induit que la mémoire est (forcément) toujours cohérente. Même si on ne s'intéresse pas aux communications entre les processus, on peut remarquer qu'entre les accès $E2(b)$ et $L1(b)$, il y aura obligatoirement communication de la donnée partagée entre les deux processus. Cependant, en séquentialisant tous les accès, ce modèle crée un goulot d'étranglement qui fait chuter les performances. C'est la raison pour laquelle la plupart des MDVP proposent une duplication des données partagées qui permet, au minimum, d'effectuer des lectures en parallèle de la même donnée partagée.

Exemple. Application compatible avec une cohérence stricte : calcul du montant du solde d'un compte en banque. L'unicité des données garantit ainsi une vue cohérente à tout moment du montant total. Cette sémantique de cohérence est aussi utilisée dans les systèmes transactionnels (systèmes de gestion de bases de données par exemple) fondés sur des opérateurs de verrouillage en lecture et verrouillage (exclusif) en écriture.

– **Cohérence linéarisable ou atomique**

La cohérence linéarisable (équivalente à la cohérence atomique) ressemble à la cohérence stricte mais permet la création de plusieurs copies d'une même donnée partagée.

Cette sémantique de cohérence utilise un ordre partiel noté PO sur les opérations de lectures et d'écritures (voir définition 3.9).

Définition 3.10 *Une exécution parallèle est linéarisable si et seulement si un ordre total TO (appelé historique de processus) peut être déduit d'un ordre de précédence partiel PO et si l'exécution séquentielle associée à TO a le même comportement (même résultat) qu'une exécution en temps réel.*

Les concepts à la base de la cohérence linéarisable sont donc les suivants :

- duplication des données ;
- lectures et écritures parallèles autorisées ;
- si une opération d'écriture A précède une écriture B, alors les lectures suivantes doivent retourner la valeur écrite par B. Si A et B sont concurrentes, les opérations de lecture rendent toutes la valeur écrite par A ou celle écrite par B.

Définition 3.11 *Un protocole de cohérence est linéarisable si et seulement si l'exécution de toute application utilisant ce protocole est linéarisable.*

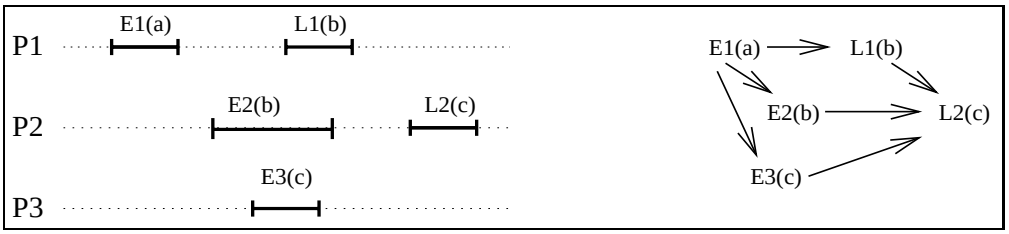


FIG. 3.5 - *Cohérence linéarisable : trois processus accèdent à la même donnée partagée*

Dans la figure 3.5, trois processus P1, P2 et P3 modifient et lisent la même donnée partagée. Cette exécution est linéarisable car l'historique de processus : $E1(a)E2(b)L1(b)E3(c)L2(c)$ est compatible avec l'ordre partiel sur les opérations de lecture/écriture.

Exemple. Application compatible avec la cohérence linéarisable : contrôle de température : n processeurs connectés à des capteurs thermaux modifient en parallèle la valeur de la température en sortie. Pendant ce temps-là, p processus lisent concurremment la valeur de la température et réalisent différentes opérations en fonction de la valeur lue.

– **Cohérence séquentielle**

Cette forme de cohérence moins restrictive a été introduite par Lamport [Lam79]. Elle a été largement utilisée par la suite (exemples : [AH90, ZMS93]...).

Définition 3.12 *Une mémoire partagée est séquentiellement cohérente si et seulement si le résultat de toute exécution d'application est linéarisable, la relation de précédence étant restreinte aux accès s'exécutant sur le même processeur.*

Considérer un temps global n'a, de fait, pas réellement de sens dans les systèmes parallèles ou distribués. En effet, on ne peut contrôler les vitesses des processeurs et les horloges sont difficilement synchronisables. En pratique, le seul critère fiable et déterministe est l'ordre séquentiel des opérations de lecture et d'écriture sur un processeur.

Les concepts à la base de la cohérence linéarisable sont donc les suivants :

- écritures et lectures concurrentes autorisées ;
- cohérence linéarisable mais avec un ordre partiel limité aux processus s'exécutant sur un même processeur : une opération d'accès A précède B si et seulement si A est terminée avant que B ne commence et si A et B sont exécutées sur le même processeur.

La cohérence séquentielle s'apparente donc fortement à une sémantique proche de celle fournie par une architecture à base de mémoire partagée avec caches locaux et bus de communication.

Ainsi, dans la figure 3.6, l'historique $E1(a)L1(a)E2(c)E3(b)L2(b)L3(b)$ est compatible avec l'exécution réelle. On peut remarquer que cette exécution est séquentiellement cohérente mais non linéarisable.

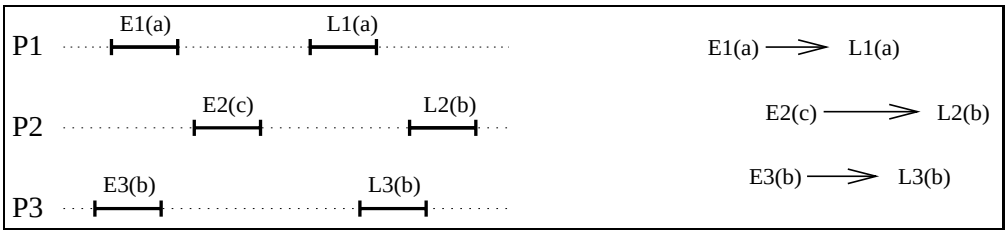


FIG. 3.6 - *La cohérence séquentielle : partage d'une même donnée par trois processus*

Exemple. Application compatible avec la cohérence séquentielle : exemple précédent du contrôle de température. En cohérence linéarisable, si une mesure de température précède une autre, le système assure que cette mesure n'est plus valide pour tous les processus. La cohérence séquentielle assure simplement que les accès ordonnés sont compatibles avec l'exécution séquentielle sur chaque processeur. Un processus pourrait donc encore lire une mesure invalide, mais si ce processus écrit une nouvelle mesure alors les accès en lecture suivants seront valides.

Le dénominateur commun des sémantiques de cohérences fortes présentées ici réside dans le fait qu'elles assure que tous les processeurs sont d'accord sur un ordre global de toutes les opérations d'écriture. Ainsi, si un processeur *perçoit* les écritures dans un certain ordre, alors aucun autre processeur ne peut voir ces écritures dans un ordre différent. Cependant, la construction d'un ordre total partagé par tous les processeurs nécessite de lourdes opérations de synchronisation, limite les possibilités de recouvrement et réduit le degré de parallélisme des applications.

3.4.2 Les cohérences faibles

Les critères de cohérence faible proposent une alternative aux cohérences fortes. En résumé, une cohérence faible permet des lectures et des écritures concurrentes mais ne fournit pas d'ordre global des accès. Ainsi les opérations d'écriture peuvent être perçues différemment par les processus. La sémantique des accès en lecture ou en écriture est déterminée par l'application elle-même. L'utilisation correcte de ces cohérences incombe donc à la charge du programmeur puisqu'il doit intégrer des points de synchronisation dans ses applications. Ainsi la plupart des cohérences faibles proposées sont définies plutôt par des protocoles qu'à l'aide de descriptions formelles.

- **Cohérence causale**

  Initialement dédiée aux MDVP à objets, la cohérence causale a d'abord été introduite par [ABHN91, ABHN92] puis détaillée par la suite dans [RMN92, RM93]. La cohérence causale est basée sur le principe de causalité potentielle de Lamport [Lam78]. Elle se situe à la frontière entre les cohérences fortes et relâchées.

  Définition 3.13 *Un protocole est causalement cohérent si et seulement si sur chaque processus, il existe un ordre total reliant l'ensemble des opérations d'écriture et de lecture compatible avec la relation de précédence causale.*

Les idées de base sont les suivantes :

- tous les processus perçoivent toutes les opérations d'écriture (ce qui ne sera plus vrai, pour d'autres cohérences faibles) ;
- les écritures sur un objet partagé peuvent être concurrentes et perçues différemment selon les processus.

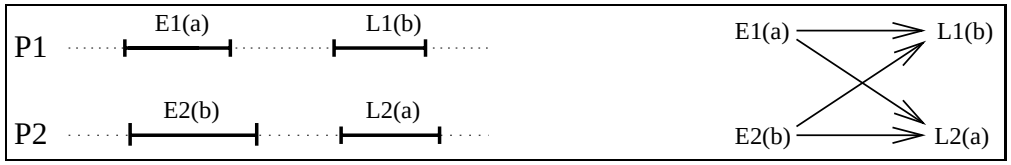


FIG. 3.7 - *Cohérence causale : partage d'une même donnée par deux processeurs*

En d'autres termes, chaque processeur a une perception personnelle des opérations d'écritures et construit ainsi un ordre qui lui est propre. Mais la causalité des accès est conservée et des accès en écriture effectués séquentiellement par un même processeur sont perçus dans le même ordre. Dans la figure 3.7, nous pouvons observer les deux historiques des accès réalisés par l'application tels qu'ils sont vus par les deux processeurs. Sur le processeur P1 : $E1(a)L2(a)E2(b)L1(b)$ alors que sur P2 : $E2(b)L1(b)E1(a)L2(a)$.

Exemple. Application utilisant la cohérence causale : consultation d'informations (titre, auteur, résumé) sur les ouvrages d'une bibliothèque. Dans ce cas-là, ce sont surtout les lectures des données qui sont nombreuses. Il y a peu de modifications des informations. Si ces modifications des données sont perçues de manière différente, cela importe peu.

- **Cohérence Pipelined Random Access Memory**

Ce type de cohérence faible a été introduit par [LS88] et utilisé par le système MERMERA [HS92a]. La cohérence Pipelined Random Access Memory (P.R.A.M.) est basée sur l'utilisation massive de protocoles d'écriture-diffusion totale pour la mise à jour des données partagées (paragraphe 3.7.2). Chaque processus dispose d'une copie des données partagées. Ainsi, les lectures se font toujours sur la copie locale. Les écritures nécessitent une modification de la copie locale puis une diffusion de la nouvelle valeur à tous les processus.

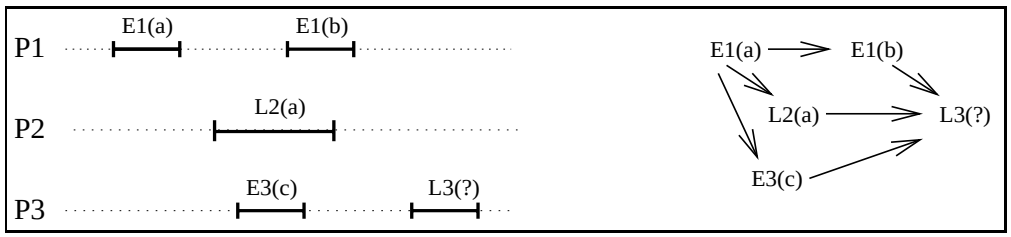


FIG. 3.8 - *Trois processus partagent une même donnée à l'aide d'une cohérence PRAM*

On peut se rendre compte intuitivement qu'une cohérence PRAM est causale seulement entre des paires de processus. Mais l'utilisation d'une telle cohérence est encore trop coûteuse en termes de communications entre les processus.

Exemple. Application compatible avec la cohérence PRAM : service des cartes bancaires dans une banque. Toute déclaration de vol d'une carte bancaire entraîne automatiquement une opposition dans la banque du client qui la propage aussitôt aux serveurs des autres organismes bancaires.

D'autres sémantiques de cohérence telles que la mémoire lente [HA90], la mémoire localement cohérente [HS92a] ou la mémoire faible [Hel90] ont été proposées dans la littérature. Elles reposent sur de légères différences avec les cohérence P.R.A.M. et causale et elles difficilement employables dans le cadre d'applications réelles.

– **Cohérence faiblement ordonnée**

Proposée par [DSB88] et utilisée par [BH90], la cohérence faiblement ordonnée s'inspire de la cohérence séquentielle. Dans ce modèle, seuls les accès mémoire à des variables de synchronisation définies par le programmeur ont la garantie d'être exécutés dans un ordre séquentiellement cohérent (points de synchronisation).

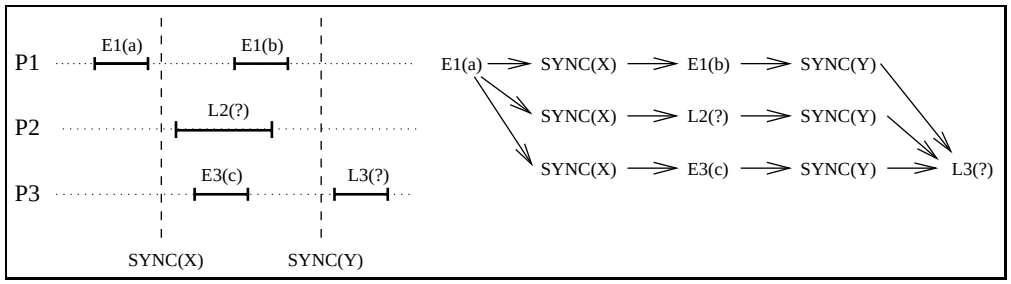


FIG. 3.9 - *Cohérence faiblement ordonnée entre trois processus*

Dans la figure 3.9, les accès aux variables de synchronisation sont notés $(SYNC(X))$. Ainsi, les accès à X sont séquentiellements cohérents. Les accès en lecture $Ln(?)$ signifient qu'on ne peut garantir la valeur qui sera retournée au processus n lors de la lecture de la donnée partagée. Pour éviter d'exécuter des applications non-déterministes, chaque processus doit garantir que tous les accès aux données sont terminés avant l'exécution d'un point de synchronisation.

– **La cohérence à la libération**

Ce modèle, proposé par [GLL+89], relâche les contraintes d'ordonnancement sur les variables de synchronisation en introduisant deux nouveaux opérateurs de synchronisation : *acquire* et *release*. Ces deux opérateurs sont appelés par un processus pour accéder à des données partagées; *Acquire* permet d'obtenir un accès à la donnée et *Release* libère l'exclusivité et met à jour les copies de la donnée partagée.

Dans la figure 3.10, *Acq2* et *Rel2*, signifient que le processus P2 a demandé un accès à la donnée. La cohérence à la libération permet aussi l'utilisation de primitives de synchronisation identiques à celles employées en cohérence faiblement ordonnée. Lors du relâchement (*Rel2* de l'objet), tous les processus reçoivent une copie valide de la donnée. En dehors des sections encadrées par les opérateurs *acquire* et *release*, les processus peuvent ne pas connaître la dernière valeur d'une donnée partagée. L'utilisateur est donc complètement responsable du

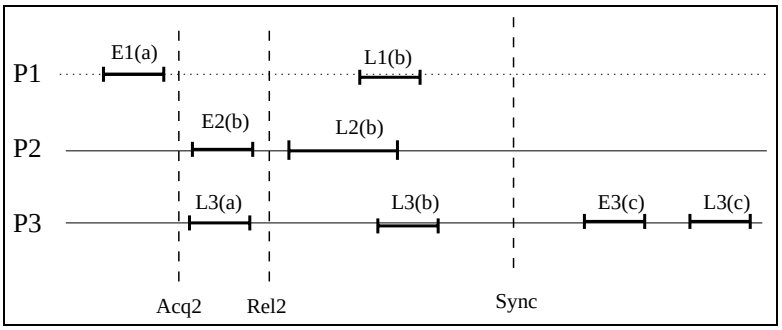


FIG. 3.10 - *Cohérence à la libération : partage d'une même donnée entre trois processus*

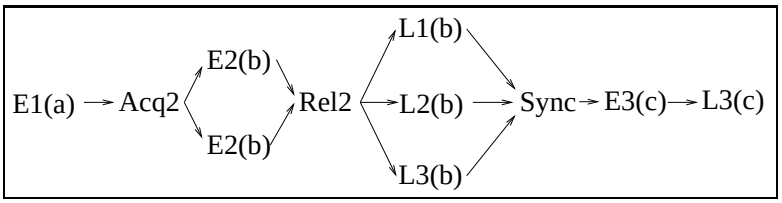


FIG. 3.11 - *Ordre total des opérations de lecture/écriture*

déclenchement de la mise à jour des copies des données partagées (réalisé, on l'a dit, par l'opérateur *Release*).

Exemple. Application utilisant la cohérence de à la libération : Calcul optimisé d'un maximum distribué. Chaque processus calcule la valeur maximum parmi ses propres données (maximum local), puis il met à jour le maximum global via une phase d'acquisition/relâchement de la donnée.

– **La cohérence en entrée**

Ce modèle de cohérence proposé par [BZ91] pour le système MIDWAY garantit une vue cohérente de la mémoire lorsqu'un le processeur entre dans une section critique. Ainsi, comme pour la cohérence à la libération, les accès aux données partagées peuvent être combinés avec des appels aux opérateurs de synchronisation (*Acquire* et *Release*). Contrairement à la cohérence à la libération qui charge l'opérateur *Release* de la mise à jour des copies, la cohérence en entrée impute cette fonction à l'opérateur *Acquire*.

– **Cohérence à la libération paresseuse** La cohérence à la libération paresseuse [KCZ92] propose un compromis entre cohérence en entrée et cohérence à la libération. Dans ces modèles, le relâchement (ou l'acquisition) d'une donnée partagée se traduit par l'envoi un message d'invalidation signalant la modification de la donnée. Cet envoi est adressé à tous les processus qui ont accédé à l'objet (voir paragraphe 5.4.4). Or, seul le prochain processus réalisant une acquisition de l'objet est réellement concerné par cette invalidation. Ainsi, en cohérence à la libération paresseuse seul le prochain processus ayant besoin de la donnée partagée est avisé de la modification de cette donnée. L'implémentation de ce modèle nécessite une gestion de files d'attente afin de mémoriser les futurs accès aux données.

Ainsi de nombreux modèles de cohérence ont été étudiés au cours des dernières années. La plupart de ces modèles résultent d'un vide théorique qui permet de passer d'un modèle à un autre. Mais leurs spécificités théoriques sont difficilement applicables.

En pratique seuls les modèles extrêmes sont utilisés. D'un côté, on trouve les sémantiques de cohérence qui déchargent totalement l'utilisateur de toute intervention (stricte, séquentielle...). Mais, on assiste aussi à l'implémentation de modèles de cohérence qui considèrent, au contraire, que l'utilisateur est totalement responsable du déclenchement des synchronisations. Ces modèles extrêmes correspondent à une réalité applicative. Il est ainsi difficile de concevoir la différence d'exécution d'applications qui utilisent des modèles proches. Très peu de modèles différents ont de facto été implémentés dans des systèmes à Mémoire Distribuée Virtuellement Partagée: cohérence stricte, séquentielle, en entrée, à la libération.

3.5 Duplication des données en écriture : écrivains multiples

Nous avons vu dans le paragraphe 3.3 que la duplication des données permet d'augmenter le degré de parallélisme des applications.

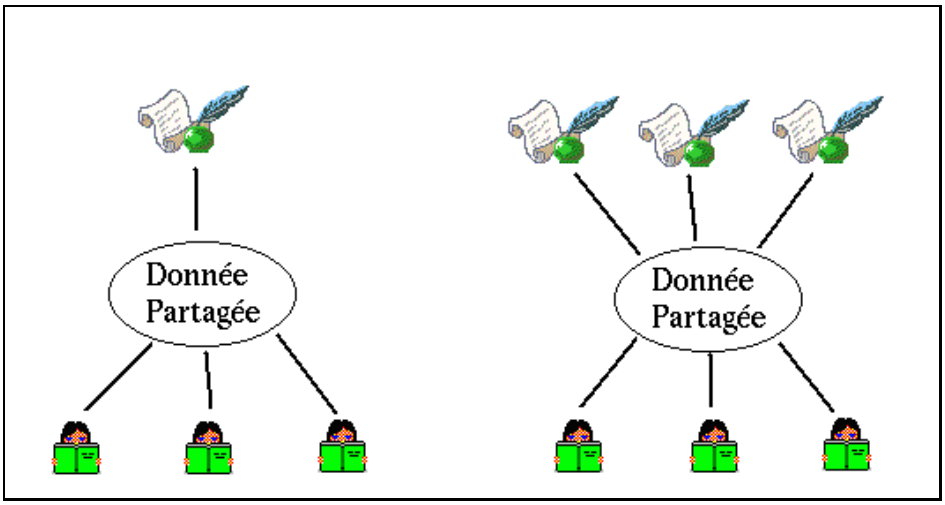


FIG. 3.12 - *Écrivain unique ou écrivains multiples?*

En pratique, toutes les MDVP existantes reposent sur deux grandes approches (figure 3.12) :

- ***Écrivain unique et lecteurs multiples*** : dans cette stratégie, un seul processus a la possibilité de modifier une donnée partagée. Toutes les autres copies de la donnée présentes parmi les processus sont disponibles en lecture seulement. Cette technique implémentée par de nombreux systèmes de MDVP permet d'alléger considérablement les protocoles de cohérence des données partagées ;
- ***Écrivains multiples et lecteurs multiples*** : on considère ici que plusieurs processus peuvent écrire dans une même donnée partagée en même temps. Pour ce faire, chaque processus écri-

vain dispose d'une copie de la donnée modifiable. Lorsque plusieurs écrivains ont ainsi la possibilité d'écrire sur une même donnée, on considère, alors, que cette donnée est protégée en écriture. Toute écriture sur cette donnée se traduit donc par la création d'un jumeau (figure 3.13) afin de permettre des modifications locales de la donnée.

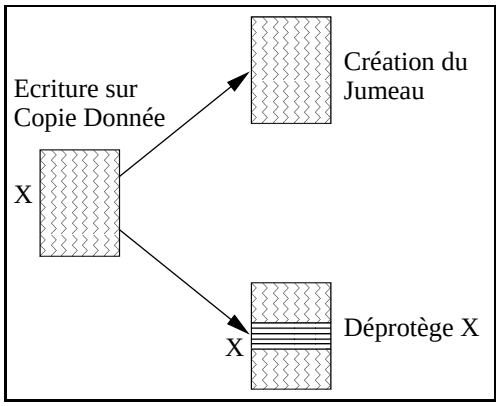


FIG. 3.13 - *Création d'un jumeau*

Le jumeau sert ainsi de référence pour une comparaison des copies de la donnée avec la mise à jour de la donnée à la fin des opérations des écritures. La terminaison des écritures dépend du modèle de cohérence choisi (synchronisation, relâchement de la donnée...). A cette étape, le système recherche les différences entre le jumeau et les copies modifiées et met à jour les copies de la donnée après une destruction des jumeaux (figure 3.14). La donnée est ensuite reprotégée en écriture jusqu'au prochaines modifications.

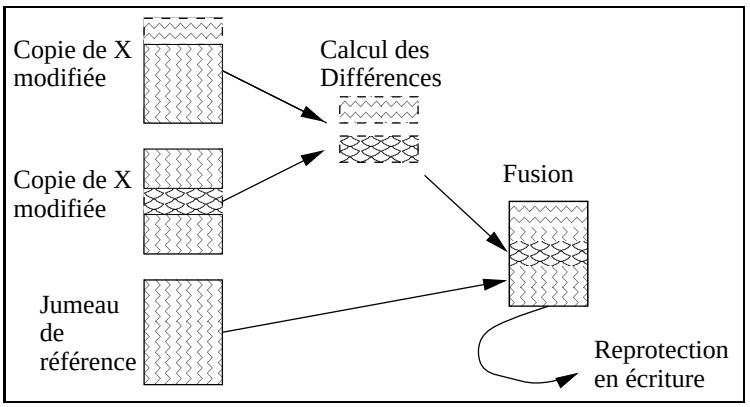


FIG. 3.14 - *Mise à jour des données avec utilisation d'écrivains multiples*

Cette technique est notamment utilisée pour permettre la modification de zones de données présentes à des emplacements différents au sein même de la donnée partagée (mise à jour des pages dans KOAN [BBLP91] ou TreadMarks [ACD+96]).

Si les zones de données modifiées (les différences) se recouvrent c'est le rôle du protocole de cohérence de déterminer la valeur valide de la donnée partagée. Si le protocole de cohérence ne satisfait pas à cette exigence, le système peut induire des comportements non-déterministes aux applications. Enfin, cette stratégie qui permet d'améliorer les performances des systèmes

nécessite l'utilisation de protocoles de cohérence adaptés tels que la cohérence faiblement ordonnée, en entrée ou à la libération qui implémentent des points de synchronisation.

3.6 Propriétaire des données

Lorsqu'un processus désire modifier ou lire une donnée partagée non-présente dans sa mémoire locale, le premier problème est de localiser le processus gestionnaire de cette donnée. On trouve dans la littérature trois grands types de gestion de propriété de donnée dérivés des méthodes proposées par Li et Hudak [LH89].

3.6.1 Gestion centralisée

Dans ce type de gestion, introduite dans les premières MDVP, toutes les données partagées sont gérées par un processeur dédié, le gestionnaire central. Ce gestionnaire est localisé sur un processeur connu de tous les processus de l'application. Il gère toutes les données partagées et est le seul à pouvoir en modifier le contenu.

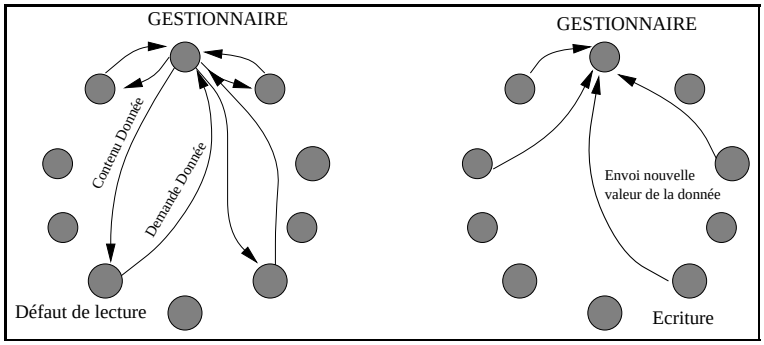


FIG. 3.15 - *Gestionnaire centralisé*

Cette approche peut s'appliquer parfaitement à une MDVP *"1 écrivain - multiples lecteurs"*. Le temps de localisation d'une donnée partagée est très faible. Les processus qui font une lecture de la donnée dispose d'une copie (en lecture) locale pour des accès (en lecture) ultérieurs. Par contre, il y a des risques de goulot d'étranglement puisque tous les processus doivent communiquer avec le seul processeur gestionnaire pour accéder aux données partagées (figure 3.15). En outre, lorsque les processus désirent modifier la donnée, ils ne possèdent pas de copie locale accessible en écriture. Ils envoient systématiquement donc la nouvelle valeur de la donnée au gestionnaire.

Pour éviter cette perte de parallélisme due à l'unicité du gestionnaire, les approches suivantes ont eu pour but de distribuer la tâche de gestion des données partagées parmi les processus. De plus, afin de permettre la réalisation de MDVP *"multiples écrivains - multiples lecteurs"*, la gestion d'une donnée et la possibilité d'avoir une copie en écriture ont été dissociées par l'introduction de la notion de processus Propriétaire.

3.6.2 Approche distribuée statique

Cette approche nécessite la définition de deux types de processus différents : le gestionnaire et le propriétaire.

Définition 3.14 *On appelle propriétaire d'une donnée, un processus qui dispose dans sa mémoire locale d'une copie de la donnée partagée accessible en écriture. Le propriétaire d'une donnée partagée peut être indépendant du gestionnaire de cette même donnée. Par contre s'il existe plusieurs copies en écriture d'une même donnée, le propriétaire est celui qui dispose de la copie principale (copie de référence en cas de litige entre différentes écritures (voir paragraphe 3.5).*

Définition 3.15 *On appelle gestionnaire d'une donnée, un processus qui connaît la structure de la donnée partagée et le propriétaire de la donnée.*

Chaque donnée partagée est associée à un processus gestionnaire fixé. Ce processus est réparti de manière statique parmi l'ensemble des processus. Une des techniques les plus souvent utilisées est de répartir les gestionnaires de manière équitable (cyclique) entre tous les processus du réseau. Ce gestionnaire est utilisé pour connaître le propriétaire de la donnée lors du premier accès à cette donnée. Le gestionnaire est aussi utile pour retrouver le propriétaire d'une donnée qui s'est déplacé.

Quand un processus désire consulter le contenu d'une donnée partagée dont il ne connaît pas le propriétaire, il s'adresse au gestionnaire de cette donnée qui, par l'intermédiaire du propriétaire, lui renvoie le contenu valide de cette donnée.

Par contre si un processus veut modifier une donnée partagée, plusieurs techniques de *négociation* sont possibles :

- **Propriétaire immobile** (figure 3.16) : dans cette approche, le propriétaire est celui qui a accédé pour la première fois à la donnée partagée au début de l'exécution. Le propriétaire est fixé une fois pour toutes et ne peut varier au cours de l'exécution. Il est donc le seul processus habilité à modifier la copie principale de la donnée partagée. Le gestionnaire n'est utilisé que pour le premier accès aux données. Bien entendu, lors du premier défaut en écriture (figure 3.16) si le propriétaire est déjà connu, la demande lui y est retransmise. Le processus propriétaire mémorise la liste des processus qui ont accédé à la donnée afin de gérer la liste des copies (*copy sets*).

  Lors des accès suivants, les processus peuvent donc se dispenser de communiquer avec le gestionnaire et s'adresser directement au propriétaire. Cette approche est intéressante puisqu'elle permet de distribuer les propriétaires en fonction des premiers accès effectués. De plus, le propriétaire étant fixé, il est très facilement accessible par tous les processus qui communiquent directement avec lui. Par contre, la technique du *Propriétaire Immobile* peut se révéler peu performante lors de l'utilisation de données intensivement partagées. En effet, un autre processus qui accède intensivement à une donnée a alors intérêt à devenir propriétaire de cette donnée ;

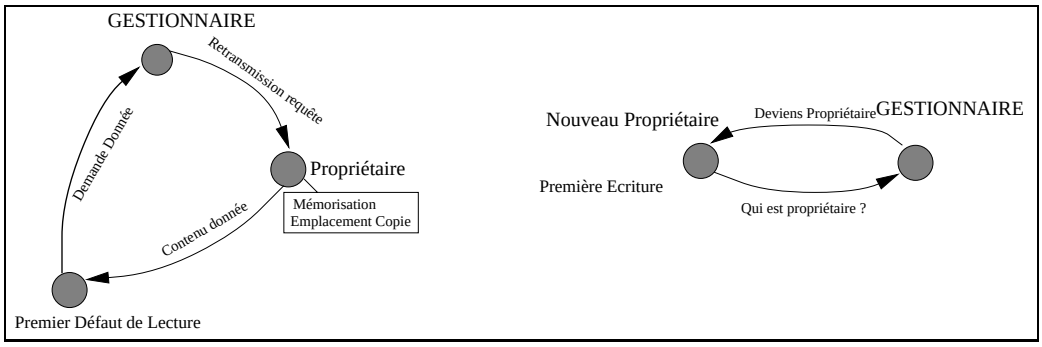


FIG. 3.16 - *Technique du propriétaire immobile lors du premier accès. A gauche, premier accès en lecture pour un processus; le gestionnaire est nécessaire pour trouver le propriétaire. A droite, un processus réalise une première écriture et devient ainsi le propriétaire de la donnée. utilisation du gestionnaire*

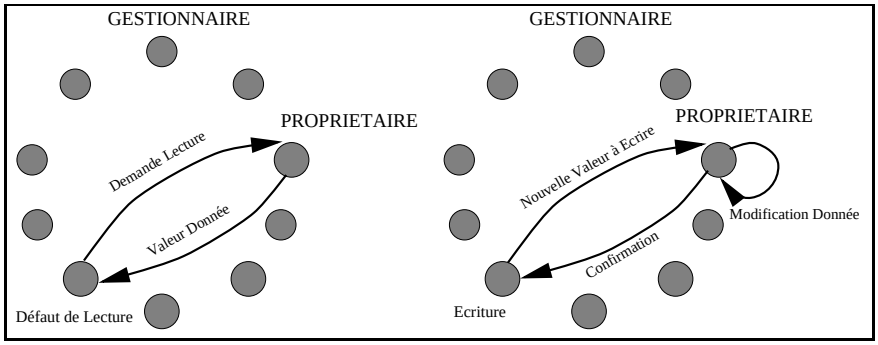


FIG. 3.17 - *Propriétaire immobile : lors des accès suivants en lecture et en écriture, les processus s'adressent directement au propriétaire.*

– **Propriétaire variable - Gestionnaire fixé** : lors des premiers accès aux données, l'utilisation du processus gestionnaire est identique à celle du protocole du *Propriétaire Immobile*. Par contre, lors des accès ultérieurs, on considère qu'il est plus judicieux de changer temporairement de propriétaire lors d'écritures sur une donnée partagée. Ainsi lorsqu'un processus modifie une donnée, il se retrouve momentanément en possession de la copie principale (Figure 3.18). Il peut donc réaliser plusieurs modifications sur cette donnée de manière locale. Ce transfert de propriétaire est réalisé par le processus gestionnaire. Ainsi chaque accès ultérieur nécessite une communication avec le processus gestionnaire qui est le seul à connaître le propriétaire actuel de la donnée.

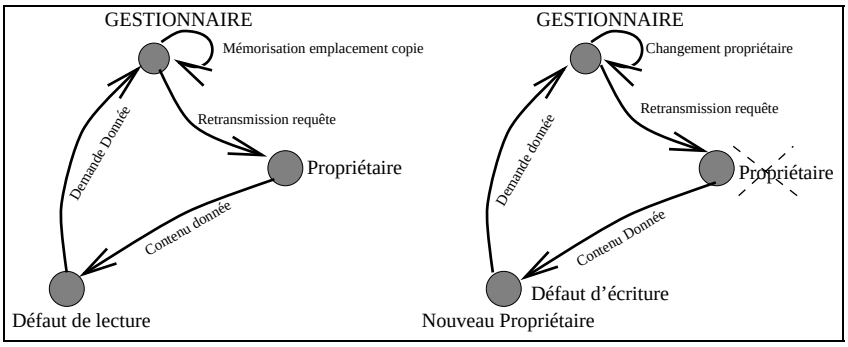


FIG. 3.18 - *Propriétaire variable et Gestionnaire fixé : à gauche, le processus demande au gestionnaire le propriétaire actuel de la donnée. A droite, lorsqu'un processus effectue une écriture, il en fait la demande au gestionnaire qui le considère alors comme le nouveau propriétaire.*

Ces approches distribuées statiques permettent de retrouver un bon degré de parallélisme, par contre la répartition *a priori* des gestionnaires au début de l'application ne tient pas compte de la distribution des accès qui auront lieu au cours de l'exécution. C'est la raison pour laquelle certains systèmes préfèrent implémenter des protocoles dynamiques de gestion des données.

3.6.3 Approche dynamique

On a vu avec l'approche statique et la notion de propriétaire variable que les fonctions de gestionnaire et de propriétaire sont, en fait, assez proches. Les techniques dynamiques préfèrent fusionner les deux processus en un seul, à la fois gestionnaire et responsable de la copie principale de la donnée. On ne parle donc plus de processus gestionnaire mais uniquement de processus propriétaire qui doit donc être redéfini.

Définition 3.16 *On appelle Propriétaire d'une donnée, le processus associé à cette donnée. Il dispose de la liste des copies présentes dans le réseau et de la copie principale de la donnée.*

Dans cette approche, le propriétaire d'une donnée partagée varie en cours d'exécution au gré des requêtes de modification faites par les processus.

Lors du premier accès en écriture à une donnée, le processus demandeur devient propriétaire de celle-ci. Deux techniques sont possibles pour que les autres processus connaissent le nouveau propriétaire :

- Propriétaire actif : le nouveau propriétaire diffuse un message informant l'ensemble des autres processus qu'il est devenu le propriétaire de la donnée. Tous les processus mettent à jour leur table des données pour mémoriser le nouveau propriétaire. ;
- Propriétaire paresseux : lors du premier accès, le nouveau propriétaire ne signale rien aux autres processus. Par contre, lorsqu'un processus veut accéder pour la première fois à une donnée dont il ne connaît pas le propriétaire, il doit demander à tous les autres processus si le propriétaire n'existe pas déjà. Les accès suivants sont réalisés grâce à un propriétaire *probable*. En effet lors d'un accès en écriture (figure 3.19) le processus demandeur récupère la copie principale et devient ainsi le nouveau propriétaire de la donnée. Mais ce changement de propriétaire n'est pas connu des autres processus. Ainsi lors d'un accès, un processus peut demander au processus qu'il croît propriétaire la valeur d'une donnée partagée, alors que ce processus n'est plus le propriétaire actuel de la donnée. La demande est alors retransmise entre les anciens propriétaires pour aboutir au détenteur actuel de la copie principale (figure 3.19) . Il existe différentes techniques, non détaillées ici, qui permettent de réduire le chemin nécessaire à l'obtention de la copie principale [LH89].

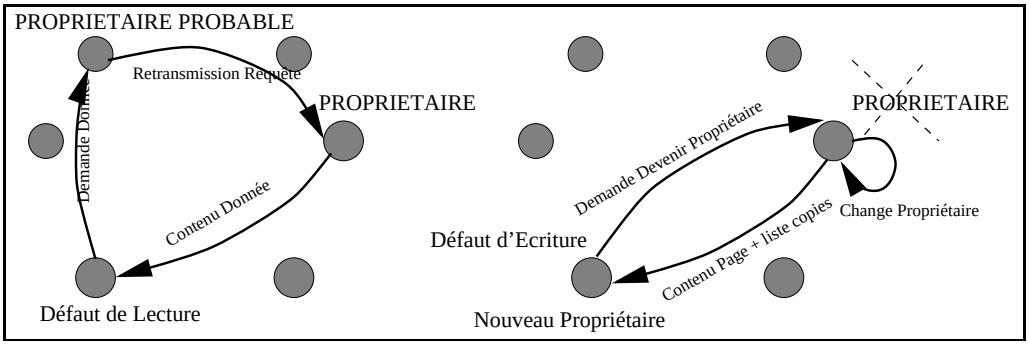


FIG. 3.19 - *Approche dynamique : le propriétaire paresseux. Toute lecture implique la recherche du véritable propriétaire de la donnée. Chaque écriture se traduit par un changement de propriétaire.*

Ces deux protocoles dynamiques ont chacun leurs avantages et leurs inconvénients. Le protocole du *propriétaire actif* est plus utilisé dans le cas d'objets faiblement partagés en écriture. En effet, chaque écriture (changement de propriétaire) nécessite l'envoi du nouveau propriétaire aux autres processus. Par contre, l'accès en lecture est performant puisque chaque processus connaît à tout moment le propriétaire de la donnée. Le protocole du *propriétaire paresseux* s'adapte parfaitement à des données fortement partagées en écriture. Chaque changement de propriétaire ne nécessite que quelques communications localisées. Par contre, une lecture peut être assez coûteuse puisqu'elle peut nécessiter le parcours d'une chaîne de propriétaires *probables* pour aboutir au véritable propriétaire de la donnée.

La gestion de propriétaire des données partagées est fondamentale dans la réalisation d'un système de MDVP dans la mesure où elle en détermine, en grande partie les performances.

3.7 Mise à jour des données partagées

La duplication des données rend les protocoles de cohérence beaucoup plus complexes, surtout en ce qui concerne la mise à jour des données. En fait, les stratégies de mise à jour sont peu nombreuses et n'ont pas évolué entre les différents systèmes ; elles peuvent être divisées en deux types d'approches :

3.7.1 L'invalidation des données

Ce protocole est basé sur des techniques connues de gestion de caches locaux. Lorsqu'une donnée est modifiée par un processus, celui-ci doit prévenir les autres processus. A cet effet, il envoie un bit d'invalidation à tous les processus ou au sous-ensemble des processus qui disposent d'une copie si le système le connaît. Ainsi on aboutit à deux types de comportements suivant les systèmes :

- s'il n'existe qu'une seule copie en écriture (écrivain unique), le système invalide toutes les copies en lecture seule de la donnée présentes chez les autres processus ;
- s'il existe plusieurs copies en écriture, le système invalide toutes les copies qu'il s'agisse de copies en écriture ou en lecture seule. Si, au même moment, un processus invalide lui-aussi sa copie en écriture après modification, c'est aux protocoles de cohérence de déterminer la nouvelle valeur valide de la donnée partagée.

Une bonne utilisation de ce protocole repose à la fois sur des aspects techniques et sur des aspects applicatifs :

- la diffusion de petits messages d'invalidation dans le réseau doit se faire à un coût faible. Ainsi l'utilisation de ce protocole dans des systèmes répartis (tel qu'un réseau de stations) est difficilement envisageable puisque le coût d'envoi des messages est surtout basé sur le nombre de messages et non sur la taille des messages. Ce genre de protocole est donc plutôt utilisé dans des architectures parallèles ;

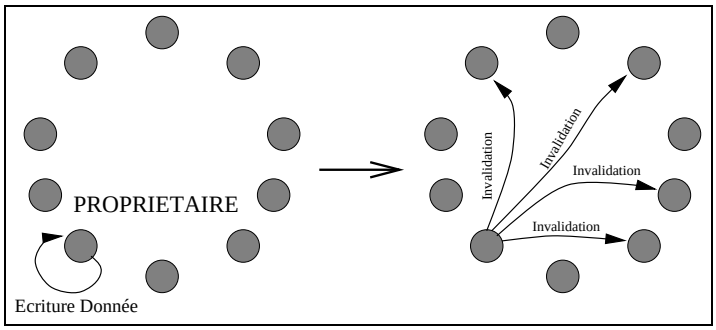


FIG. 3.20 - *Protocole d'invalidation des données*

– les données doivent être faiblement partagées par les processus. Sinon, on peut aboutir à un exemple tel que celui de la figure 3.21 dans laquelle une donnée a été invalidée chez tous les processus. Si ceux-ci ont aussitôt besoin de cette donnée, ils doivent la redemander au propriétaire. Celui-ci retransmet la nouvelle valeur en faisant une diffusion à tous les processus ce que peu de systèmes permettent d'effectuer de manière efficace. Ainsi la troisième étape se traduit souvent par des communications séquentialisées à destination de tous les processus. Ce comportement peut donc générer un gros volume de communications.

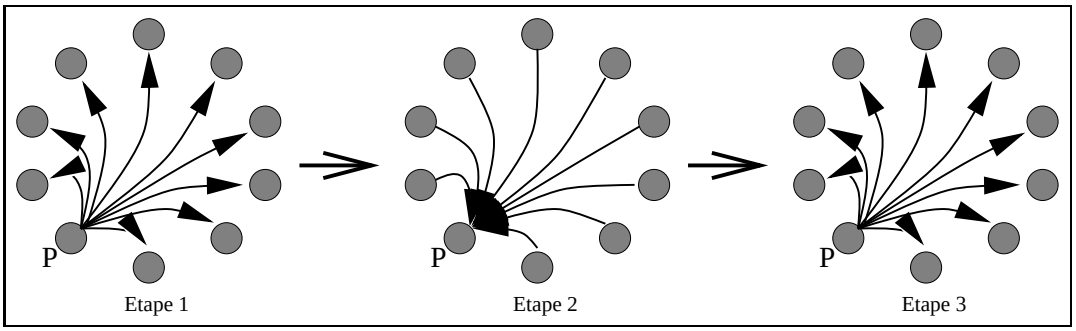


FIG. 3.21 - *Invalidation d'une donnée intensivement partagée en trois étapes. Première étape : le processus propriétaire P invalide la donnée. Deuxième étape : tous les processus demandent la nouvelle valeur de la donnée. Troisième étape : le processus propriétaire communique la nouvelle valeur à tous les processus.*

3.7.2 L'écriture-diffusion

Une autre approche a été proposée et utilisée dans certaines MDVP [HS92b, TKB92] afin de permettre une meilleure adaptation à des systèmes distribués. Dans ces protocoles, dits d'écriture-diffusion (ou écriture-mise à jour), on considère qu'il est à peu près aussi coûteux d'envoyer un bit d'invalidation que la donnée elle-même. On préfère donc invalider les copies des données partagées en envoyant directement la nouvelle valeur de la donnée aux autres processus. De même que pour l'invalidation des données, la diffusion des nouvelles valeurs peut être globale ou simplement limitée à l'ensemble des processus qui partagent la copie.

Ainsi, on aboutit à deux types de comportements suivant l'implémentation des systèmes :

- s'il n'existe qu'une seule copie en écriture (écrivain unique), le système met à jour toutes les copies en lecture seule de la donnée présentes dans les autres processus ;
- s'il existe plusieurs copies en écriture, le système met à jour toutes les copies en lecture seule. Quand un processus qui dispose d'une copie en écriture reçoit une nouvelle valeur, c'est le modèle de cohérence choisi qui détermine la nouvelle valeur de la donnée partagée.

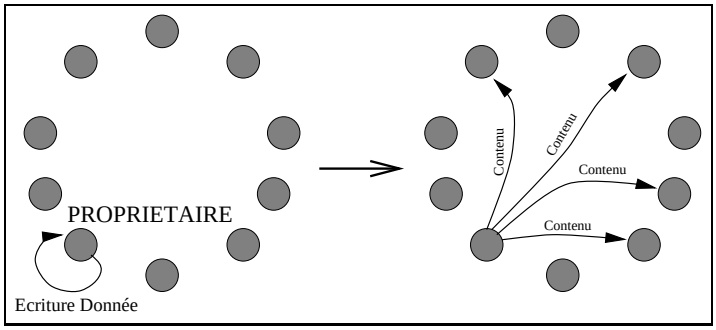


FIG. 3.22 - *Le protocole d'écriture-diffusion*

L'utilisation de l'écriture-diffusion repose sur différents aspects techniques et applicatifs :

- la diffusion des nouvelles valeurs peut parfaitement s'adapter à une utilisation sur une architecture répartie ;
- ce protocole limite la génération de nombreux messages échangés dans le cas d'une donnée intensivement partagée. Comme on peut le voir dans la figure 3.23 on obtient un nombre de messages minimal ;
- l'écriture-diffusion doit être utilisée avec précaution lorsqu'elle s'applique dans des systèmes à base d'objets. En effet, ces systèmes permettent le partage des objets complexes dont la communication sur le réseau est coûteuse. Ainsi, le système doit éviter de mettre à jour inutilement ces objets.

3.7.3 Quel protocole choisir ?

Comme nous l'avons vu, le choix d'un protocole de mise à jour repose sur de nombreux paramètres. Il y a un compromis à faire entre la génération de nombreux petits messages et un volume plus limité de messages plus conséquents. Leur étude permet de réaliser quelques déductions.

- Dans le cas de partage de donnés qui nécessitent beaucoup de mises à jour (intensivement partagés), l'emploi d'un protocole d'écriture-diffusion réduit le nombre de messages émis ;

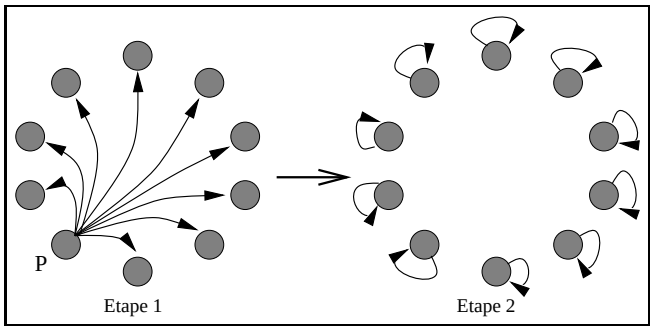


FIG. 3.23 - *Mise à jour d'un objet intensivement partagé à l'aide de protocoles d'écriture-diffusion. Première étape, le propriétaire P met à jour toutes les copies. Deuxième étape: les processus qui veulent accéder à la donnée peuvent travailler avec leur copie locale, puisqu'ils disposent de la nouvelle valeur de la donnée partagée.*

- Pour des données faiblement partagés, les protocoles d'invalidation des données limitent la quantité d'informations transmises sur le réseau ;

- L'utilisation de gros objets partagés favorise les protocoles d'invalidation qui limitent leur coûteuse transmission ;

- Le partage d'objets de petite taille suggère l'emploi d'un protocole d'écriture-diffusion, puisque le coût d'envoi d'un bit d'invalidation est très proche de celui de l'envoi de l'objet ;

- La combinaison des paramètres (taille de l'objet, intensité de partage) aboutit à un compromis entre l'utilisation des deux protocoles.

Ces deux protocoles peuvent être utilisés indépendamment du modèle de cohérence choisi. Nous verrons, par la suite, que quelques systèmes (comme MUNGI [HERV93] ou DOSMOS) essaient, en les combinant, de tirer avantage des deux protocoles.

3.8 Espace d'adressage unique ou objets partagés ?

Les MDVP, on l'a dit, peuvent être divisées en deux grandes familles selon la granularité de partage choisie. Les premiers systèmes décrits dans la littérature étaient des systèmes de Mémoire Virtuelle Partagée proposant un espace d'adressage unique fondé sur un mécanisme de partage de pages mémoire de taille fixe. Par la suite, apparurent les systèmes à Mémoire Distribuée Partagée fondés sur un mécanisme de partage d'objets de granularité (taille) variable en fonction de l'application.

3.8.1 Espace d'adressage virtuellement partagé

Structures de données

La granularité de partage est fondamentale dans l'implémentation d'une Mémoire Virtuelle Partagée. La bonne utilisation de pages mémoire dépend grandement de l'architecture matérielle sous-jacente et du réseau d'interconnexion. La taille moyenne des pages est le plus souvent de quelques kilo-octets. A chaque page est associée une structure de donnée qui contient les informations de gestion les plus usuelles telles que les droits d'accès, les champs de synchronisation, le propriétaire, la liste des processus qui disposent d'une copie... Même si des pages de plus grande taille peuvent réduire les coûts de gestion de la mémoire partagée, elles augmentent aussi les risques de faux partage (paragraphe 3.8.1). Réciproquement, des pages de petite taille nécessitent des tables de gestion de grande taille importante.

Espace d'adressage

Le but principal des systèmes de Mémoire Virtuellement Partagée est de pouvoir fournir un grand espace d'adressage virtuel unique constitué de l'ensemble des mémoires locales des processeurs. Les applications manipulent leurs données en utilisant des adresses logiques sans référence à la localisation physique des donnés. C'est donc au système de prendre en charge la gestion des adresses mémoire logiques et leur transposition en adresses physiques.

Remplacement de pages

Comme dans un système centralisé, un système de Mémoire Virtuellement Partagée doit être capable de procéder à des remplacements de pages lorsqu'un processus désire charger une page distante dans sa mémoire locale et que celle-ci ne dispose plus d'espace mémoire. La technique la plus couramment utilisée est la suivante : si le processus a une copie d'une page dont il n'est pas propriétaire, il la désalloue et prévient le propriétaire qu'il ne possède plus de copie locale de cette page. Par contre, si le processus est propriétaire de toutes les pages présentes dans sa mémoire locale, il doit trouver un emplacement libre chez un autre processus ou détecter un processus qui possède une copie d'une page dont il est le propriétaire. Le système doit ensuite effectuer une migration de propriétaire et, si nécessaire, un transfert de la page.

Le problème du faux partage

Le placement des données à l'intérieur des pages est réalisé par le système d'exploitation. Dans le cas d'utilisation de petites données dont la taille est inférieure à celle d'une page, il peut arriver que plusieurs données soient localisées dans la même page. Or c'est cette page qui est partagée entre les processus et non les données, elles-mêmes ! Le faux partage intervient quand plusieurs processus accèdent à des données différentes localisées dans une même page de mémoire partagée.

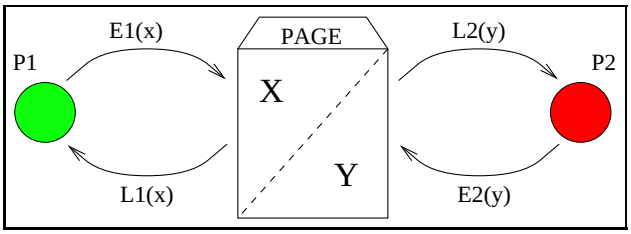


FIG. 3.24 - *Faux partage entre deux processus*

Lorsque le système est fondé sur une cohérence forte, cela pose un problème de verrouillage de la donnée qui se traduit par un effet *ping-pong* lorsque la page est fréquemment accédée en écriture par plusieurs processus. Les processus acquièrent, en effet, successivement le droit d'écrire dans la page en verrouillant l'accès à l'ensemble des données stockées dans cette page. Les performances de la MDVP chutent alors fortement puisque le système transmet incessamment cette page entre les processus ce qui se traduit par une attente avant chaque écriture. De plus, les accès aux données présentes dans la page sont séquentialisés. Ainsi, dans la figure 3.24, les processus *P1* et *P2* pourraient travailler en parallèle sur leur donnée (X ou Y). Mais le faux partage séquentialise les accès : *P1* ne peut ainsi modifier X que lorsque *P2* a fini d'accéder à Y. Tous les systèmes de Mémoire Virtuellement Partagée tentent de résoudre le problème du faux partage. Plusieurs méthodes ont été implémentées; MIRAGE [FP89] propose d'adjoindre un paramètre temporel à la page afin d'empêcher l'effet ping-pong. Le système KOAN [BMP93] permet l'accès et la modification simultanés de sous-parties d'une même page. Enfin, d'autres systèmes comme MIDWAY [BZ91] ou MUNIN [BCZ90] utilisent des techniques de compilation pour améliorer le placement des données à l'intérieur des pages.

Transparence et adaptation

Le développement d'applications au-dessus de MDVP paginées apporte à l'utilisateur un réel confort de programmation. Tous les accès aux donnés partagées sont, en effet, transparents et gérés par le système de MDVP. Mais cette gestion totale se traduit par un gel de tous les paramètres du système. Ainsi l'utilisateur n'a aucun contrôle sur le placement de ses données et sur la granularité d'accès. Les pages ont une taille fixée qu'il ne peut modifier et qu'il ne connaît peut-être même pas ! Les données sont placées à l'intérieur des pages de manière totalement arbitraire par le système d'exploitation. Cela peut aboutir à des difficultés lors de l'exécution (faux partages, goulots d'étranglement...) dont l'utilisateur n'a aucune idée et sur lesquelles il n'a aucun (ou presque) moyen d'action. Ce manque d'adaption aux applications et cette opacité sont les principaux problèmes générés par les mémoires distribuées virtuellement partagées à base de pages.

Portabilité et performances

Tous les systèmes à base de pages sont conçus à l'aide d'une implémentation bas-niveau. Ils requièrent pour la plupart une modification du système d'exploitation, qu'ils soient dédiés à une

machine parallèle (KOAN...) ou à un réseau de stations de travail (IVY...). Ce dialogue direct avec l'architecture sous-jacente leur permet d'atteindre de bonnes performances pour des applications spécialisées à ce type de machines. Mais la réalisation de telles MDVP est délicate et leur mise en oeuvre est complexe et coûteuse car elle nécessite un lourd développement. De plus, la faible pérennité des architectures parallèles ou réparties freine le développement de ces systèmes qui restent souvent à l'état de prototypes. Enfin, la spécificité de leur implémentation peut nuire à une généralisation de leurs résultats dans la mesure où il est souvent impossible de comparer des systèmes fondés sur la spécificité du système d'exploitation et du matériel sous-jacents.

3.8.2 Objets distribués partagés

Alors que de nombreux systèmes ne considèrent les objets partagés que comme un ensemble désordonné de pages (par exemple, le système ANGEL [WSG+92a]), de nombreux systèmes de MDVP ont implémenté de réelles notions d'objets partagés (ex: LINDA [CG89], CLOUDS [DAM+90] ou ORCA [TKB92]).

Gestion de haut niveau

Contrairement aux systèmes à base de pages, les MDVP à objets sont plus concernées par des problèmes de concurrence ou coopération entre processus que par une réelle gestion de la mémoire [Ray92]. Ainsi les problèmes de fractionnement de la mémoire et de remplacement d'objets ne sont jamais pris en compte par ces systèmes.

Transparence et adaptation

L'utilisation d'objets permet une adaptation parfaite aux contraintes de l'application. A l'exception de systèmes comme ADAM [MF92] qui travaillent avec des objets de taille fixée, les autres MDVP permettent l'utilisation d'objets de taille variable. L'utilisateur peut ainsi contrôler le placement de ses données et il dispose d'une granularité adaptable. Ceci permet en outre de manipuler des objets partagés complexes (matrices, listes, structures...). Par contre cette adaptation de la granularité est contrebalancée par le fait que les applications peuvent parfois être moins portables que lors de l'utilisation d'une mémoire paginée. En effet, ces systèmes fournissent un espace partagé mais seulement accessible à l'aide de procédures ou macros spécialisées (LINDA, CLOUDS...) ou d'une programmation orientée-objet (ADAM). La tendance actuelle des nouveaux systèmes à base d'objets est d'offrir une MDVP fournissant un accès transparent aux objets, la seule spécificité du système résidant sur l'utilisation des opérateurs de cohérence.

Portabilité et performances

Les systèmes de MDVP à base d'objets sont généralement réalisée comme une sur-couche logicielle au système d'exploitation existant (sous la forme de bibliothèques ou librairies). Ainsi la mise

en oeuvre de tels systèmes est grandement simplifiée puisqu'elle est basée sur une programmation haut-niveau. Cela permet de disposer de MDVP indépendantes de l'architecture sous-jacente et portable sur différentes plates-formes. On peut ainsi comparer les performances de ces systèmes suivant leur implémentation. Par contre, cette portabilité se paie parfois par des performances amoindries par rapport à des systèmes paginés. L'accès aux objets est, en effet, réalisé par l'intermédiaire de nombreuses couches logicielles coûteuses en temps. Cette contrainte a ouvert la voie aux recherches sur les protocoles de cohérence relâchée qui limitent les communications générées pour la gestion des données partagés. Cette limitation des performances n'a cependant pas empêché le fort développement de systèmes comme LINDA [ACG86] qui demeure la MDVP la plus distribuée et utilisée dans le monde.

3.9 Améliorations récentes

Sans se différencier fortement des idées initiales des MDVP, de nombreuses recherches ont été menées en vue d'améliorer les fonctionnalités des systèmes de MDVP :

3.9.1 Tolérance aux pannes

La conception de MDVP sur des architectures réparties a amené les concepteurs à proposer des systèmes présentant une certaine tolérance aux pannes : les mémoires partagées recouvrables. Ces nouveaux types de mémoire partagée permettent l'exécution d'applications sur des systèmes dont les nœuds peuvent défaillir. Les techniques mises en oeuvre tournent autour de la collecte d'informations au fur et à mesure de l'exécution, sous forme de point d'arrêts (équivalents aux *snapshots* utilisés en bases de données). Ainsi, si une défaillance est détectée, l'exécution peut reprendre un cours normal à partir du dernier point d'arrêt qui avait été sauvegardé. Depuis peu, quelques MDVP intègrent ce concept de recouvrabilité en utilisant des composantes logicielles [KCG+95] ou du matériel dédié au stockage des points d'arrêt [WF90].

3.9.2 Persistence

En se basant sur le fait que de nombreuses applications utilisent des données partagées qui varient peu et dont la durée de vie est supérieure à celle des applications qui les consultent, des recherches sont menées sur la persistence des données partagées [AJL92, SPFA94, WSG+92a]. Ces données persistentes s'adaptent parfaitement aux besoins d'applications coopératives dans lesquelles les données sont partagées par un petit nombre de processus mais dont la durée de vie des données partagées est importante [WCF+93]. enfin, afin d'optimiser la gestion de la mémoire, de nombreux systèmes implémentent des algorithmes de ramasse-miettes qui réorganisent l'espace mémoire utilisé par des données distribuées et persistentes [LSB92, FS94, SF95].

CHAPITRE 4

Principaux systèmes de Mémoire Distribuée Virtuellement Partagée

En une décennie, de nombreux systèmes à base de Mémoire Distribuée Virtuellement Partagée ont été proposés dans la littérature. L'objet de ce chapitre est de dresser un panorama des principaux systèmes qui ont abouti aux avancées les plus significatives à la fois du point de vue théorique mais aussi du point de vue applicatif. Pour une description plus complète des autres systèmes, on pourra se reporter au paragraphe 4.3.

Nous avons relevé, dans la littérature, l'existence de plus d'une centaine de systèmes de MDVP différentes. Cette mise en oeuvre d'un espace virtuel global a été mené dans trois grandes directions. De nombreux systèmes ont été implémentés comme un ajout au système d'exploitation de certaines machines parallèles (KOAN, PLATINUM, MERMAID...). D'autres, au contraire s'adressent au cadre plus large des réseaux de stations de travail (IVY, MIRAGE, MUNIN, TreadMarks, MIDWAY...). Une autre grande famille de systèmes propose des librairies spécialisés (LINDA, ORCA, DOSMOS...). Enfin, des systèmes intègrent des composantes matérielles dédiées et sont implémentés au coeur même de l'architecture des machines parallèles (GALACTICA [WLJI+93], DASH, MEMMET...).

Dans ce chapitre, nous avons préféré regrouper les différents systèmes de Mémoire Distribuée Virtuellement Partagée suivant la granularité des données qu'ils partagent : pages de données ou objets. Cette division nous paraît plus pertinente qu'une taxonomie fondée sur les caractéristiques internes des systèmes (bibliothèque, système d'exploitation) qui relèvent exclusivement de l'implémentation d'une MDVP et non pas de son mode effectif d'utilisation et de conception. Ce chapitre ne s'intéresse pas aux véritables machines à Mémoire Distribuée Partagée (Cray T3D [LBR94b], KSR-1 [Ken92]). Celles-ci, mentionnées dans la figure 4.1 mettent en jeu des composantes matérielles spécifiques qui ne permettent pas de comparaison avec les MDVP logicielles. Les différentes MDVP analysées, ici, sont donc exclusivement des systèmes logiciels.

Le lecteur pourra se reporter à la figure 4.2 qui dresse une synthèse globale de l'ensemble des MDVP présentées ici.

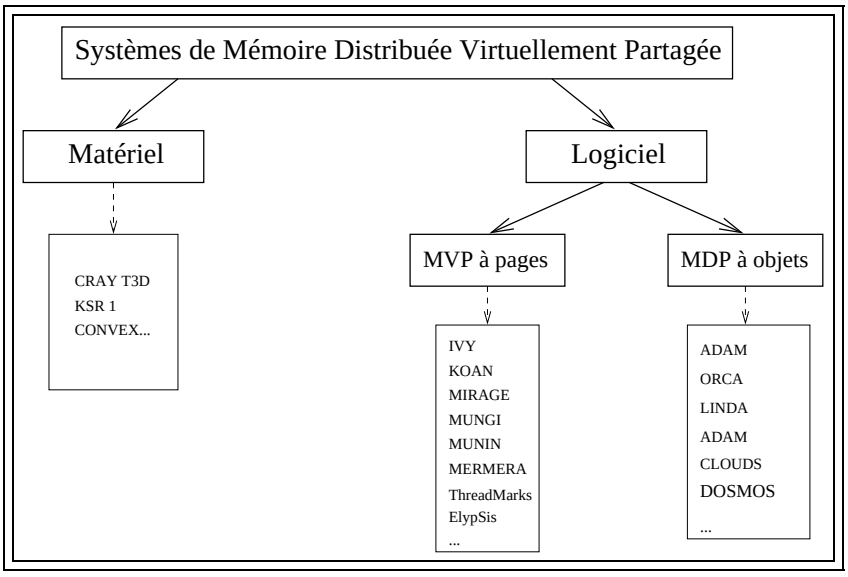


FIG. 4.1 - *Taxonomie des principaux systèmes*

4.1 Systèmes de MDVP à base de pages

IVY

Ce système est considéré comme la première MVP ayant été implémentée sur un système parallèle [LH86, LH89]. Il a été développé sur un réseau de stations de travail *Apollo Domain* [Li88] et sur la machine parallèle à topologie en hypercube iPSC/2 d'Intel [LS89]. Caractéristiques principales :

- La mémoire partagée est partitionnée en pages qui sont déplacées à la demande des processeurs. Ainsi une copie de la table de pages réside sur chaque processeur, ce qui lui permet de connaître les principales informations maintenues pour chaque page (droits d'accès, propriétaire, ensemble des copies).
- Le système IVY est basé sur des gestionnaires distribués statiques. Lorsque un défaut de page distant se produit, le processus demandeur est bloqué le temps que le gestionnaire de mémoire retrouve la page manquante sur la mémoire d'un autre processeur ou sur disque.
- Les gestionnaires mémoire sont responsables de la cohérence de la mémoire partagée à tout instant (cohérence forte). Dans IVY, une page appartient à un seul processeur. Ce propriétaire est le dernier processeur à avoir modifier la page.
-  Une seule copie de la page est disponible en écriture. Avant chaque modification, le processeur qui a le droit d'écriture sur la page, invalide toutes les copies en lecture seule distribuées dans le réseau.

IVY est un travail fondamentale pour le développement des systèmes de Mémoire Distribuée Virtuellement Partagée. Il a montré que la réalisation d'une MDVP était possible sur une architecture parallèle à faible échelle [Li88]. Bien sûr ce système contient de nombreuses restrictions.

Chaque écriture à la mémoire partagée est, on l'a dit, globalement synchronisée. Le *Page trashing*[1] peut apparaître quand de nombreux processeurs veulent modifier une page au même moment. Les risques de faux partage existent puisque IVY ne teste pas la possibilité d'avoir des données différentes sur la même page.

KOAN / MYOAN

Dans le système KOAN [BBLP91, LP92, BEP93, BKP93, BMP93], les pages de données sont réparties parmi les processeurs en utilisant une fonction de placement qui assure que chaque processeur a le même nombre de pages. Chaque mémoire locale est divisée en deux parties :

- La première est utilisée pour sauvegarder le code des processus et leurs variables locales.
- La seconde sert de cache logiciel pour sauvegarder les pages. On utilise une politique LRU (*Least Recently Used* - moins récemment utilisée) pour la gestion des défauts de pages
- Le noyau de KOAN consiste en 7 serveurs et gestionnaires (défaut lecture/écriture, changement des accès, messages de confirmation...)
  Sur chaque processeur, KOAN doit séquentialiser les demandes locales (défauts de pages) et les demandes extérieures (demandes de pages) qui ont lieu pour une page donnée.
  Le système utilise un arbitre qui décide soit de transmettre les demandes aux serveurs concernés, soit de les sauvegarder dans une file d'attente soit de les rejeter en les renvoyant au processeur demandeur.
- La MVP KOAN utilise un gestionnaire distribué fixé avec un protocole d'invalidation qui garantit la cohérence de la mémoire à tout instant. Les serveurs partagent deux structures de données :
  - Une table des pages : cette table est utilisée pour la translation des adresses virtuelles en adresses physiques. Elle contient également les droits d'accès à la page.
  - Une table contenant la liste des pages présentes dans la mémoire locale. Pour chaque page, on mémorise les identificateurs des processeurs possédant une copie en lecture de la page (méthode des *copy sets*, voir paragraphe 5.4.4) et le propriétaire.
- Un remplacement des pages est mis en œuvre quand il y a un défaut de page et que la mémoire est pleine. Le problème intervient lorsque l'on doit enlever une page dont on était le propriétaire en écriture. On réalise alors une migration de la page chez un processeur ayant encore de l'espace mémoire disponible.
- Des expérimentations ont été réalisées sur de petites matrices en utilisant un protocole de cohérence faible : on considère que plusieurs processeurs peuvent modifier en même temps une même page. En fait, chaque processeur n'accède qu'à une sous-partie de cette page. Puis on réalise une fusion et une mise à jour des différentes copies de la page (dérivé de la méthode des

1. Page trashing : Perte de la page

jumeaux avec des écrivains multiples, voir paragraphe 3.5). Cela nécessite une modification importante de l'algorithme parallèle, mais évite les problèmes de faux partage des pages.

-  Les expérimentations effectuées avec le système KOAN font apparaître des résultats intéressants en ce qui concerne des applications numériques [BGL+93, Hah93] ou d'imagerie [BBLP91, BBP94]. Les concepteurs de KOAN ont aussi réalisé différentes expérimentations afin de comparer ce système avec des librairies d'échange de messages [PL92, BGLP95].

- Un portage du système KOAN a été réalisé sur la machine parallèle PARAGON sous la forme du système MYOAN [CPP94] (voir Figure 4.2).

MERMERA

Le système MERMERA [HS92b, HS92a, HS93] est considéré comme l'un des premiers à laisser le choix de la sémantique de cohérence mémoire à l'utilisateur. Cette caractéristique permet au programmeur de choisir parmi un ensemble de protocoles le comportement le plus adapté à son application. Lipton et Sandberg [LS88] ont montré qu'il est impossible d'implémenter une mémoire partagée avec une cohérence forte, dont les temps d'accès soient moins que linéaires au pire cas d'une communication entre les processeurs. C'est la raison pour laquelle différents protocoles de cohérence faibles ont été proposés (les concepteurs de MERMERA désignent ces protocoles sous le nom de mémoires non-cohérentes). En outre, à notre connaissance, MERMERA est l'un des premiers systèmes qui permit de mélanger des cohérences fortes et faibles dans une même application.

Caractéristiques principales :

- MERMERA propose une duplication complète des pages de données. Chaque processeur dispose donc d'une copie de la page ;

- Ce système est implémenté sur réseau de stations de travail et sur la machine parallèle BBN TC2000 ;

- Différents types de sémantiques de cohérence mémoire sont disponibles (voir paragraphe 3.4) : Mémoire cohérente, Mémoire pipelinée, Mémoire Lente, Mémoire localement consistente.

- Le programmeur choisit la sémantique de cohérence de la mémoire en invoquant les primitives MERMERA appropriées. Différents types d'accès en écriture à la mémoire partagée sont proposés : Écriture cohérente, Ecriture-PRAM, Ecriture lente, Ecriture locale. Si l'utilisateur utilise des primitives d'écriture d'un seul type, MERMERA considère que l'application s'exécute avec un seul type de cohérence mémoire. Enfin, toutes les écritures sont au minimum ordonnées pour satisfaire la cohérence locale.

- La cohérence mémoire est implémentée grâce à l'utilisation d'un protocole d'invalidation des données à base d'écriture-diffusion. Une écriture en mémoire partagée met à jour la copie et propage la nouvelle valeur vers tous les autres processus ;

–  Les performances de MERMERA laissent apparaître de bons résultats d'exécution lors de l'utilisation des mécanismes de cohérence faible avec des temps d'exécution de 3 à 65 fois plus rapides qu'avec des opérations fortement cohérentes.

MIDWAY

Le système MIDWAY[BZ91, BZS93] est une Mémoire Distribuée Virtuellement Partagée basée sur une nouvelle forme de cohérence mémoire : la cohérence en entrée (paragraphe 3.4.2).

Caractéristiques principales :

– Ce système est basé sur trois composantes principales : une librairie de fonctions pour annoter le programme parallèle, un système qui implémente la cohérence en entrée et un compilateur ;

– MIDWAY est la première réalisation d'une MDVP basée exclusivement sur la cohérence en entrée. Cette cohérence, combinée avec une mise à jour des données à l'aide de protocoles d'écriture-diffusion, semble fournir des résultats expérimentaux intéressants avec une réduction notable du nombre de communications générées comparé aux cohérences plus fortes.

– Le système MIDWAY est novateur dans le sens où c'est un des premiers systèmes de MDVP à avoir proposé une ébauche d'environnement complet avec la possibilité d'utiliser différents langages de programmation et plusieurs plate-formes de développement.

MIRAGE

La MDVP MIRAGE [FP89] est un système intermédiaire entre le partage de pages et le partage d'objets; il est basé sur un partage de segments et a été implémenté dans le noyau de divers systèmes d'exploitation (e.g. VAX...).

Caractéristiques principales :

– MIRAGE implémente un regroupement des pages en segments. Chaque segment est associé à un site spécifique appelé le site librairie. Celui-ci contrôle toutes les pages du segment ;

– Les pages demeurent néanmoins les unités de distribution qui circulent dans le réseau en raison de leur taille fixe et pour des questions de facilité d'implémentation dans les différents systèmes d'exploitation ;

– Quand un défaut de page apparaît, un gestionnaire détecte si la page demandée est partagée. Si tel est le cas, le site librairie du segment incluant la page est repéré. Le processeur demandant la page effectue alors une requête (en lecture ou en écriture) au site librairie qui stocke les demandes dans une file d'attente avant de les traiter. Ainsi les demandes en écriture sont traitées séquentiellement alors que les demandes de lectures sont traitées par lot pour améliorer les accès ;

– L'originalité de MIRAGE est qu'il essaie de réduire les problèmes de faux partage (voir paragraphe 3.8.1). Cette Mémoire Distribuée Virtuellement Partagée associe ainsi à chaque page de donnée partagée un paramètre temporel qui définit un temps minimum pendant lequel la page ne pourra changer de processeur. De plus ce paramètre s'ajuste dynamiquement pour s'adapter aux différents types d'applications.

MUNGI

Le système MUNGI [HERV93] peut être considéré comme un système d'exploitation à espace d'adressage unique pour un réseau local de machines homogènes.

Caractéristiques principales :

– Le système MUNGI implémente un regroupement des pages en segments, à l'instar du système MIRAGE. Au niveau système, les processeurs ne partagent, en fait, que des pages de données ;

– Quand un défaut de page apparaît, une table des pages, dupliquée sur chaque nœud, est consultée pour localiser l'adresse physique de la page, d'une manière identique à un système de pagination traditionnel, exception faite que la page peut résider sur un site distant. Ainsi les différents états d'une page sont : résidente, sur disque, distante, non-allouée ou inconnue ;

– Chaque page est possédée par un propriétaire unique qui est le nœud disposant de la copie principale de la page. MUNGI permet aussi de dupliquer cette page en proposant plusieurs copies en lecture seule. Une requête d'écriture provoque un changement de propriétaire de page et un transfert du contenu de celle-ci au processeur demandeur ;

– MUNGI utilise un protocole d'invalidation original pour annuler toutes les copies en lecture seule de la page. Pour réaliser cela, le propriétaire de la page gère une liste des sites qui disposent d'une copie de la page. Cette liste est de taille réduite et fixée. Ainsi si la liste de sites déborde, le système considère que cette page est partagée de manière intensive et il diffuse alors le contenu de la page à tous les processeurs. C'est une des grandes originalités de MUNGI de permettre d'utiliser en même temps les protocoles d'invalidation et d'écriture-diffusion.

ElypSis

Le système ELYPSIS [DRDS+91] est une plate-forme de programmation parallèle dédiée à des applications s'exécutant dans un environnement partagé.

Caractéristiques principales :

– L'espace d'adressage proposé par ELYPSIS est divisé en pages et secteurs. Chaque page comprend un nombre fixé de secteurs qui sont en fait, l'unité de donnée partagée par le système.

- ELYPSIS implémente des protocoles de cohérence faiblement ordonnée mettant en oeuvre des points de synchronisation ;

- Ce système propose une exécution hybride qui combine de l'échange de messages pour l'ordonnancement et le contrôle et de la mémoire partagée pour l'implémentation de l'environnement partagé. Mais l'utilisateur ne peut profiter de ce système hybride qui est simplement utilisé pour le portage d'ELYPSIS sur différents types d'architectures ;

-  Les résultats du système ELYPSIS laissent apparaître des performances limitées avec une restriction à des systèmes disposant d'un faible nombre de processeurs (≤ 32).

MUNIN

Le système MUNIN [BCZ90] propose une MDVP qui utilise la puissance d'une mémoire globale virtuelle unique sur des architectures faiblement couplées.

Caractéristiques principales :

- MUNIN permet d'annoter le code parallèle de l'application pour choisir la sémantique d'accès aux pages partagées. Par ce biais, le système permet de spécifier une politique de cohérence adaptée à chaque type de donnée de façon à améliorer les accès.

- MUNIN est le premier système à proposer une implémentation d'une Mémoire Distribuée Virtuellement Partagée fournissant une cohérence à la libération.

-  De nombreuses expérimentations réalisées avec MUNIN, laissent apparaître des performances intéressantes dans le cadre d'applications numériques.

TreadMarks

Proposé par les concepteurs du système MUNIN, le système TreadMarks [ACD+96] permet un partage de pages sur un réseau de stations de travail en reprenant certains concepts du système MUNIN.

Caractéristiques principales :

- Ce système, dédié aux stations de travail utilisant UNIX, est fondé sur l'utilisation de *sockets* et permet ainsi une communication entre les machines. Il est donc présenté comme une librairie qui ne requiert aucune modification du système d'exploitation.

- TreadMarks considère la mémoire comme un ensemble linéaire d'octets accessible uniquement par l'intermédiaire d'une cohérence à la libération. Ce système est basé, en fait, sur une modification de cette cohérence appelée cohérence à la libération fainéante [KCZ92] (voir

paragraphe 3.4.2). En outre, cette Mémoire Distribuée Virtuellement Partagée permet à plusieurs écrivains de modifier en même temps les données partagées grâce à l'utilisation de pages jumelles (voir paragraphe 3.5).

–  Ce nouveau système TreadMarks affiche des résultats intéressants sur des expérimentations en programmation linéaire et algorithmique génétique. De plus, des comparaisons d'expérimentations effectuées entre des versions d'applications au-dessus de TreadMarks et des versions à base d'échanges de messages donnent des résultats tout à fait comparables [LDCZ95].

4.2 Les systèmes de MDVP à base d'objets

ANGEL

ANGEL [WSG+92b, WSG+92a] est un système d'exploitation qui s'adresse à des architectures partagées ou distribuées.

Caractéristiques principales :

– Le système ANGEL introduit une notion d'objets partagés basée sur le partage de pages de données. Un objet n'est en fait qu'un domaine d'adresses mémoire protégées. Ces objets sont persistents et ne dépendent pas du processus qui les a créés.

– Ce système a été porté sur l'architecture TOPSY [WP90] basé sur un ensemble de stations de travail reliées par un réseau en grille.

– Comme l'implémentation est faite jusqu'à un niveau très proche du matériel, la mémoire est considérée comme un seul ensemble d'adresses fortement cohérent.

ADAM

La Mémoire Distribuée Virtuellement Partagée ADAM [MF92] est un système orienté objet dédié à la machine à flot de donnée du même nom.

Caractéristiques principales :

– Les programmes doivent être écrits à l'aide du langage fonctionnel SISAL qui grâce à l'utilisation d'un compilateur dédié essaie d'extraire le parallélisme des applications et génère des blocs de code de taille différentes afin d'améliorer la granularité des applications ;

– Le système ADAM implémente une notion d'objet partagé mais ceux-ci ont une taille fixée et sont accessible par une référence et un index à l'intérieur de l'objet ;

– Ce système propose trois classes d'objets avec différentes caractéristiques d'accès. Les objets locaux peuvent être accédés rapidement par le propriétaire mais causent des attentes pour

des requêtes venant de processeurs distants. Les objets distribués ont généralement une taille importante et sont concurremment lus et modifiés par quelques processeurs. Les objets répliqués sont dupliqués dans les mémoires locales de chaque processeur. Les écritures en parallèle sont séquentialisées d'où le risque de goulots d'étranglement. Les mises à jour utilisent un protocole d'écriture-diffusion ;

- L'originalité du système ADAM réside dans son adressage basé sur des objets. Celui-ci aide le programmeur en fournissant une structure mémoire sémantiquement proche des structures de données habituelles. Cet adressage permet d'améliorer les performances du système en assignant des objets à des classes spécifiques suivant leurs caractéristiques d'accès.

CLOUDS

Le système CLOUDS[RAK89, DAM+90, DCM+90, DLAR91] est un système d'exploitation distribué orienté-objet qui propose une Mémoire Distribuée Virtuellement Partagée.

Caractéristiques principales :

- CLOUDS permet le partage d'objets composés de segments. Pour ce système, un objet est une encapsulation de code et de données. L'utilisateur définit ainsi les fonctions permettant d'accéder à l'objet (points d'entrée) et les services qu'il offre. Cette notion est identique à la notion d'objet des langages orientés objet ;
- Les processus légers (*threads*)sont les seules entités actives dans le système. Un processus léger s'exécute toujours à l'intérieur d'un objet. Il peut accéder à un autre objet en invoquant son point d'entrée ;
- Un noeud mémoire est composé de deux parties : la zone objets mémoire qui contient les segments créés localement et la zone de cache réseau pour les segments venant d'objets distants.
- Le système CLOUDS exploite un mécanisme de synchronisation de processus pour simplifier la gestion de la cohérence mémoire. Le propriétaire d'un segment est le processus qui a créé ce segment, c'est donc lui qui garantit la cohérence de ses données. Dans CLOUDS un processus a accès à un segment jusqu'à ce qu'il le relâche explicitement.
- L'utilisateur dispose de différents types d'accès associés à chaque segment : l'accès *sans-mode* garantit un accès exclusif à un segment, mais ce segment peut être réquisitionné à n'importe quel moment par un autre processus. Le mode *lecture faible* est utile pour des applications qui n'ont pas besoin d'avoir une vue exacte de la mémoire partagée à tout instant. Les modes *lecture* et *lecture-écriture* ne garantissent pas les accès (si d'autres opérations s'effectuent en parallèle). Ainsi un processus peut se trouver bloqué sur un segment jusqu'à ce que l'état de cohérence désiré soit atteint.
-  L'un des inconvénients majeurs du système CLOUDS est que l'utilisateur doit explicitement entourer chaque accès aux données partagées d'une paire de primitives (acquisition, relâchement). Le fait d'oublier de relâcher un objet peut créer des situations d'interblocages.

LINDA

Le système LINDA[ACG86, BH89, CG89, SZ90, NL91] propose de structurer la mémoire partagée comme une base de données. LINDA est souvent considérée comme une Mémoire Distribuée Virtuellement Partagée à part à cause de l'utilisation de tuples qui nécessitent la mise en œuvre d'opérations spécialisées non transparentes à l'utilisateur. Mais la plupart des mécanismes et des protocoles sont suffisamment proches de ceux des MDVP pour que ce système soit mentionné ici.

Caractéristiques principales :

- La mémoire partagée dans LINDA est basée sur un ensemble d'enregistrements appelés tuples. Tous les processus partagent logiquement le même espace de tuples quel que soit l'endroit où les tuples sont mémorisés.
- Les tuples sont accessibles par l'intermédiaire de trois primitives dédiées : *Read* lit un tuple existant de l'espace des tuples. *Out* ajoute un nouveau tuple et *In* lit puis efface un tuple existant.
- Les tuples sont adressés par contenu et non par location. Il est possible d'ajouter ou d'enlever des tuples de l'espace des tuples mais il est impossible de modifier un tuple sans le récupérer. Si un processus veut modifier un tuple, il le récupère dans sa mémoire locale, modifie la valeur du tuple puis le replace dans l'espace des tuples partagés. Avec cette méthode, aucun problème de synchronisation n'est possible.
- La localisation des tuples est complètement transparente pour l'utilisateur; ceux-ci peuvent être centralisés sur le même processeur ou distribués sur plusieurs sites.
-  Atteindre de bonnes performances avec ce système est assez difficile puisqu'il n'y a pas de réelle duplication des données. Certains travaux tels que [BH89] ont montré que les performances du système LINDA dépendent grandement de la granularité des applications. Ainsi seuls les algorithmes parallèles pouvant s'adapter à un parallélisme à gros grain peuvent tirer bénéfice d'un système comme LINDA. Pour pallier à cette limitation, de nombreux ajouts ont été proposés au système LINDA afin d'améliorer les performances d'accès [Sys92, CF94].

ORCA

ORCA [BT88, Tan91, TKB92] est un langage procédural qui permet au programmeur de définir des objets abstraits et des méthodes pour gérer ces objets.

Caractéristiques principales :

- Les objets dans ORCA sont des objets passifs, ils ne contiennent pas de processus. Cependant, chaque objet définit une structure de donnée qui contient la description d'une ou plusieurs opérations qui permettent d'accéder à cet objet.

- ORCA utilise le concept de duplication totale : tous les objets sont partagés par tous les processus. Ainsi, chaque processus a une copie de chaque objet, ce qui garantit de bonnes performances en lecture puisqu'un accès à un objet partagé a un coût équivalent à un accès local.

- Pour permettre la modification des objets partagés, le système ORCA propose un protocole qui permet d'envoyer la nouvelle valeur à l'ensemble des processus (diffusion totale). En outre, pour garantir l'indivisibilité (l'atomicité) des opérations sur les objets, les diffusions sont séquentialisées et ne peuvent s'inter-croiser.

- La sémantique proposée par ORCA est proche de celle proposée par les architectures à mémoire partagée. Mais pour atteindre ce but, l'implémentation d'ORCA nécessite une modification du noyau afin d'utiliser des protocoles de diffusion fiables minimisant le nombre de contentions dans le réseau.

-  ORCA propose une méthode de parallélisation facile des applications. Mais ORCA repose sur des contraintes matérielles (diffusions totales peu coûteuses) qui limitent la portabilité du système à beaucoup de plate-formes de développement.

4.3 Travaux annexes

De nombreux autres systèmes de Mémoire Distribuée Virtuellement Partagée ont été proposés :

- Agora [BF88] est un des premiers systèmes à implémenter des protocoles de cohérence faible pour le partage de structures entre différents processeurs ;

- Arcade [CGCS91] permet de partager des données définies par l'utilisateur dans un réseau de machines hétérogènes ;

- Choices [JC89, CIJ+91] propose un système d'exploitation basé sur une approche orientée objet hiérarchique ;

- Emerald [JLHB88] fondé sur un langage orienté objet, c'est un système qui fournit des mécanismes de MDVP par l'intermédiaire de la mobilité des objets ;

- Gothic [PBR91] propose une Mémoire Distribuée Virtuellement Partagée basée sur une cohérence forte et tolérante aux fautes ;

- Kali [KMR90] est un espace d'adressage global qui permet de partager des structures de données sur des architectures distribuées ;

- Mermaid [Zho92] est implémentée sur un réseau de stations de travail et sur machine parallèle DEC Firely. Il permet de réaliser des applications hétérogènes à l'aide de routines de conversion automatiques ;

Nom de la MDVP	Niveau d'implémentation	Donnée partagée	Cohérence	Mise à jour	Implémentation
ADAM	Système d'exploitation	Objets	Forte	Ecriture-diffusion	ADAM
ANGEL	Système d'exploitation	Pages / Objets	Forte		TOPSY[WP90]
CLOUDS	Système d'exploitation	Objets	Forte	Invalidation	Ra OS Kernel
ELYPSIS	Système d'exploitation	Pages/secteurs	Faiblement ordonnée	Invalidation	Simulateur
IVY	Système d'exploitation	Pages	Forte	Invalidation	Appolo Domain iPSC/2
KOAN	Système d'exploitation	Pages	Forte/faible	Invalidation	iPSC/2
Linda	Exécution	Tuples	Forte	Copie unique	Portable
Mermera	Système	Pages	Forte/faible	Ecr. Diff.	BBN / Lan SPARC
Midway	Système d'exploitation	Pages	Cohérence d'Entrée		Mach
MIRAGE	Système d'exploitation	Segments/Pages	Forte	Invalidation	Divers (VAX...)
MUNIN	Système d'exploitation	Pages	A la libération	Invalidation	Divers
MUNGI	Système d'exploitation	Segments/pages	Forte	Inv./Ecr. Diff.	Réseau local
MYOAN	Système	Pages	Forte/faible	Invalidation	Paragon
ORCA	Système d'exploitation	Objets	Forte	Diffusion totale	Divers
TreadMarks	Librairie	Pages	A la libération	Invalidation	Stations UNIX

FIG. 4.2 - *Comparaison des principaux systèmes de MDVP*

- Le système Platinum [CF89] implémente une mémoire partagée fortement cohérente sur une machine avec une mémoire à accès non uniforme (NUMA) ;
- Spectre [Rai90] est une MDVP dédiée à une implémentation sur une machine à base de transputers pour bénéficier de la rapidité de changement de contexte afin de limiter les temps d'attente ;
- Le projet Inria SOR qui développe des mécanismes de partage d'informations dans des systèmes répartis à grande échelle en intégrant des techniques de ramasse-miettes répartis [SPFA94] et de la persistence des objets (Système Larchant [SF95]).
- Voyager [HT89] reprend les principes élaborés dans le système CLOUDS (paragraphe 4.2) mais adapte la synchronisation des processus et la cohérence des données à la gestion de bases de données ;

On pourra aussi se reporter à [Hel90, NL91, Esk96] qui répertorient d'autres travaux sur les systèmes de Mémoire Distribuée Virtuellement Partagée qui n'ont pas été cités dans ce document.

4.4 Conclusion

La mise en oeuvre des systèmes de Mémoire Distribuée Virtuellement Partagée peut se résumer en trois grandes étapes. La première génération de systèmes de MDVP s'était donné comme but de proposer un modèle de programmation très proche de celui des machines à mémoire partagée. Ces systèmes fournissaient un accès transparent à un grand espace d'adressage unique. Malheureusement les sémantiques de cohérences mises en oeuvre, très fortes, suscitaient des risques de goulots d'étranglement qui n'étaient pas réellement pris en compte dans ces systèmes. Néanmoins, l'apport fondamental de ces travaux a été, d'une part, de démontrer la faisabilité des systèmes logiciels de mémoire distribuée virtuellement partagée; d'autre part de mettre en place leurs fondements théoriques.

La deuxième génération de systèmes de MDVP se consacra aux performances des systèmes en s'intéressant aux machines parallèles ou en intégrant des composantes matérielles dédiées. Cette recherche de performance suscita des implémentations de bas niveau, la plupart des systèmes étant conçus comme des modules intégrés aux systèmes d'exploitation. Cela nécessitait de prendre en compte l'architecture sous-jacente et de modifier le noyau du système en particulier afin d'abaisser les temps de communication. Ces systèmes ont montré l'intérêt des MDVP pour la programmation parallèle et ont permis d'atteindre de bonnes performances. Mais la faible durée de vie des machines parallèles et la spécificité de ces systèmes ont abouti limité leur développement de ces systèmes en raison de leur non-portabilité.

Les systèmes les plus récents s'intéressent autant à la qualité et à la facilité de programmation qu'aux performances. Néanmoins, un certain nombre de points fondamentaux ont été peu ou pas adressés.

Modularité : la grande diversité des architectures parallèles ou réparties nécessitent des systèmes

très modulaires, ce qui n'est pas toujours le cas des systèmes de MDVP.

Hétérogénéité : peu de systèmes permettent de prendre en compte des applications hétérogènes.

Parallélisme massif: tous les systèmes implémentent un partage à plat des données i.e. l'ensemble des données sont partagées par tous les processeurs. Ceci *de facto* limite drastiquement l'extensibilité des applications, le coût de gestion de la MDVP devenant prohibitif.

Aide à la programmation : programmer avec une MDVP ne dispense pas de penser parallèle ! Or penser parallèle est souvent très déroutant pour le programmeur habitué à la programmation séquentielle. Il est donc indispensable de proposer des outils interactifs d'aide à la programmation *parallèle* sous MDVP.

Analyse d'exécution et feedback : la plupart des systèmes ne fournissent à l'utilisateur que des informations très parcellaires sur le comportement effectif des applications. En outre, la structure de certains systèmes (en particulier ceux à base de pages) ne permet pas à l'utilisateur d'agir concrètement sur ce comportement.

L'objet des chapitres qui suivent est de proposer des méthodologies qui permettent d'adresser ces différents points. Le chapitre 5 analyse la structure d'un nouveau modèle de MDVP. Le chapitre 6 présente le système DOSMOS fondé sur ce modèle. Enfin le chapitre 7 décrit l'environnement de programmation parallèle associé à DOSMOS.

CHAPITRE 5

Un nouveau modèle de Mémoire Distribuée Virtuellement Partagée

Ce chapitre décrit les fondements d'un nouveau modèle de Mémoire Distribuée Virtuellement Partagée qui essaie de répondre à certaines des limitations conceptuelles détectées dans les systèmes existants.

Ce modèle est fondé sur une structuration hiérarchique de l'espace des données partagées. Conceptuellement il s'appuie sur une approche modulaire découplant gestion de la MDVP et exécution des applications. Enfin, pour des raisons d'efficacité, ce modèle s'appuie sur l'utilisation de sémantiques de cohérence faible et permet l'intégration de codes programmés en échanges de messages.

5.1 Un modèle modulaire

Ce modèle fait très peu d'hypothèses minimalistes sur l'architecture sous-jacente utilisée et les composantes matérielles qui s'y rattachent :

- Les processeurs doivent disposer d'une mémoire locale et permettre d'utiliser une programmation parallèle de type MIMD ou SPMD.
- Le réseau d'interconnexion permet la communication entre tous les sites présents.
- La mémoire locale d'un processeur doit permettre le stockage de données locales mais aussi le stockage de données virtuellement partagées.

Notre modèle de base est donc constitué d'un ensemble de processus exécutant le code de l'application. Ces processus doivent pouvoir accéder à un ensemble de données partagées de manière transparente (figure 5.1).

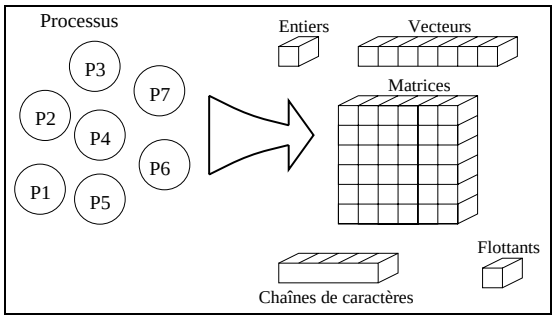


FIG. 5.1 - *Modèle de base : des processus qui exécutent l'application parallèle fondée sur la manipulation de données distribuées virtuellement partagées*

5.2 Structure des objets

Après avoir profondément analysé les différentes modèles de Mémoire Distribuée Virtuellement Partagée, l'utilisation d'une approche "objet" pour concevoir un nouveau modèle nous apparaît plus pertinente. Les objets sont adaptables aux applications qui les utilisent, ils permettent une grande portabilité en ne reposant pas sur des exigences matérielles, ils ouvrent la voie à la conception d'environnement de programmation haut-niveau et permettent une implémentation aisée des systèmes. Notre modèle est donc fondé sur un ensemble d'objets passifs distribués parmi les processus de l'application. Ces objets sont passifs au sens où ils ne contiennent aucune méthode permettant de les manipuler (comme on peut le rencontrer avec une programmation orientée objet (ex: ORCA...)). De plus ces objets ne répondent à aucune spécificité technique autant du point de vue logiciel que matériel.

5.2.1 Différents types d'objets

Les processeurs disposent d'une mémoire locale qui leur est dédiée. Ils peuvent ainsi stocker deux sortes d'objets :

- Les objets locaux : ils sont uniquement connus et instanciés par le processus qui les a déclarés. Ce processus est le seul à pouvoir y accéder ;
- Les objets partagés : ils peuvent éventuellement être présents dans la mémoire locale (copie) mais ils sont aussi connus par d'autres processus et leur contenu peut être temporairement invalide ou inaccessible.

Différents types d'objets sont manipulables, des variables de type simple aux objets complexes.

- Objets simples : ce sont les unités de granularité la plus fine. Ils correspondent aux types de base rencontrés le plus fréquemment dans les langages de programmation : entiers, caractères, réels, booléens...

– Objets complexes : ce sont les structures de données fondées sur l'assemblage d'objets simples. Ils comprennent les chaînes de caractères, les vecteurs, les matrices, les enregistrements, les listes...

5.2.2 Appartenance des objets

Duplication et placement des objets

Notre modèle repose sur la duplication des objets partagés en plusieurs copies (identiques ou non selon la cohérence choisie) distribuées parmi les processus de l'application. La duplication des objets n'est pas forcément totale; elle doit pouvoir être adaptée à l'application. La duplication est ainsi réalisée au fur et à mesure des accès aux objets. Le premier accès à l'objet permet la création de la copie principale. Les accès suivants induisent la création de copies locales chez les processus correspondants. Cette approche garantit un placement optimisé des objets afin de profiter au mieux, lors des accès, de la localité des objets.

Accès aux objets

Qu'un objet soit présent dans la mémoire locale ou distant, il doit pouvoir être accédé de la même manière par un processus. Les protocoles mis en oeuvre pour accéder aux objets doivent être transparents pour l'application et gérés par le modèle. Un processus qui accède à un objet dont il ne possède pas de copie valide (dans sa mémoire locale) se trouve dans une situation de *défaut d'objet*. Le modèle doit alors initier un dialogue entre ce processus demandeur et un processus qui dispose d'une copie valide afin de lui permettre de rapatrier une copie de l'objet.

5.2.3 Des objets complexes adaptables aux applications

L'unité de partage étant basée sur l'objet, l'utilisateur doit pouvoir appliquer des méthodes de découpage pour permettre une utilisation efficace des objets partagés complexes. Nous proposons de permettre un découpage des objets complexes en sous-objets répartis de taille plus réduite, appelés *objets système*.

Ces objets-système sont générés par le système en fonction des choix de l'utilisateur de l'adaptabilité de ses objets à l'application. Comme on peut le voir sur la figure 5.2, l'objet complexe matrice est découpé en quatre objets système. Les objets peuvent éventuellement avoir des tailles différentes si le découpage n'est pas parfait (le dernier objet système contient une ligne de moins que les autres). Ainsi, alors que l'utilisateur n'a déclaré qu'une seule matrice partagée, il dispose, en fait, de nombreux objets systèmes qui peuvent être partagés par différents processus. Bien entendu, ces objets systèmes sont eux-aussi dupliqués parmi les processus qui les manipulent. Ce découpage permet d'adapter la granularité des objets aux contraintes de l'application et du système d'implémentation. Nous verrons que le découpage d'objets permet d'améliorer grandement l'efficacité des applications parallèles qui utilisent notre modèle (expérimentations dans le paragraphe 8.3).

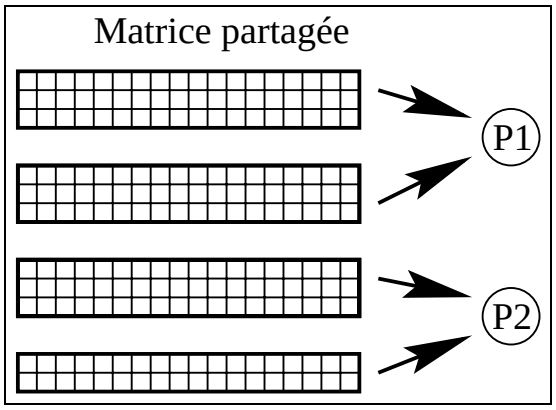


FIG. 5.2 - *Découpage d'une matrice partagée en* objets-système *partagés par différents processus*

Nous avons vu dans le paragraphe 3.4 que des modèles de cohérence forte facilitent la réalisation des systèmes de MDVP et fournissent une certaine sécurité d'accès aux données pour l'utilisateur. Malheureusement, ces modèles ne sont pas suffisants car ils restreignent l'efficacité et les possibilités d'extensibilité des applications. Ainsi pour obtenir de bonnes performances et offrir une adaptabilité aux besoins de l'utilisateur, il est indispensable de proposer différents modèles de cohérence, restrictifs et relâchés.

5.3 Sémantiques de cohérence des données

Pour garantir de bonnes performances, il faut pouvoir dupliquer les objets partagés. Ces duplicas doivent être maintenus cohérents pour les différents processus. Un grand nombre de sémantiques ont été étudiées d'un point de vue théorique (paragraphe 3.4), mais très peu d'entre elles (essentiellement les plus fortes) ont été implémentées. Fondé sur des approches similaires aux modèles à base de multi-cohérences, notre modèle propose l'utilisation de divers types de cohérence, qui nous semblent les plus utilisables du point de vue applicatif. En effet, des systèmes comme Mermera proposent d'innombrables protocoles de cohérence directement inspirés des recherches théoriques qui ont été menées. Mais ces modèles, à l'intérêt théorique évident, peuvent dérouter l'utilisateur de part les subtilités qu'ils impliquent au niveau de la programmation des applications. Notre modèle se restreint donc à trois grands types de cohérence : la non-cohérence, des cohérences fortes et la cohérence à la libération.

- Non-cohérente : la première approche que nous avons suivie est de nous dire que l'utilisateur connaît suffisamment son application pour comprendre les types d'accès que son application va générer.

- Une sémantique de cohérence forte : le facteur qui inquiète (à juste titre) les utilisateurs de MDVP est le délai de mise à jour des données et la validité des données partagées lues. C'est la raison pour laquelle notre modèle intègre des sémantiques de cohérence forte (cohérence stricte et cohérence linéarisable).

– Une sémantique de cohérence faible : la cohérence à la libération [GLL+89]). Ce modèle de cohérence est fondé sur l'utilisation de deux primitives de synchronisation "acquire" et "release" (voir paragraphe 3.4.2). La première permet à un processus d'acquérir un droit d'accès sur une variable en écriture exclusif. Il ne définit pas une section critique au sens propre du terme, c'est-à-dire durant laquelle un seul processus pourrait accéder à cette variable, mais une *section à accès restreint* (SAR) durant laquelle le processus pourra manipuler sans restriction une copie de la variable, sans qu'il soit cependant interdit aux autres processus d'accéder à leur propre copie[1]. Cependant, à l'instar des sections critiques, sur une même variable, une seule section à accès restreint peut être active à un instant donné. L'opérateur "release" termine une section à accès restreint en relâchant la variable qui redevient alors accessible aux autres processus. Cette opération conduit à invalider ou à mettre à jour les autres copies de l'objet. Ce modèle de MDVP s'appuie sur l'utilisation de protocoles de mise à jour adaptés en fonction de la taille des objets. Dans le cas d'objets de petite taille, une mise à jour des copies est réalisée à l'aide de protocoles d'écriture-diffusion dans la mesure où leur coût est sensiblement identique à celui d'une d'invalidation (messages de tailles comparables). A l'inverse, les objets de taille importante sont uniquement invalidés.

  En résumé, on peut donc distinguer 3 types d'accès à une variable:

  – Une écriture effectuée dans une SAR est propagée (lors de l'opération *release*) vers les différentes copies de la variable ;
  – Une lecture effectuée dans une SAR renvoie la dernière valeur écrite dans une SAR ;
  – Une lecture effectuée en dehors d'une SAR ne renvoie pas forcément la dernière valeur écrite.

  Les opérateurs *acquire* et *release*, outre la *cohérence à la libération*, permettent de modéliser très simplement deux sémantiques de cohérence fortes: la cohérence séquentielle et la cohérence stricte [GLL+89]. En effet, il suffit d'effectuer toutes les écritures dans des sections à accès restreint pour garantir une exécution séquentiellement cohérente. En imposant en outre que l'ensemble des lectures soient réalisées dans des sections à accès restreint, on garantit alors *de facto* une cohérence d'exécution stricte puisque tous les accès se font dans ce cas en section critique.

5.4 Structuration hiérarchique de la programmation parallèle

Comme nous l'avons vu précédemment, les systèmes de Mémoire Distribuée Virtuellement Partagée proposent des modèles "*à plat*" de gestion de la mémoire partagée. Tous les processus peuvent ainsi accéder de manière complètement anarchique à toutes les données partagées. Une telle approche ne peut clairement pas offrir un modèle extensible aux applications. Il suffit d'imaginer le coût du maintien de la cohérence de 100 données partagées par 100 processus pour s'en rendre

1. ...à leurs risques et périls ! En effet, les éventuelles modifications que ces processus seraient amenés à faire seront invalidées à l'issue de la fin de la SAR (*acquire*).

compte ! Afin de limiter ces coûts et de proposer d'autres approches pour le partage de données distribuées nous proposons d'introduire la notion de *hiérarchies de groupes de processus.*

Si on observe les résultats d'exécution d'une application standard qui utilise une MDVP. On se rend compte que les processus accèdent à différents types d'objets. Une application est donc fondée sur un ensemble d'objet locaux et d'objets partagés accédés par des processus. Ces processus peuvent être amenés à utiliser certains objets de manière intensive, afin de sauvegarder leurs résultats. De plus, la gestion de ces objets doit permettre le respect de la sémantique de cohérence choisie par l'utilisateur. D'un autre côté, les processus accèdent aussi à des objets de manière exceptionnelle et n'ont pas réellement besoin d'avoir, à tout instant, une copie valide de ces objets. En prenant en compte ces besoins applicatifs, nous proposons d'introduire la notion de structuration hiérarchique de la programmation parallèle.

5.4.1 Domaines d'objets

Définition 5.1 *On appelle* domaine *d'un objet, l'ensemble des processus qui partagent cet objet.*

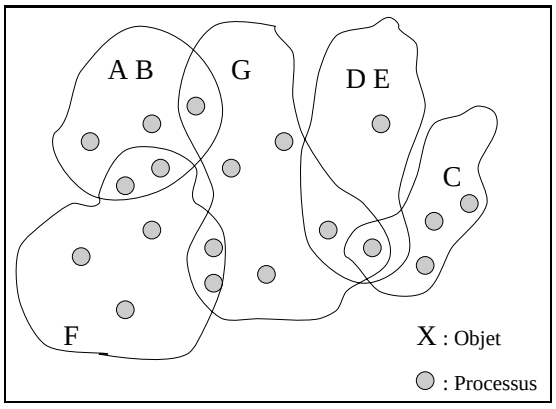


FIG. 5.3 - *Domaines d'objets*

Le découpage fonctionnel, à l'aide de domaines, assure le fait que les processus d'un domaine n'auront pas besoin, au cours de l'application, d'accéder à d'autres objets. La difficulté réside dans la distribution des objets dont les domaines sont fortement intersectés. Il en résulte des problèmes de placement des processus qui appartiennent à plusieurs domaines. Par exemple dans la figure 5.3, on voit que le processus situé à l'intersection des domaines C, DE et G devra être parfaitement placé dans le réseau car ils accède à trois domaines différents au cours de l'application. Difficilement utilisable et offrant peu d'attraits à l'implémentation, la découpage fonctionnel fondé sur les domaines est difficilement envisageable. Nous proposons donc un modèle qui permette un regroupement des processus en fonction de leurs réels besoins d'accès aux objets pendant l'exécution.

5.4.2 Groupes de processus

On peut structurer l'ensemble des objets accédés par un processus P de la manière suivante :

- L'Espace des Objets Accédés (E.O.A.) est l'ensemble des objets locaux ou partagés accédés par le processus durant son exécution ;
- L'Espace Virtuel Global (E.V.G.) contient les objets partagés accédés (en lecture ou écriture) par P durant son exécution ;
- L'Espace Non Partagé (E.N.P.) contient les objets locaux manipulés exclusivement par le processus P.

On a donc :

$$E.O.A.(P) = E.V.G.(P) + E.N.P.(P)$$

L'idéal est bien entendu de transférer un maximum d'objets présents dans l'EVG du processus dans son Espace Non Partagé afin d'améliorer les accès aux données. Mais cela n'est pas toujours possible, sinon le partage de données n'aurait plus de raison d'être ! Analysons donc plus finement l'Espace Virtuel Global (figure 5.4) qui peut être défini par :

$$Espace\ Virtuel\ Global(P) = Espace\ Virtuel\ Local(P) + Espace\ Virtuel\ Externe(P)$$

avec

- *L'Espace Virtuel Local (E.V.L.) contient l'ensemble des objets partagés accédés de manière intensive par le processus.*
- *L'Espace Virtuel Externe (E.V.E.) contient les objets rarement accédés et qui nécessitent, à ce titre, peu d'invalidations.*

Le principal inconvénient des systèmes *à plat* vient du fait que lorsqu'un objet O est modifié, un message d'invalidation est envoyé à tous les processus i tels que $O \in EVG(i)$. Pire, dans certains systèmes comme ORCA[TKB92], la nouvelle valeur de l'objet est aussi envoyée aux processus j tels que $O \notin EVG(j)$. Ainsi même si un processus n'accède pas à un objet, il reçoit l'ensemble des messages d'invalidations qui concernent cet objet. Nous proposons, au contraire, de restreindre les invalidations aux seuls processus i tels que $O \in EVL(i)$.

L'idée consiste à structurer l'application en tâches exécutées par des groupes de processus partageant les mêmes objets.

Définition 5.2 *Un groupe de processus est un ensemble de processus qui partagent un ensemble d'objets. On note* Groupe(O) *le groupe gérant l'objet* O. *Un objet n'est géré que par un seul groupe.*

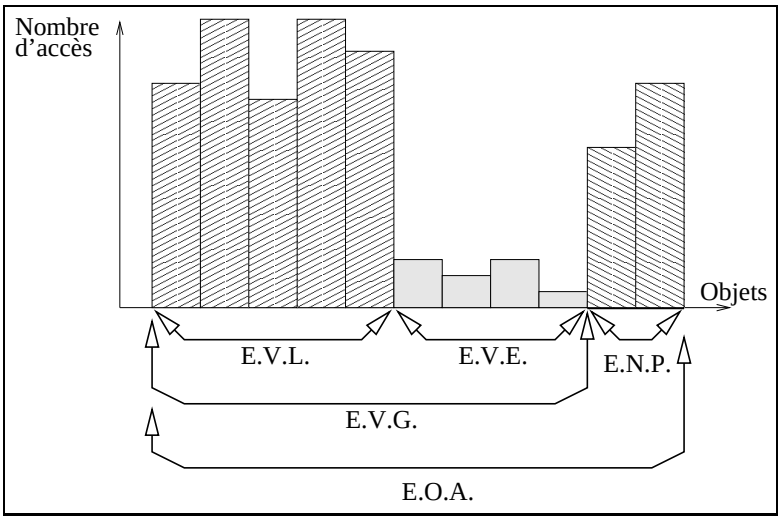


FIG. 5.4 - *Distribution des accès pour un processus donné*

Définition 5.3 *Abonnement à un groupe : un processus i est dit abonné à Groupe(O) si et seulement si* $O \in E.V.L.(i)$.

Un groupe rassemble donc un ensemble de processus qui partagent les mêmes objets et y accèdent fréquemment. Ainsi, un groupe peut servir de base de partage à plusieurs objets (X et Y sont partagés par les mêmes processus dans la figure 5.5). Les processus sont abonnés à ces groupes pour toute la durée de l'exécution. La gestion des objets du groupe est assurée par le groupe lui-même de manière décentralisée et transparente.

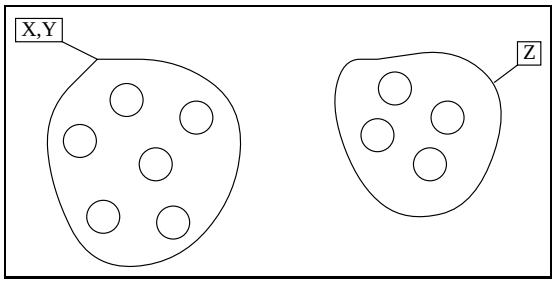


FIG. 5.5 - *Des groupes de processus*

Un problème peut intervenir dans le cas d'objets partagés intensivement par un vaste ensemble de processus. Pour prendre en compte ces objets *globalement* partagés, on peut avoir tendance à agrandir les groupes et, à la limite, on risque d'aboutir à un partage *à plat* (où tous les processus font partie d'un groupe unique contenant l'ensemble des objets). Le concept de groupes doit donc intégrer un aspect hiérarchique afin de permettre un partage adapté des objets à tous les niveaux. La figure 5.6 propose un exemple de structuration hiérarchique. Dans cet exemple, tous les processus partagent l'objet G qui se trouve dans le groupe de plus bas niveau. On peut voir, aussi, deux sous-ensembles de processus qui partagent l'un les objets D et E l'autre A et B. Enfin, seuls quelques processus des niveaux supérieurs partagent l'un C l'autre A et B. Ainsi, un processus qui est abonné au groupe qui contient l'objet C partage automatiquement (par descendance hiérarchique) les objets D, E et G. Cette notion de groupes hiérarchiques permet ainsi d'éviter les communications globales nécessaires au maintien de la cohérence des objets. Nous verrons aussi que les groupes hiérarchiques ont une influence positive sur la programmation parallèle et qu'ils permettent une bonne adéquation aux applications massivement parallèles ou hétérogènes (paragraphe 5.7).

Ce modèle propose ainsi trois types d'accès aux données.

Accès intra-groupes

Les accès à l'objet à l'intérieur du groupe de processus dont il dépend sont complètement localisés et le coût de maintien de la cohérence est fortement réduit. Prenons l'exemple de la figure 5.6. Si le processus P du groupe de C modifie l'objet C, le système génère quelques communications seulement. Plus précisément, il envoie un message d'invalidation aux (seuls) autres processus membres du groupe qui ont une copie de l'objet. Ainsi, la diffusion de l'invalidation est restreinte aux quatre autres processus qui partagent réellement C au lieu d'atteindre tous les processus qui auraient occasionnellement accédé à C.

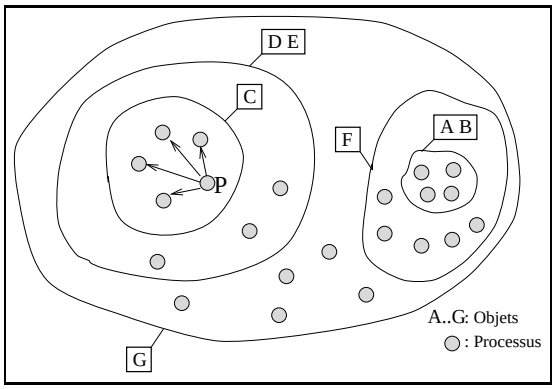


FIG. 5.6 - *Groupes hiérarchiques de processus: accès intra-groupes*

Accès inter-groupes

Avec cette structure hiérarchique, les processus peuvent accéder à des objets présents dans des groupes englobants. Ainsi dans la figure 5.7, tous les objets peuvent modifier et lire l'objet G puisqu'il est partagé par un groupe contenant l'ensemble des processus de l'application. De même (figure 5.7), une modification de l'objet D provoque la diffusion de messages d'invalidation à tous les processus qui partagent D, du plus bas niveau hiérarchique au plus élevé.

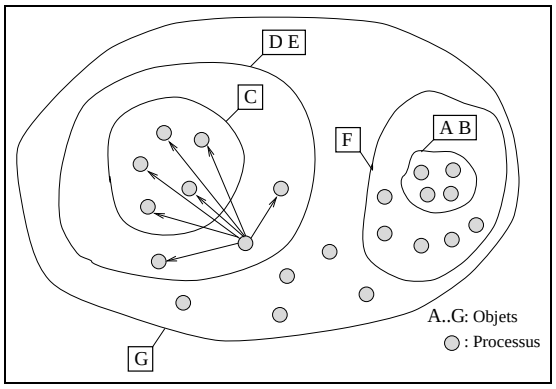


FIG. 5.7 - *Groupes hiérarchiques de processus: accès inter-groupes*

Accès extra-groupes

Mais nous avons vu dans les définitions précédentes, que les processus manipulent aussi des objets de manière plus exceptionnelle. Ainsi, pour accéder à l'Espace Virtuel Externe des processus, il est nécessaire d'implémenter la possibilité d'accès extra-groupes. Dans un accès extra-groupe, la valeur de l'objet retournée est correcte à cet instant donné mais la cohérence de cette valeur n'est ensuite plus maintenue par le modèle. La figure 5.8 présente ainsi le cas d'un processus qui partage habituellement les objets A et B (F et L) et qui doit accéder à l'objet C partagé par un autre groupe de processus. Il envoie donc une demande au processus propriétaire de l'objet C. Si cet accès est une écriture, la requête contient la nouvelle valeur à insérer dans la l'objet C. Si c'est une simple lecture, le processus propriétaire de C renvoie une valeur valide de l'objet, mais la validité de cette copie ne sera pas assurée par la suite. Le processus propriétaire de C ne garde d'ailleurs aucune trace de l'envoi de la copie à l'extérieur du groupe.

Tous les protocoles pour l'implémentation des protocoles de gestion des groupes sont décrits dans le paragraphe 6.6.

5.4.3 Groupes adaptables

Pour être efficace, ce modèle nécessite une connaissance correcte de l'application parallèle de la part de l'utilisateur. Celui-ci doit avoir une idée assez précise du comportement de l'application vis-à-vis des objets partagés. Ainsi, pour éviter des diffusions d'invalidations à tous les processus

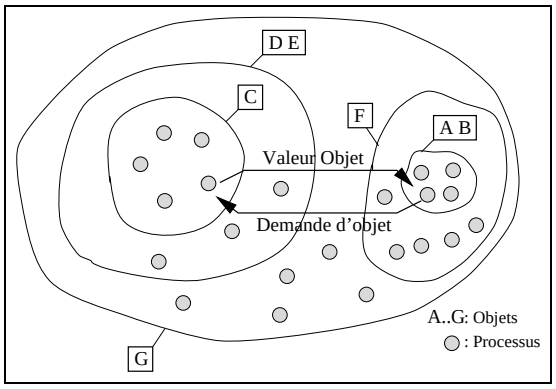


FIG. 5.8 - *Groupes hiérarchiques de processus: lecture extra-groupe*

d'un groupe, même si ceux-ci n'en partagent pas encore les objets, un processus n'est réellement abonné à un groupe que lorsqu'il a accédé, au moins une fois, à l'objet. De même, un processus signale son abonnement à un groupe en envoyant un message aux membres du groupe en même temps que la première invalidation qu'il réalise sur l'objet.

Cette notion de groupes hiérarchiques permet de s'adapter à de nombreux types d'applications. Étudions les cas d'utilisation extrêmes de groupes hiérarchiques.

Non-utilisation des groupes

Dans sa forme la plus restrictive, ce modèle peut être utilisé avec des groupes de processus limité à un seul élément. Seul le processus qui manipule l'objet est habilité à le partager. Cette utilisation bénéficie à la fois de la robustesse des anciens systèmes qui ne fournissaient pas de mécanismes de cohérence mais elle propose aussi de nouvelles sémantiques d'accès.

Prenons l'exemple numérique classique de la *Multiplication de Matrices*. Sa réalisation nécessite l'utilisation de trois objets partagés, deux matrices de données en entrée et une matrice de résultats. Supposons que les deux matrices d'entrée sont découpées et partagées l'une en lignes et l'autre en colonnes. La matrice résultat est découpée en colonnes.

Ces trois objets manipulés nécessitent un partage effectif entre les différents processus. Chaque processus accède à une partie de chaque objet. L'utilisation des groupes dans ce cas, tire bénéfice du faible besoin de cohérence des différents sous-objets. Chaque processus dispose d'un accès exclusif en lecture (pour les matrices en entrée) ou d'un accès exclusif en écriture (pour la matrice résultat). On pourrait alors s'interroger sur l'utilité d'une MDVP dans un tel exemple. Mais le partage est, ici, indispensable et facilement utilisable. En effet, après le calcul du résultat, une phase de synchronisation a lieu et tous les processus peuvent lire la matrice partagée à l'aide d'accès extra-groupes. Dans un cas comme la multiplication de matrices, on aboutit donc à une utilisation optimale d'un modèle de Mémoire Distribuée Virtuellement Partagée. Il n'y a pas de génération de communications superflues. Seules les communications nécessaires à l'accès aux objets partagés sont générées par le système. On est alors dans une sémantique d'accès aux objets partagés très

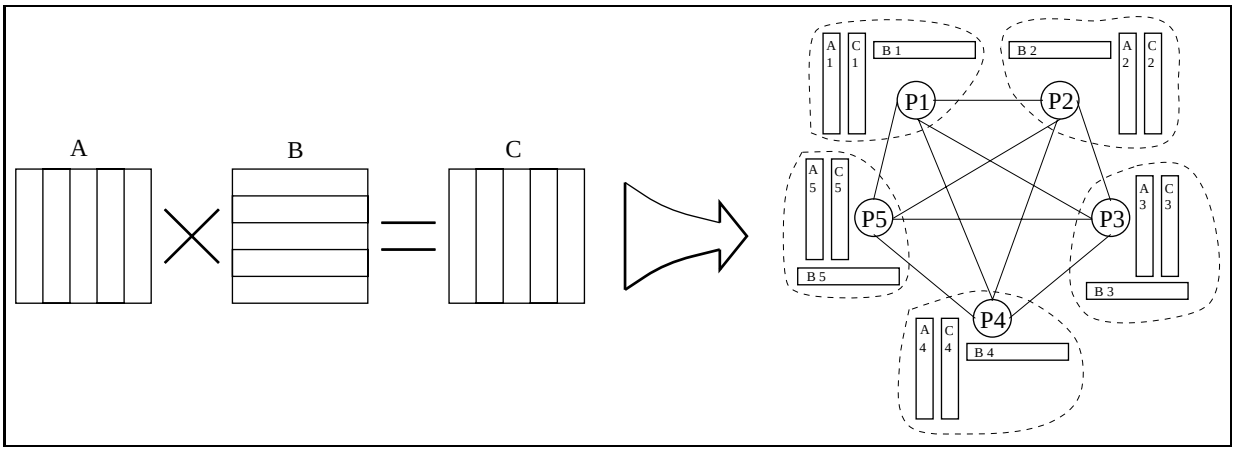


FIG. 5.9 - *Multiplication de matrices à l'aide de cinq processus sans utilisation de groupes hiérarchiques*

proche de l'échange de messages.

Groupes complets

Lors d'une première utilisation de notre modèle, l'utilisateur peut se satisfaire de la création d'un groupe unique contenant l'ensemble des processus de l'application. Avec cette configuration, tous les processus peuvent accéder à tous les objets du groupe. Le partage est total et le modèle ne tire aucun avantage de l'utilisation des Espaces Virtuels Locaux. Dès qu'un processus accède à un objet même présent dans son Espace Virtuel Externe (i.e. faiblement accédé), il devient abonné à cet objet et reçoit tous les messages ultérieurs nécessaires à la gestion de la cohérence (invalidations, mises à jour). C'est donc une configuration de partage à plat des objets très proche de l'utilisation des *listes de copies* (*copy sets*) Cela nous amène à une comparaison de notre modèle hiérarchique avec cette technique couramment utilisée dans les autres modèles existants.

5.4.4 Groupes et listes de copies

La plupart des modèles de Mémoire Distribuée Virtuellement Partagée disponibles actuellement proposent une gestion des copies fondée sur les *copy sets* proposés par Kai Li pour son système IVY [Li86] (voir paragraphe 4.1). Nous allons donc rappeler dans un premier temps les principes des listes de copies puis nous comparerons cette approche avec la notion de groupes de processus.

A l'instar des groupes de processus, les listes de copies sont un moyen de limiter les coûts de gestion de la cohérence des données. Certains modèles (sans *copy sets*) étaient fondés sur une diffusion totale des messages de cohérence (invalidation, recherche de l'objet...). Ainsi, un processus qui n'accédait pas aux données partagées recevait néanmoins tous les messages d'invalidations concernant cette donnée. A l'inverse, le *copy set* associe à chaque donnée partagée une structure de donnée dédiée à la sauvegarde des identificateurs de processus qui disposent d'une copie de la

donnée (figure 5.10). L'identificateur de chaque processus qui partage une donnée se retrouve inscrit sur la liste de copies dès le premier accès à la donnée. Les modèles basés sur des listes de copies proposent de limiter la diffusion des messages aux seuls processus présents dans cette liste.

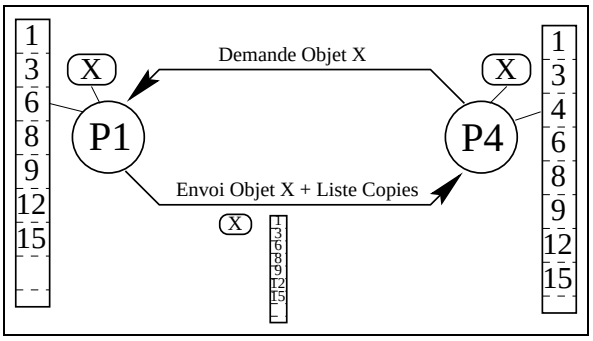


FIG. 5.10 - *Gestion de* copy sets *(K. Li)*

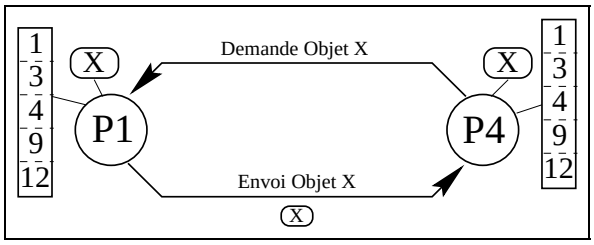


FIG. 5.11 - *Gestion des groupes de processus*

La structuration hiérarchique en groupes propose une autre modélisation des accès aux données partagées et permet de réaliser différentes améliorations théoriques et pratiques :

- **Réduction de la taille des communications** : si le modèle utilise la notion de propriétaire dynamique ou momentanément mobile (paragraphe 3.6.2), le déplacement de la copie principale entraîne un déplacement de toute la structure de données accompagnant la donnée. Ainsi la liste des copies doit être communiquée en même temps que la donnée partagée (figure 5.10). A l'inverse, lors de l'utilisation des groupes, les processus abonnés dans un groupe connaissent l'ensemble de leurs voisins. L'abonnement à un groupe a lieu lors du premier accès à l'objet. Comme les groupes sont fixés dès le début de l'exécution, le maintien de ces groupes ne nécessite donc aucune communication par la suite.

- **Réduction du nombre de communications** : un processus qui accède à des données partagées de manière exceptionnelle se retrouve automatiquement inscrit sur la liste des copies. Ainsi, par la suite, toute modification de la donnée entraîne l'envoi d'un message d'invalidation à ce processus même si celui-ci n'est plus intéressé par cette donnée. Les groupes hiérarchiques permettent une réduction du nombre de communications en n'envoyant des messages d'invalidations qu'aux processus qui ont fait la demande explicite d'être dans le groupe de partage.

- **Gestion des données partagées simplifiée** : les listes de copies d'objets doivent être maintenues et dupliquées chez l'ensemble des processus qui disposent d'une copie de l'objet.

  Une liste des copies associée à un objet contient l'ensemble des identificateurs de processus dont l'objet appartient à l'Espace Virtuel Global. L'espace mémoire nécessaire au stockage de la liste de copies n'est pas précisément déterminable a priori. La taille doit être adaptée à la sauvegarde de l'ensemble des identificateurs de processus du réseau. La structure de donnée associée à chaque objet partagée peut être lourde à gérer.

  Les groupes de processus étant fixés par l'utilisateur, leur taille est connue a priori. De plus, les groupes de processus se contentent d'un espace mémoire dont la taille est liée au nombre de

processus du groupe. Les groupes permettent donc de réduire la taille mémoire nécessaire à la gestion des copies. Enfin, dans le cas des modèles de MDVP à base de pages, il existe un grand nombre de pages partagées. L'utilisation de listes de copies peut ainsi s'avérer difficile avec de tels modèles puisque chaque page doit être associée à un *copy set* de taille éventuellement importante. Les groupes de processus peuvent donc s'employer parfaitement avec des modèles à base de pages. La structure de données associée à chaque page est alors réduite et fixée par l'utilisateur.

- **Synchronisation de processus** : les groupes de processus permettent un regroupement virtuel mais aussi spatial pour l'implémentation de la synchronisation de processus ayant le même comportement. Cela permet une implémentation de protocoles de synchronisation décentralisés et localisés sur un faible nombre de processus. Cette caractéristique est très appréciée actuellement pour la conception de barrières de synchronisation dédiées à des architectures à base de grappes de machines (voir [SH95]).

- **Programmation MIMD** : dans le cadre d'une programmation SPMD, les processus peuvent avoir des comportement assez proches. Mais l'utilisation d'une conception MIMD peut conduire à spécialiser les processus dans des fonction très différentes. Par exemple, alors qu'un ensemble de processus accède aux objets exclusivement en écriture, d'autres peuvent avoir une fonction de lecteur des objets partagés (type client-serveur). Les listes de copies sont peu adaptées à la programmation MIMD car elles considèrent que tous les processus ont un comportement similaire et régulier (tous les processus accèdent aux mêmes objets et de la même manière). Les groupes de processus permettent une utilisation adaptée, en plaçant les processus à différents niveaux dans la hiérarchie ou en les excluant de certains groupes. Ce modèle ouvre notamment la voie à une utilisation orientée vers des modèles coopératifs (paragraphe 3.9.2).

- **Maîtrise par l'utilisateur** : la définition et l'utilisation des groupes de processus imposent un effort supplémentaire à l'utilisateur. Mais celui-ci peut ainsi parfaitement maîtriser le placement et la duplication de ses objets. A l'inverse, s'il désire bénéficier d'une transparence totale, il lui est aisé d'utiliser un modèle de partage à base de groupes complets qui lui apportent un comportement d'exécution identique à une application utilisant les listes de copies.

En résumé, la structuration hiérarchique à l'aide de groupes de processus peut être vue comme une généralisation des modèles de listes de copies de données.

5.4.5 Le compromis des groupes dynamiques

Dans ce modèle, tel qu'il a été défini jusqu'ici, un groupe est créé au moment de la compilation en accord avec les directives de l'utilisateur. Ceci suppose que l'utilisateur ait une vision précise de son application. Réciproquement, une mauvaise spécification peut conduire à des comportements très peu efficaces voire catastrophiques en raison du coût élevé des accès extra-groupes.

La prise en compte de telles situations nécessite d'inclure dans le modèle des fonctionnalités d'adaptation dynamiques. Quatre principales stratégies sont envisageables :

- **Abonnement à un objet partagé externe** : supposons qu'un processus accède fréquemment à un objet géré par un autre groupe que le sien. Il peut être, alors, intéressant d'abonner dynamiquement ce processus à cet objet. Le processus dispose alors d'une copie de l'objet maintenue à jour au même titre que les copies réparties au sein du groupe dont dépend l'objet. D'un point de vue formel, ceci consiste à ajouter l'objet dans l'EVL du processus.
- **Abonnement à un groupe** : si un processus accède fréquemment à la plupart des objets gérés par un groupe externe (i.e. non hiérarchiquement dépendant[2]), tout en continuant à accéder aux objets gérés par son groupe, il peut être pertinent d'abonner dynamiquement ce processus à ce groupe externe. Cela revient à abonner le processus à l'ensemble des objets gérés par le groupe.
- **Déplacement hiérarchique** : si un processus accède fréquemment à la plupart des objets gérés par un groupe hiérarchiquement dépendant, on a intérêt à déplacer ce processus vers ce dernier groupe. Cela se traduit par l'inclusion dans l'EVL du processus des objets gérés par le groupe dépendant.
- **Déplacement vers un groupe indépendant** : de même, si un processus accède fréquemment à des objets gérés par un groupe externe non-hiérarchiquement dépendant, on peut avoir intérêt à déplacer ce processus vers ce groupe. Cela se traduit par un désabonne ment aux objets gérés par le groupe initial suivi d'un abonnement aux objets gérés par le groupe destination.

L'intérêt de ces différents stratégies est de minimiser le nombre des accès extra-groupes dont on verra (paragraphe 6.6) qu'ils nécessitent une mise en oeuvre complexe et coûteuse.

Ces stratégies reposent toutes sur la notion de fréquence d'accès. Définir le seuil à partir duquel on a intérêt à abonner ou à déplacer un processus dépend de nombreux paramètres: volume des communications, coûts, saturation du réseau d'interconnexion, caractéristiques de l'application...Une politique laxiste d'abonnements multiples peut ainsi conduire à une dégénérescence de fait du modèle de groupes en un modèle de type gestion de *copy sets*.

L'idéal serait de tendre vers un modèle mixte dans lequel les abonnements ne remettent pas en cause les bénéfices apportés par la notion de groupes.

Dernier point : la mise en oeuvre de telles stratégies dynamiques peut parfois dérouter l'utilisateur qui voit varier les performances de ses applications au gré des abonnements et déplacements de processus. Il est donc indispensable de garder une trace des abonnements et déplacements afin de permettre à l'utilisateur d'analyser le comportement de son application et éventuellement d'adapter sa structuration hiérarchique.

2. Un groupe, noté A, est dit hiérarchiquement dépendant par rapport à un autre, noté B, lorsque ces deux groupes appartiennent à la même hiérarchie et que le premier est situé à un niveau inférieur. En d'autres termes, *EVL(ensemble des processus du groupe B)* $\subset$ *EVL(ensemble des processus du groupe A)*. Ainsi figure 5.6, le groupe gérant l'objet C est hiérarchiquement dépendant du groupe gérant D et E.

5.5 Groupes hiérarchiques et sémantiques de cohérence

La notion de groupes de processus influence fortement l'implémentation et le concept des sémantiques de cohérence. Ainsi si certains systèmes comme MERMERA ou MUNIN proposent différentes sémantiques de cohérence, une donnée partagée est cependant toujours gérée selon une seule sémantique. Ce principe est remis en cause par la notion de groupes. Les accès extra-groupe à un objet sont gérés de facto selon une sémantique de cohérence stricte (chaque accès génère une requête au propriétaire de l'objet), alors que les accès intra-groupes peuvent répondre à une sémantique beaucoup plus faible. Cette ambivalence est cohérente avec le modèle de programmation implicitement suggéré dans ce modèle, fondé sur un découpage de l'application en tâches réalisées par des groupes de processus. C'est le rôle du programmeur d'intégrer cette caractéristique dans son application.

A l'intérieur d'un groupe toute sémantique de cohérence est envisageable et des sémantiques différentes peuvent être associées à des objets différents. Par contre, comme on l'a vu, à l'extérieur du groupe, les accès se font selon une sémantique stricte. Ainsi modifier la structure des groupes affecte la sémantique de cohérence associée aux objets partagés sans réécriture du code de l'application.

Dernière remarque : il n'est pas inintéressant de rapprocher les notions de groupes de processus et de cohérence à la libération fainéante telle qu'elle est utilisée dans le système TREADMARKS (paragraphe 4.1). Dans cette approche, l'invalidation est limitée au prochain processus qui va avoir besoin de la donnée partagée. Dans le concept de groupes, l'invalidation concerne l'ensemble des processus faisant partie du groupe (c'est à dire les processus censés avoir fréquemment, donc prochainement, besoin de cette donnée).

5.6 Mélange de modèles de programmation

Il serait illusoire d'affirmer qu'un modèle de mémoire distribuée virtuellement partagée s'adapte parfaitement à tous les types d'applications. Utiliser de l'échange de messages dans une application parallèle permet aussi d'atteindre de bonnes performances; même si cela se fait souvent au détriment de la simplicité de programmation. Il nous semble indispensable d'intégrer dans notre modèle la possibilité de mélanger ces deux modèles de programmation. L'utilisateur peut ainsi disposer d'un accès transparent aux objets partagés et de primitives d'accès explicites fondées sur l'échange de messages. Ceci offre une modélisation des applications adaptable à tous les types d'utilisateurs, du débutant qui ne connaît aucun modèle de programmation, jusqu'à l'expert en échange de messages. Utiliser une telle interconnexion des modèles de programmation préfigure de nouveaux moyens pour la parallélisation d'applications. L'utilisateur habitué à un seul modèle de programmation continue ainsi à l'utiliser pour profiter de l'adéquation à l'application apportée par le modèle. avec notre modèle. Mais, lors d'opérations de ré-engineering par exemple, il peut mélanger les deux modèles de programmation pour obtenir des performances accrues.

5.7 Un modèle adaptable

Ce modèle de programmation original permet de prendre en compte un grand nombre d'architectures et d'applications complexes et diversifiées.

5.7.1 Adaptation aux applications parallèles

Ce modèle de Mémoire Distribuée Virtuellement Partagée est basé sur une fusion de méthodes existantes et de propositions originales. Ainsi, le mélange de modèles de cohérence faible avec différents modèles de programmation permet une adaptation à une vaste gamme d'applications (applications de type client-serveur avec beaucoup d'accès en lecture et peu en écriture, applications utilisant l'échange de messages et qui désirent accéder facilement à des objets partagés...). De plus, le découpage personnalisé des données permet un accès et une disponibilité des objets propres aux besoin spécifiques de certaines applications (calcul numérique à base de matrices de taille importante, imagerie avec découpage des scènes...). Enfin la connexion avec une programmation structurelle à base de groupes de processus permet d'optimiser des applications spécifiques (bases de données parallèles où chaque groupe de processus gère un cluster de données, applications tolérantes aux fautes fondées sur une redondance logicielle...).

5.7.2 Adaptation à l'architecture-cible : du distribué au massivement parallèle

Ce modèle est aussi adapté à une très vaste variété d'architectures distribuées (réseaux locaux et réseaux longue distance) ou parallèles (machines à base de clusters et machines massivement parallèles).

Réseau de stations

La principale problématique des modèles de Mémoire Distribuée Virtuellement Partagée destinés à des architecture réparties réside dans la lourdeur des coûts de gestion de la cohérence entre les tâches de l'application. Tous les sites du réseau communiquent avec le même médium (Ethernet) qui est lent et peu fiable. Il faut donc contourner au maximum la faiblesse de la bande passante et la grande latence du réseau en limitant les communications entre les stations de travail. La cohérence faible et les groupes de processus permettent de limiter ces communications coûteuses en isolant les processus d'un même groupe sur un nœud du réseau. De plus, le découpage à la demande des gros objets partagés permet une adaptation aux caractéristiques particulières des réseaux de stations. L'envoi d'un petit message et ainsi presque aussi coûteux que l'envoi d'un message de taille plus importante à cause de la grande latence du réseau. En choisissant un découpage approprié, l'utilisateur peut regrouper ses communications en envoyant des blocs de données de taille optimale. De cette manière, notre modèle propose une bonne adaptation aux contraintes posées par les systèmes de Mémoire Partagée Réseau [OMW+92] qui évitent toute gestion centrale de la cohérence des données partagées et limitent les diffusions globales qui détruisent toute possibilité d'extensibilité.

Réseau grande distance

Jusqu'à présent, les systèmes permettant des implémentations sur des réseaux longue distance sont tous fondés sur de la programmation à base d'échange de messages. Les développeurs ne veulent pas utiliser la mémoire distribuée virtuellement partagée dans ce cadre car les accès distants aux données sont difficilement prévisibles et le coût de maintien de la cohérence entre les sites distants est très élevé. La programmation à base d'échange de messages nécessitant le codage complet de toutes les communications, les développeurs peuvent ainsi parfaitement contrôler leurs applications. Notre modèle à base de cohérence faible et de structuration hiérarchique des accès aux données pourrait permettre d'intéressantes implémentations proches de celles profitant de l'échange de messages. Ainsi, la figure 5.12 nous présente le cas d'un réseau longue distance avec trois sites principaux. Dans un tel contexte, notre modèle peut offrir la possibilité de déclarer des groupes de processus indépendants sur chaque site principal. Les communications les plus coûteuses sont celles qui auront lieu entre les sites principaux. Elles se feront donc via l'utilisation d'accès extra-groupes ou d'échanges de messages. Les coûts de maintien de la cohérence des objets seront localisés sur chacun des sites.

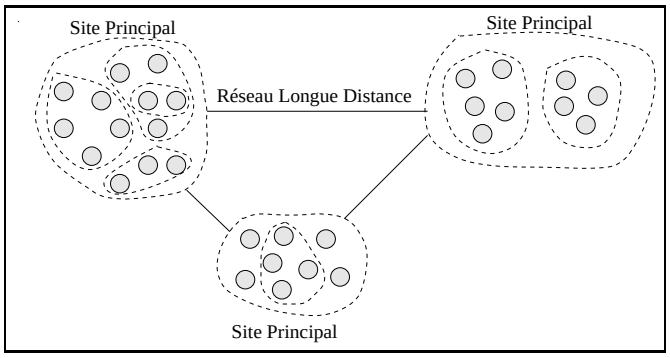


FIG. 5.12 - *Des groupes hiérarchiques de processus sur des configurations de réseaux longue distance*

Machines à base de grappes de processeurs

D'une manière identique à l'utilisation de réseaux longue distance, les architectures parallèles à base de grappes de processeurs (*clusters*) peuvent bénéficier des apports de notre modèle. La figure 5.13 décrit un exemple de machine parallèle à base de clusters : la machine MATRA CAPITAN [MAT95]. Elle se présente sous la forme d'un anneau de communications à grand débit reliant des hypernœuds. Chaque hypernœud est constitué d'un ensemble de processeurs reliés par un bus de communication. Ainsi, une application peut (doit) bénéficier de la localité des communications en localisant au maximum les communications à l'intérieur des hypernœuds et en évitant l'accès à l'anneau de communications inter-hypernœuds. Dans ce cadre, l'exploitation de la localité spatiale peut être mise en œuvre par l'intermédiaire de groupes de processus localisés sur les processeurs des hypernœuds de la machine.

Un découpage approprié des objets combinée à une utilisation des groupes hiérarchiques permet une distribution des sous-objets transparente et optimisée entre les différents *clusters* de la machine.

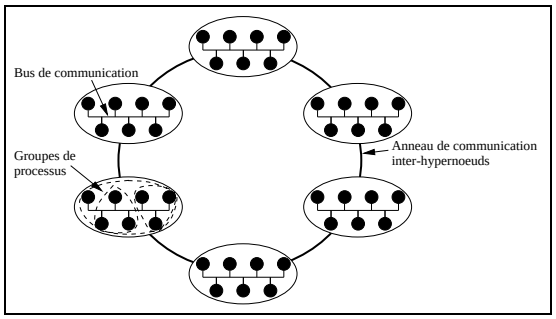


FIG. 5.13 - *Groupes hiérarchiques de processus sur des machines à base de grappes de processeurs : la machine CAPITAN*

Machines massivement parallèles

Deux types de machines massivement parallèles coexistent : celles dont la conception (le réseau d'interconnexion, mécanismes de routage) ne tire aucun avantage de la proximité des processus et celles qui sont sensibles à la localité des communications entre les processeurs. Dans le premier cas, notre modèle hiérarchique peut permettre une réduction notable du nombre de communications globales nécessaires à la gestion de la cohérence. Les groupes de processus offrent, en outre, la possibilité de tirer bénéfice de la proximité des processus présents dans un même groupe. Ainsi, les communications globales peuvent être réduites et localisées. La difficulté réside dans le placement des groupes sur les architectures utilisées (figure 5.14). Un environnement de programmation fondé sur ce modèle devra aider l'utilisateur dans cette tâche (voir paragraphe 7.2.3).

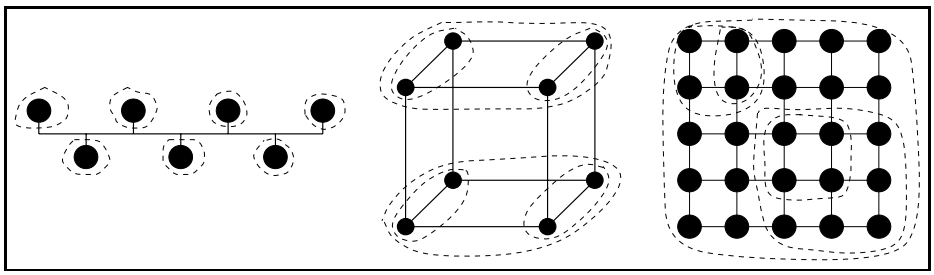


FIG. 5.14 - *Groupes hiérarchiques de processus mappés sur différentes topologies : grille, hypercube*

5.8 Conclusion

Nous avons essayé de jeter les bases d'un nouveau modèle de Mémoire Distribuée Virtuellement Partagée. Ce modèle définit un espace d'*objets* partagés. En effet, gérer des objets plutôt que des pages offre, comme nous le verrons paragraphe 7 et 8, offre une plus grande souplesse de programmation et une meilleure potentialité d'adaption aux spécificités des applications.

Grâce à des techniques de découpage des objets partagés complexes ce modèle garantit une meilleure disponibilité des données ainsi qu'une plus grande efficacité (minimisation des attentes, réduction du volume des communications et de l'importance des goulots d'étranglement).

Ce modèle s'appuie sur une structuration des processus en groupes hiérarchiques. Cette notion nouvelle qui généralise et étend la notion de *copy sets* présente de nombreux avantages : efficacité et extensibilité grâce à la réduction du coût du maintien de la cohérence des données; adaptabilité à une vaste gamme d'architectures allant des systèmes répartis aux machines massivement parallèles; ouverture à un large éventail d'applications grâce à un modèle de programmation fondé sur un découpage fonctionnel en tâches. Cette approche a été récemment reprise dans d'autres systèmes en vue d'améliorer leurs performances [AM96, CCGST96].

En conclusion, par rapport aux travaux existant, ce modèle introduit de nouvelles fonctionnalités de programmation. Rien n'oblige, cependant, le programmeur à les utiliser ! Le découpage d'objets et la structuration en groupes ne peuvent néanmoins qu'améliorer, on le verra, les performances et l'extensibilité de ses applications (paragraphe 8).

Le chapitre suivant propose une analyse du système DOSMOS qui implémente l'ensemble des concepts introduits dans ce modèle. Enfin, tirer le meilleur parti des fonctionnalités de ce système passe par une bonne connaissance de l'application en tant qu'application *parallèle*. Cela justifie la définition et l'implémentation d'un véritable environnement de développement à même de fournir à l'utilisateur une interface souple et conviviale lui permettant de structurer l'application et d'analyser son comportement (paragraphe 7).

CHAPITRE 6

Le système DOSMOS

Dans le chapitre précédent, nous avons introduit les bases conceptuelles d'un nouveau modèle de MDVP. Le système DOSMOS [1] (DOSMOS : Distributed Objets Shared MemOry System, c'est à dire Système de Mémoire Partagée à base d'Objets Distribués) implémente ces concepts. L'objet de ce chapitre est de décrire la structure fonctionnelle et les choix d'implémentation sur lesquels nous nous sommes appuyés : architecture du système, implémentations des communications, protocoles de gestion de la cohérence, mise en oeuvre de la notion de groupe.

6.1 L'architecture du système

6.1.1 Des processus dédiés

Le système DOSMOS s'appuie sur un découplage fonctionnel entre gestion de l'espace des objets partagés et code application. Ainsi, à la base, notre système est composé de deux classes de processus :

- **Processus Application (P.A.)**: il contient et exécute le code de l'application. Dans la version actuelle, cette application doit être écrite en C et peut intégrer des primitives PVM (section 6.7). Ce processus est connecté à un processus mémoire (Figure 6.1) ;
- **Processus Mémoire (P.M.)**: ce processus traite les requêtes émanant des processus application dont il a la charge (voir Figure 6.2). Il communique pour cela, on le verra, avec les autres processus mémoire de son groupe.

On pourrait reprocher à ce découplage fonctionnel d'accroître le nombre de processus, d'où un nombre de changements de contexte plus important. Mais il présente surtout de nombreux avantages : plus grande modularité, facilité de maintenance, souplesse d'utilisation, meilleure structuration de l'application (par le biais de l'association PM/PA), facilité de traçage d'exécution et d'optimisation de code (chapitre 7).

1. DOSMOS ©: Logiciel déposé auprès de l'agence de protection des programmes. Janvier 1996 Numéro : IDDN.FR.001.110021.00.S.P.1996.000.10100

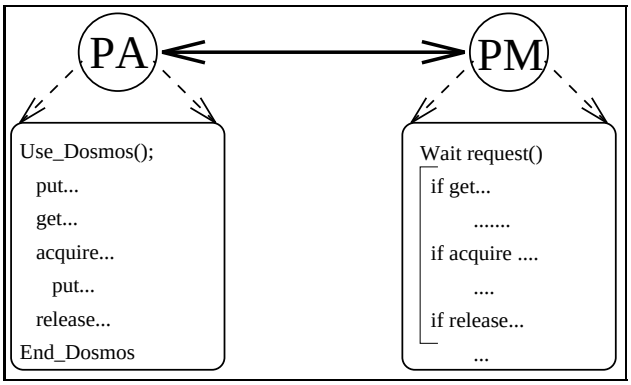


FIG. 6.1 - *Processus mémoire et application*

Les processus mémoire communiquent par le biais de routines PVM. Notre choix s'est porté sur PVM pour deux principales raisons :

- PVM est disponible sur la quasi-totalité des architectures parallèles. Il est également installé sur la plupart des réseaux de stations. PVM était le seul standard disponible lors de la mise en œuvre du premier prototype de DOSMOS. Aujourd'hui concurrencé par MPI, il demeure néanmoins la bibliothèque de communications de référence.

- PVM a été conçu pour programmer des plates-formes hétérogènes. Or celles-ci sont une de nos cibles privilégiées (section 5.7) ;

Pour plus de précisions sur le système PVM, le lecteur pourra se reporter à [GBD+93].

Processus Mémoire et processus Application sont générés par un processus maître. Ils peuvent être placés sur différents processeurs d'une manière transparente pour l'utilisateur (voir section 7).

Un processus mémoire peut "superviser" différents processus application à condition qu'ils fassent tous partie du même groupe de processus[2] (section 6.6). Regrouper différents processus Application autour d'un même processus mémoire présente, en effet, plusieurs intérêts :

- moins de PM $\Longrightarrow$ plus de mémoire disponible et moins de changements de contexte ;

- moins de PM $\Longrightarrow$ moins de communications, d'où un réseau moins chargé ;

- plusieurs PA associés à un même PM $\Longrightarrow$ possibilités d'optimisation (si un PA demande une donnée dont le propriétaire est "client" du même PM, l'accès à cette donnée est réalisé sans communication inter-PM) ;

- plusieurs PA associés à un même PM $\Longrightarrow$ localité des accès (sur les architectures qui le permettent)

2. Dans la suite, on dira qu'un PM *fait partie* du groupe de processus dont dépendent les PA qu'il supervise.

Une telle multi-association n'est cependant viable que si le processus mémoire n'est pas trop surchargé par les requêtes de ses processus application.

L'interaction processus application - processus mémoire est régie selon un mode client-serveur (figure 6.1) : si un P.A. veut accéder à un objet partagé non présent dans sa mémoire locale (ou invalidé), il envoie sa requête au processus mémoire dont il dépend. Ce dernier prend alors en charge toutes les communications nécessaires pour retrouver l'objet partagé (voir section 6.2). Un processus mémoire joue donc un rôle de serveur de requêtes émanant soit des P.A. dont il a la charge, soit d'autres P.M..

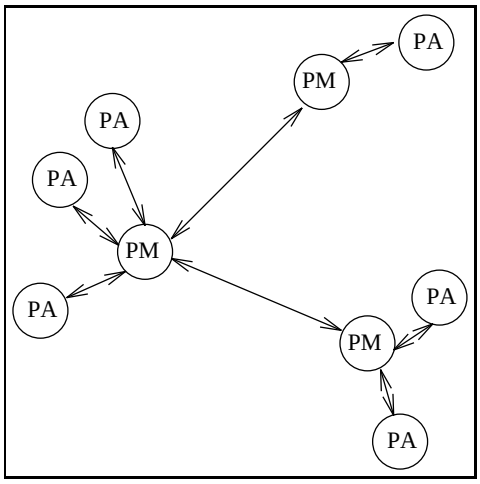


FIG. 6.2 - *Exemple de configuration avec des Processus Application et Processus Mémoire*

Cette implémentation modulaire à base de processus indépendants permet d'adapter le système DOSMOS à des machines parallèles qui ne supportent que des systèmes mono-processus (ex: Cray T3D [LBR94b, LBR94a]). Les Processus Mémoire et les P.A. sont alors placés sur des processeurs indépendants. Sur des machines parallèles avec des processeurs multi-processus ou sur des réseaux de stations, il peut s'avérer pertinent, en terme de performances, d'exécuter concurremment un (des) processus application et son (leur) Processus Mémoire associé sur le même noeud. Enfin, dans un souci de portabilité, nous ne faisons aucune hypothèse sur l'architecture et la topologie utilisée pour connecter ensemble les processus mémoire. Tous ces paramètres sont pris en compte par la couche PVM qui gère l'hétérogénéité du système.

6.1.2 Langage de programmation

DOSMOS propose un langage de programmation composé de primitives dédiées à la l'utilisation d'objets partagés et combinées avec le langage C. En d'autres termes, l'utilisateur définit son application en langage C enrichi de directives de compilation et de primitives spécifiques à la gestion des objets partagés. Un analyseur de code et le transforme en code C ANSI intégrant des appels aux routines PVM (pour plus de détails sur le pré-compilateur voir section 7).

Déclaration des objets partagés

La déclaration des objets partagés se fait par l'intermédiaire de la directive *shared* :

shared type_objet nom_objet(nb_lignes, nb_colonnes) [***nb_ligne_bloc,nb_col_bloc***];

Cette primitive permet de déclarer au système DOSMOS les objets partagés et de donner les directives de découpage des objets partagés complexes. Dans DOSMOS, l'utilisateur a ainsi le choix entre trois types d'objets :

- **Objets locaux** : nous n'avons pas opté pour le partage total. L'utilisateur peut déclarer des objets locaux (variables du langage C) ;
- **Simples objets partagés** : le système offre la possibilité de manipuler des objets de petite taille qui sont partagés par les différents processus. Les types de ces variables partagées sont basés sur ceux du langage C : entiers, flottants et chaînes de caractères.... Chaque déclaration de variable avec la directive **shared** génère un objet partagé (figure 6.3). Cet objet est ensuite associé à la création d'une entrée dans la table des objets (section 6.1.2).

```
shared int Simple_entier;
shared float Pi;
shared char Caractere_resultat;
```

FIG. 6.3 - *Exemples de déclarations d'objets partagés simples*

- **Objets partagés complexes de grande taille** : pour des tableaux de donnés, l'utilisateur peut préciser un mode de découpage. Ces techniques de découpage s'apparente au découpage de tableaux proposé par des langages de parallélisme de données comme HPF. Nous proposons trois méthodes pour découper rapidement des objets complexes :
  - **Découpage par ligne ou par colonne** : avec la primitive *shared*, l'utilisateur peut préciser ***by row*** pour un découpage en lignes ou ***by col*** pour un découpage en colonnes. Le système génère alors autant d'objets systèmes qu'il y a de colonnes et de lignes dans l'objet partagé global de référence (figure 6.5) ;
  - **Découpage en blocs de données** : cette méthode permet le découpage des objets complexes en blocs de taille fixée par l'utilisateur. Les système génère alors le nombre de sous-objets correspondant au nombre de blocs. La plupart des blocs ont la même taille. Le système DOSMOS génère des blocs de tailles différentes lorsque le nombre d'éléments dans un objet partagé global n'est pas un multiple du nombre d'éléments d'un bloc. Par exemple, dans la figure 6.4, la déclaration **shared** *float Matrice[10,10]* **by row** génère la création de 10 objets indépendants. Chacun de ces objets contenant une ligne de 10 nombres flottants. Par contre, le découpage de l'objet *Résultat* génère des blocs de tailles différentes. Les derniers blocs de l'objet *Resultat* auront ainsi une plus petite taille que les autres (c.f. figure 6.5). Ces opérations sont totalement transparentes à l'utilisateur.

```
shared char Gros_objet(100,100);      /* Définition d'un tableau de caractères */
                                      /* non découpé */
shared float Matrice(10,10) by row;   /* Découpage de la matrice en lignes     */
shared int Vecteur[100](1,1)          /* Découpage en blocs d'un élément       */
shared double Resultat[50,70](5,3)    /* Découpage en blocs 5X3. Les derniers  */
                                      /* blocs n'auront pas la même taille que */
                                      /* les autres  */
```

FIG. 6.4 - *Exemples de déclarations d'objets partagés de grande taille*

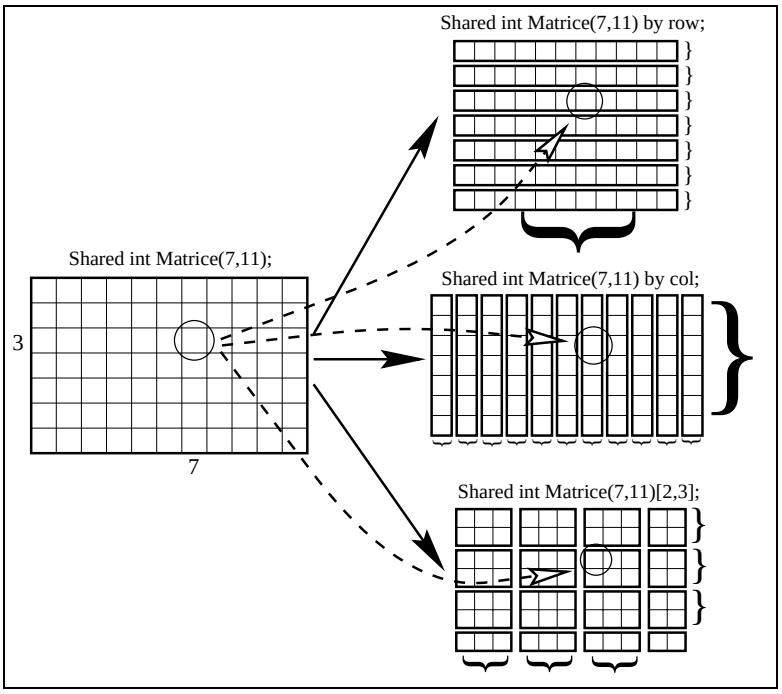


FIG. 6.5 - *Découpage des objets complexes*

L'intérêt de ce découpage en *objets systèmes* est multiple. D'une part, il permet de réduire le volume des communications, les objets systèmes manipulés étant de plus petite taille que les objets de base. D'autre part, il minimise le nombre des attentes dans la mesure où les acquisitions (section 6.4.1) ne sont plus effectués sur l'objet global mais sur les seuls objets-système concernés. Enfin, le découpage d'objet permet de diminuer le nombre de goulots d'étranglement, chaque objet système étant géré par un processus propriétaire.

Les accès aux objets système, ainsi créés, se font de manière transparente pour l'utilisateur qui a l'impression d'accéder à un seul gros objet partagé. C'est le système DOSMOS qui assure la correspondance entre les coordonnées dans l'objet partagé global et les coordonnées réelles dans les sous-objets (cf figure 6.5 : pour l'utilisateur, l'accès à l'élément en ligne 3 et colonne 7 se fait d'une manière identique, quel que soit le découpage choisi). La transparence est donc totale pour l'utilisateur qui doit simplement déclarer ses objets et donner ses directives de découpage.

Table des objets

Une table des objets est générée de manière individuelle pour chaque processus. Elle comporte les informations élémentaires concernant l'objet : son type, son nom, sa taille. Dans le cas d'objets complexes, elle comporte également le nombre de lignes et de colonnes, le mode de découpage de l'objet (***ROW***, ***COL***, ***BLOCK***) et la taille des sous-objets systèmes (***Nb Lig. Bloc*** et ***Nb Col Bloc***). Le choix du protocole d'invalidation (***Inv.***) influence le mode d'invalidation de l'objet : en invalidation simple (***I.***), en écriture-diffusion (***W.B.*** Write Broadcast) ou automatique (***AUTO***) (voir section 6.3).

Type	Nom Objet	Nb Lig.	Nb Col.	Découpage	Nb Lig. Bloc	Nb Col. Bloc	Inv.
int	Simple_entier						W.B.
float	Matrice	10	10	ROW	1	10	AUTO
char	Gros_objet	100	100				W.B.
double	Resultat	50	70	BLOCK	5	3	I.

FIG. 6.6 - *Exemple de table des objets*

Cette table des objets est connue par tous les Processus Mémoire qui sont associés à des PA abonnés aux groupes de partage des objets. De plus, une table des objets globale simplifiée est connue par les Processus Passerelle afin de permettre les accès extra-groupes (section 5.4.2).

Primitives DOSMOS

La librairie DOSMOS est composée de cinq primitives (figure 6.7). Les accès non-exclusifs (sans appel aux primitives acquire - release) sont, grâce à l'analyseur de code, totalement transparents pour l'utilisateur. Ils sont programmés de la même manière que les accès aux variables locales, à l'aide des instructions du langage C ANSI.

• #include	`Dosmos.h`
• Début / Fin du partage	`use_dosmos() - end_dosmos()`
• Accès exclusif	`acquire(`*`objet`*`),`
	`release(`*`objet`*`),`
• Synchronisation	`dosmos_sync(`*`objet`*`)`

FIG. 6.7 - *Les primitives DOSMOS*

Les primitives fournies par la librairie DOSMOS sont contenues dans le fichier *Dosmos.h* et doivent être incluses dans le programme DOSMOS. Les primitives *Use_Dosmos()* et *End_Dosmos()* permettent de signaler au système la portion de code dans laquelle l'utilisateur désire accéder aux objets partagés. Pour accéder de manière exclusive[3] aux objets, le système DOSMOS propose

3. Plus précisément pour accéder à une Section à Accès Restreint (section 5.3)

deux primitives, *Acquire* et *Release*, qui permettent de disposer d'un modèle de cohérence à la libération (section 3.4.2). L'utilisateur peut profiter d'une primitive de synchronisation *dosmos_sync* qui permet de synchroniser les processus qui partagent l'objet spécifié (section 6.5).

6.2 Accès aux objets

Différents protocoles sont mis en jeu pour fournir aux processus Application les objets dont ils ont besoin :

- **Lecture** : lorsqu'il a besoin d'un objet pour lequel il ne dispose pas d'une copie valide dans sa mémoire locale, un Processus Application émet une requête vers son processus mémoire. Quatre cas de figures sont alors possibles :
  - Le PM dispose dans sa mémoire locale d'une copie valide de l'objet. Il la transmet alors au PA demandeur.
  - Le PM ne dispose pas d'une copie valide. Par contre il est associé au Processus Application propriétaire de l'objet. Il attend que celui-ci lui transmette la nouvelle valeur de l'objet (qu'il fasse un *release*) puis la communique au processus demandeur.
  - Le PM ne dispose pas d'une copie locale et le PA propriétaire est associé à un autre PM situé dans le même groupe (figure 6.8). Le PM associé au processus Application demandeur retransmet la requête vers le PM associé au PA propriétaire. Celui-ci renvoie une copie valide de l'objet qui est ensuite communiquée au PA demandeur.
  - Le PM ne dispose pas d'une copie locale et le PA propriétaire fait partie d'un autre groupe. Ce cas nécessite de sortir du groupe dont dépend le PA demandeur. Il est décrit dans la section présentant l'implémentation des groupes hiérarchiques (section6.6).

  Quel que soit le cas de figure, le PA demandeur est bloqué en attente de la réception de la copie valide de l'objet. Dans les trois derniers cas, après avoir transmis la copie de l'objet au Processus Application demandeur, le PM met à jour sa propre copie de l'objet afin d'optimiser les accès ultérieurs.
- **Ecriture** : trois cas de figures sont possibles :
  - Ecriture intra-groupe hors Section à Accès Restreint [4] : après avoir modifié un objet dans sa mémoire locale, un processus application envoie la nouvelle valeur à son Processus Mémoire (figure 6.9). Ce dernier, selon le protocole d'invalidation choisi (section 6.3), invalide ou envoie la nouvelle valeur, d'une part, aux autres PA dont il a la charge, d'autre part, aux processus Mémoire participant au même groupe. Chacun de ces PM retransmet l'invalidation ou la nouvelle valeur à ses PA qui ont une copie de l'objet.
  - Ecriture intra-groupe dans une Section Accès Restreint : dans ce cas, les modifications ne sont pas retransmises au PM. Seule la dernière modification est notifiée au PM lors de l'appel à la primitive *release* (section6.4.1).

4. Section à Accès Restreint (SAR) : instructions encadrées par les deux primitives acquire et release

– Ecriture extra-groupe : à l'instar des lectures extra-groupes, ce cas de figure nécessite la mise en oeuvre de protocoles spécifiques (section 6.6). Remarque : comme on l'a déjà remarqué en section 5.4.2 les accès extra-groupes sont réalisés en cohérence stricte. Il est donc impossible d'utiliser les primitives acquire et release (en vue de définir une SAR) dans le cadre d'accès extra-groupe.

Toutes ces opérations sont totalement asynchrones : pendant qu'ils attendent des copies d'objet, les Processus Mémoire ne demeurent pas bloqués et continuent à répondre aux requêtes d'autres Processus Application.

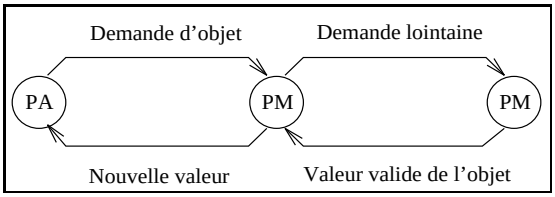


FIG. 6.8 - *Lecture d'objet*

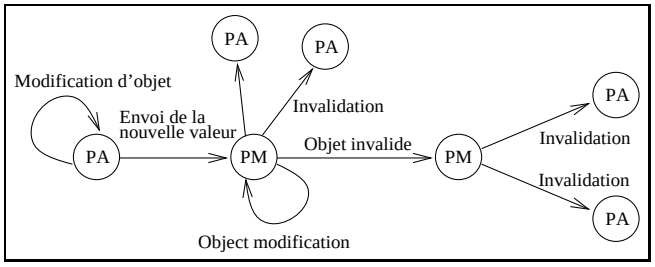


FIG. 6.9 - *Ecriture sur un objet*

6.3 Protocoles d'invalidation

Nous avons essayé d'optimiser cette étape qui est l'un des principaux goulots d'étranglement des systèmes de MDP. Ces protocoles, nous l'avons vu dans la section 3.7, sont nécessaires pour maintenir la cohérence des copies des objets partagées présents chez les autres processus. Nous adaptons une stratégie précédemment développée pour la gestion des pages [HERV93]. Celle-ci intègre les deux protocoles de mise à jour existants : l'invalidation ou l'écriture-diffusion (figure 6.10). Pour limiter des communications contraignantes sur de gros objets (e.g. : volumineux tableaux de données), nous invalidons simplement les copies des objets modifiés en envoyant un message d'invalidation de taille réduite. Cette méthode est aussi employée lors de l'utilisation d'objets découpés mais dont le facteur de découpage est encore trop élevé pour permettre une mise à jour. Cependant pour des objets de petite taille, le coût d'envoi de tels messages d'invalidation peut coûter presque aussi cher que l'envoi de l'objet lui-même. Ainsi, sur de petits objets comme les variables de base, les petits tableaux ou les objets-système de petite taille, nous mettons directement à jour les copies en envoyant la nouvelle valeur de l'objet (figure 6.11). Avant l'invalidation ou la mise à jour des copies, le système vérifie si la valeur de l'objet a été effectivement modifiée afin d'éviter des communications inutiles. En combinant toutes ces techniques, nous minimisons ainsi le coût du maintien de la cohérence des copies des objets partagés dans DOSMOS.

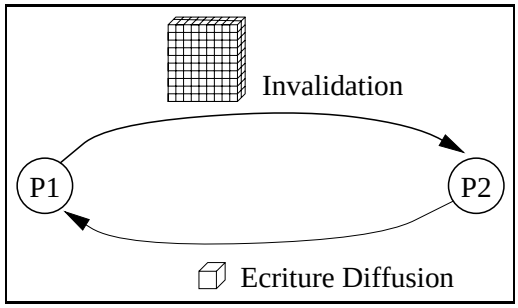


FIG. 6.10 - *Invalidation ou Ecriture-Diffusion en fonction de la taille des objets partagés*

```
Invalidation(objet)
  si Modifié (objet)
  | si Protocole == Invalidation
  | | Invalidation (objet)
  | sinon si Protocole == Ecritue Diff.
  | |     | Ecriture_Diffusion (objet)
  | |     sinon si taille(objet) > Taille(message_base)
  | |     |     | Ecriture_Diffusion (objet)
  | |     |     sinon
  | |     |     | Invalidation (objet)
  sinon Objet Non Invalidé
```

FIG. 6.11 - *Protocole d'invalidation*

6.4 Protocoles de cohérence

De nombreux protocoles de cohérence ont été proposés dans la littérature. Ils permettent de garantir une conformité d'accès en lecture et écriture aux objets partagés et de garantir un certain déterminisme à l'application (voir section 3.4). Deux implémentations de sémantiques de cohérence sont proposés par DOSMOS :

6.4.1 Cohérence faible :

Même si l'utilisation d'une cohérence forte rassure le programmeur, il n'est pas toujours nécessaire de synchroniser tous les accès aux objets partagés (cf chapitre 8) d'autant plus que ces synchronisations coûtent cher. DOSMOS offre donc la possibilité à l'utilisateur d'accéder aux objets partagés par l'intermédiaire de protocoles de cohérence faible plus performants.

L'implémentation des mécanismes de cohérence est basée sur l'utilisation de deux opérateurs: *acquire* (acquérir) et *release* (relâcher).

Acquérir un objet : *Acquire*

Cet opérateur est utilisé quand un processus veut accéder de manière exclusive à un objet (au sens d'une section à accès restreint). Selon le propriétaire de l'objet partagé (figures 6.12,6.13), la demande d'acquisition peut traverser divers Processus Mémoire avant d'atteindre le propriétaire. Un acquire ne peut être confirmé que lorsque tous les *acquire* précédents ont été relâchés. A l'issue d'un acquire, une SAR est définie sur l'objet : aucun autre acquire ne sera validé tant que le Processus Application n'aura pas relâché l'objet via un *release*. Ce Processus Application peut donc travailler sur sa copie de l'objet sans être perturbé par les autres PA. Notons que ces derniers ont toujours la possibilité de travailler avec leur propre copie de l'objet. Celle-ci sera, cependant, écrasée lors de l'opération *release*.

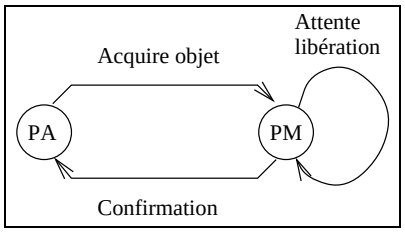


FIG. 6.12 - *Acquire quand Processus Mémoire propriétaire*

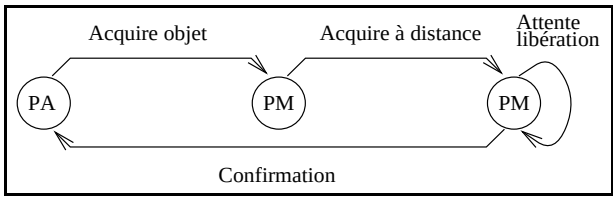


FIG. 6.13 - *Acquire quand Processus Mémoire non propriétaire*

Comme on l'a vu, un processus peut uniquement acquérir des objets présents dans son Espace Virtuel Local, c'est à dire, en l'absence de mise en oeuvre de stratégies dynamiques, dans son groupe (section 5.4.5).

L'implémentation de la primitive Acquire revêt différentes formes selon le découpage de l'objet requis :

- Acquisition d'un objet non-découpé : ceci concerne d'une part les objets simples qui ne peuvent être découpés, d'autre part, les objets complexes qui ne sont pas découpés (c'est à dire ceux dont l'utilisateur veut garantir un accès exclusif sur l'ensemble de l'objet), avec les risques que cela comporte pour les performances d'accès à l'objet.

```
acquire (pi);
acquire (Gros_objet);
```

FIG. 6.14 - *Acquisition d'objets non-découpés*

- Acquisition d'objets découpés en lignes ou en colonnes : dans le cas de l'acquisition de la figure 6.15, les comportements semblent identiques. En fait, l'acquisition de *Gros_objet* porte sur la totalité de cet objet qui n'est pas découpé. L'objet *Matrice*, lui, est découpé en lignes de 10 éléments. L'acquisition sur cet objet (figure 6.15) concerne donc seulement l'objet système correspondant à la troisième ligne. Les autres objets-système de *Matrice* restent libres et peuvent être acquis par d'autres processus.

```
acquire (Gros_objet[10]);
acquire (Matrice[3]);
```

FIG. 6.15 - *Acquisition d'objets découpés en lignes ou colonnes*

- Acquisition d'objets découpés en blocs : la primitive *Acquire* permet aussi de réserver des blocs de données lors de l'utilisation d'objets découpés. Son utilisation peut aboutir à différents

actions et protocoles. Ainsi dans l'exemple 6.16, l'objet *Résultat* est partagé en blocs de $5 * 3$ éléments. Le premier exemple provoque une réservation de l'objet complet, soit l'ensemble des sous-objets systèmes (blocs) qui composent *Résultat*. Le deuxième exemple aboutit à la réservation de l'ensemble des blocs qui comportent une partie de la huitième ligne de l'objet. Enfin, les deux derniers exemples d'acquisition de *Resultat* provoquent la réservation d'un seul bloc du gros objet. Les autres blocs peuvent être acquis par d'autres Processus Application. L'adressage du bloc correspondant peut se faire de manière dynamique et dépendre des valeurs contenues dans d'autres objets partagés (ici *Vecteur*). Dans un cas extrême d'objet découpé en blocs d'un seul élément, l'acquisition peut porter sur un élément de l'objet.

```
acquire (Resultat);
acquire (Resultat[,8]);
acquire (Resultat[10,5]);
acquire (Resultat[i*2,Vecteur[j]]);
```

FIG. 6.16 - *Acquisition d'objets découpés en lignes ou colonnes*

Relâcher un objet : *Release*

En cohérence à la libération (section 3.4), l'opérateur *release* est le seul opérateur qui garantit l'invalidation des copies avant de permettre de nouveaux accès à l'objet. Ainsi, après un appel à l'instruction *release*:

- Toutes les copies de l'objet sont invalidées ou mises à jour chez les autres processus s'il y a eu, au moins, une modification dans la Section à Accès Restreint. Sinon, aucune modification n'est propagée (figure 6.17) ;
- Le processus Mémoire associé au PA propriétaire de l'objet attend de chaque PM du groupe un message de confirmation attestant que ceux-ci ont bien invalidés les copies présentes chez les PA qu'ils supervisent (figure 6.18).

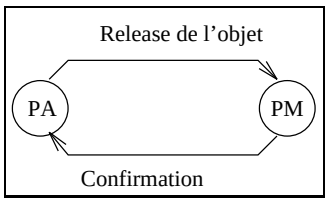


FIG. 6.17 - *Release sans modification*

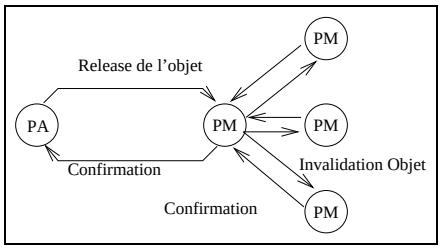


FIG. 6.18 - *Release avec modification*

- Tous les processus peuvent acquérir de nouveau l'objet qui vient d'être relâché ;

Comme on peut le voir dans les figures précédentes (6.17 et 6.18), l'invalidation n'est effective que si l'objet a été réellement modifié entre l'*acquire* et le release. ceci afin d'éviter des invalidations inutiles pour des objets qui avaient été verrouillés pour des lectures seules. Bien entendu, pour respecter notre organisation hiérarchique en groupes, nous invalidons seulement les copies chez les processus qui appartiennent au groupe partageant l'objet libéré.

Le relâchement des objets doit correspondre exactement à l'acquisition qui en avait été faite. De même que l'acquisition, la primitive Release peut s'utiliser de diverses façons : relâchement d'objets non-découpés, relâchement d'objet découpés en lignes, en colonnes ou en blocs.

```
release (pi);
release (Gros_objet);
release (Matrice[3]);
```

FIG. 6.19 - *Relâchement des objets précédemment acquis*

Des inter-blocages pourraient se produire si un objet est acquis mais n'est jamais relâché. L'outil d'analyse de code que nous avons développé (section 7.1) permet de détecter ce type d'erreurs.

6.4.2 Cohérence linéarisable

Il est très facile d'émuler une sémantique de cohérence linéarisable à l'aide des opérateurs acquire et release. Il suffit, en effet, de réaliser l'ensemble des opérations d'écriture (intra-groupe) à l'intérieur d'une Section à Accès Restreint, c'est à dire d'encadrer chaque écriture par une paire de primitives *acquire* - *release*, les lectures étant alors, quant à elles, réalisées hors-SAR.

Remarque : les accès extra-groupes étant réalisés en cohérence stricte, ils ne remettent pas en cause la linéarisabilité des accès intra-groupes.

6.4.3 Cohérence stricte

De la même manière, il suffit de réaliser l'ensemble des accès intra-groupes (lectures et écritures) à l'intérieur d'une Section à Accès Restreint pour garantir une sémantique de cohérence stricte.

6.5 Les barrières de synchronisation

Les barrières de synchronisation sont essentielles pour l'écriture d'une application en mémoire partagée (section 3.4.2). Nous avons basé les barrières de synchronisation de DOSMOS en améliorant les primitives de synchronisation standard fournies par PVM. Deux types de synchronisation

sont possibles :

- Synchronisation globale : cela permet de synchroniser tous les processus application. Cette synchronisation est réalisée via l'appel de la primitive *dosmos_sync* sans paramètre (figure 6.20). Cette possibilité est utile pour le lancement synchrone des Processus Application, ceux-ci pouvant mettre des temps différents pour se lancer sur un réseau de stations de travail.

- Synchronisation de groupes : la primitive *dosmos_sync*, avec comme paramètre un objet partagé, permet de synchroniser le groupe de processus utilisant cet objet. La primitive *dosmos_sync* ne peut s'employer qu'avec des objets utilisateur et non pas avec des objets système. En effet, les processus sont abonnés à des groupes de partage d'objets sans tenir compte de leur mode de découpage transparent à l'utilisateur. Ainsi les deux derniers exemples fournissent les mêmes résultats en synchronisant les processus inscrits au groupe de partage de l'objet *Vecteur*.

```
dosmos_sync ();
dosmos_sync (Vecteur);
dosmos_sync (Vecteur[3,5]);
```

FIG. 6.20 - *Synchronisation des processus*

6.6 Implémentation des groupes

6.6.1 Processus Passerelles

Fondé sur le modèle décrit section 3.4, DOSMOS propose une organisation hiérarchique en groupes de processus. Afin de permettre l'accès aux objets gérés par d'autres groupes, dans chaque groupe un processus mémoire joue le rôle de Processus Passerelle (P.P.) (voir Figure 6.21). Ce Processus Mémoire spécialisé prend en charge les communications inter-groupes. En pratique quand un processus Application essaie d'accéder à un objet dont il n'a pas de copie valide, il envoie une requête à son PM. Si cet objet appartient à un autre groupe, il n'est donc pas référencé dans la table d'objets de ce Processus Mémoire. Celui-ci transmet donc la requête au Processus Passerelle de son groupe. Ce dernier possède une copie de toutes les tables d'objets. Il connaît le nom des objets partagés par chaque groupe. Il retransmet alors la requête vers le processus passerelle du groupe qui gère l'objet demandé. Celui-ci renvoie une copie valide vers son correspondant qui la retransmet au Processus Mémoire initial.

Les processus Passerelles sont les seuls points d'entrée externes dans un groupe. Ils permettent une gestion décentralisée des accès extra-groupes. Le choix et le placement des processus passerelles incombe à l'utilisateur (voir section 7) pour un placement optimal au sein de la machine virtuelle.

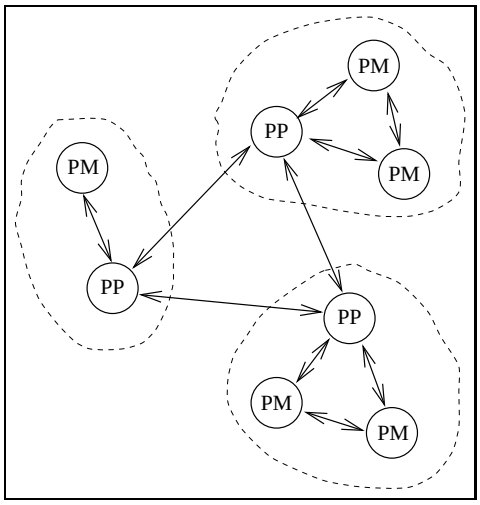


FIG. 6.21 - *Processus Mémoire et Processus Passerelles*

6.6.2 La table des groupes

Les groupes sont répertoriés dans une table dynamique spécifiée par l'utilisateur avant l'exécution. La figure 6.22 propose un exemple d'application hiérarchique. La table des groupes utilisée précise pour chaque processus les objets partagés auxquels il accède (c'est à dire ceux présents dans son Espace Virtuel Local).

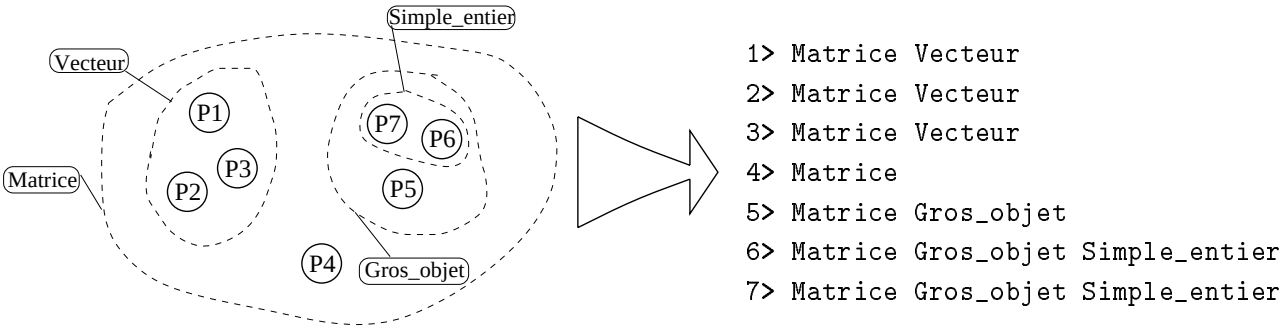


FIG. 6.22 - *Application utilisant trois groupes hiérarchiques et table des groupes correspondante*

Cette table est dupliquée dans chaque Processus Passerelle. Les PM ne connaissent que la partie de la table concernant les Processus Application de leur groupe.

Une telle structure peut sembler surprenante, dans la mesure où elle correspond plus à une notion de domaines d'objets qu'à une notion de groupes. Elle se justifie, cependant, si l'on désire mettre en oeuvre des stratégies d'adaptation dynamiques fondées sur des abonnements. En l'absence d'abonnement, tous les objets d'un même groupe partageant les mêmes objets, il suffirait de préciser pour chaque groupe la liste de ses objets.

6.6.3 Une configuration complète

La figure 6.23 présente un exemple de configuration complète. Les PA dialoguent avec leur PM dédié. Ils sont abonnés à des groupes de partage en fonction des objets auxquels ils accèdent le plus souvent. Les PM communiquent entre eux pour faire régner la cohérence à l'intérieur des groupes. Les PP dialoguent avec les PM du groupe et avec les autres PP de la configuration afin de permettre les accès aux objets en dehors des groupes.

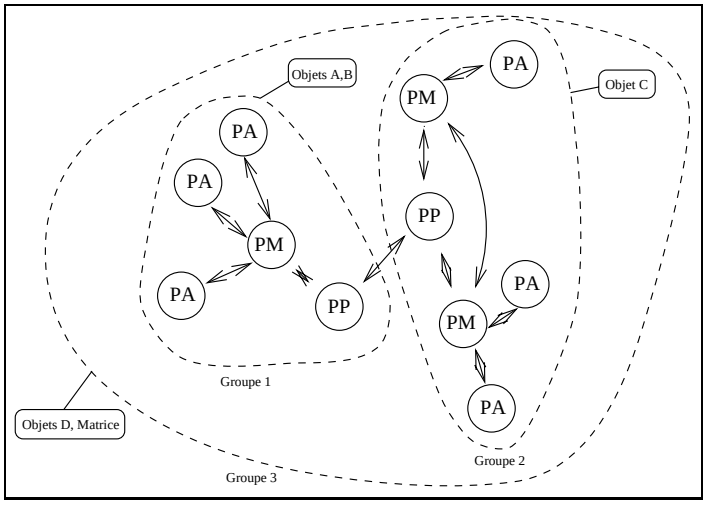


FIG. 6.23 - *Une configuration complète*

6.7 Modèles de programmation

Après avoir utilisé la primitive **Use_Dosmos()**, l'utilisateur peut accéder à ses objets de manière transparente. Pour optimiser son application et pour le confort de programmation, l'utilisateur peut choisir son modèle de programmation :

- Utilisation d'objets locaux : pour minimiser les accès aux objets partagés, l'utilisateur peut tirer avantage de l'utilisation de variables locales avant de modifier des objets partagés.

- Applications avec M.D.V.P. : pour bénéficier de la transparence en placement et gestion de ses données, l'utilisateur peut aussi entièrement programmer son application en utilisant des objets distribués partagés (à l'aide des primitives DOSMOS).

- Objets partagés et échange de messages : notre environnement permet à l'utilisateur de mélanger différents modèles de programmation. Il peut aussi combiner des accès DOSMOS à des objets partagés avec l'utilisation de primitives PVM générant de l'échange de messages. Ainsi, une ancienne application écrite en PVM peut être facilement portée sur DOSMOS après quelques modifications pour l'intégration et la manipulation d'objets partagés. Voir l'exemple de l'algorithme du Zbuffer (section 8.5).

6.8 Conclusion

La réalisation du système DOSMOS a permis l'implémentation des concepts originaux introduits dans notre modèle de MDVP. L'utilisation de processus dédiés et d'objets découpés apporte une bonne modularité aux applications (section 8.3). De plus, la combinaison de groupes hiérarchiques avec des modèles de cohérence faibles permet de réduire les coûts de gestion des objets partagés. Mais le système DOSMOS ne pourrait être réellement utilisable sans l'intégration d'un véritable environnement de programmation, décrit dans le chapitre suivant, qui prenne en charge l'utilisateur dans la conception de ses applications.

CHAPITRE 7

Vers un environnement de programmation parallèle intégré

Programmer au-dessus d'une MDVP (même au-dessus de DOSMOS !) ne dispense pas de penser parallèle. Le développement d'un véritable environnement de programmation, capable d'aider le programmeur à concevoir et optimiser ses applications parallèles, est indispensable. Beaucoup de systèmes à MDVP se contentent d'un environnement de programmation très sommaire. Seuls quelques systèmes comme KOAN [BPR96], ORCA [TKB92] ou MIDWAY [BZS93] proposent des environnements de programmation capables d'aider l'utilisateur. Mais ils sont limités à un niveau de développement : compilation, ou visualisation des exécutions. L'environnement de programmation de DOSMOS a été conçu dans le souci d'aider le programmeur à tirer partie de l'ensemble des fonctionnalités du système DOSMOS. De l'installation du système à la conception des applications et jusqu'à l'analyse de performances le programmeur dispose d'outils conviviaux et puissants. L'objet de ce chapitre est de décrire les fonctionnalités de cet environnement et d'analyser les principales modalités d'interaction de l'utilisateur avec ce dernier.

7.1 L'analyse des applications DOSMOS

Un des premiers apports que nous voulons offrir par l'intermédiaire de cet environnement de programmation est une grande transparence pour l'utilisateur. Nous ne pouvions pas nous contenter d'un environnement basé sur une bibliothèque offrant des primitives dédiées et nécessitant une réécriture complète des applications. L'environnement est donc basé sur un outil d'analyse de code complet qui prend en charge l'utilisateur. Cette analyse est fondée sur des outils d'analyse syntaxique (*Lex*) et sémantique (*Yacc*).

La transparence est assurée à différents niveaux de la programmation parallèle; la gestion des objets partagés est complètement cachée à l'utilisateur et les accès aux objets partagés sont extrêmement simplifiés. De plus, une phase de détection d'erreurs et d'optimisation du code aide l'utilisateur à réaliser les premières corrections et optimisations dans son application. Nous allons voir que cette transformation automatique du code permet une programmation sous DOSMOS d'une manière très proche de la programmation séquentielle.

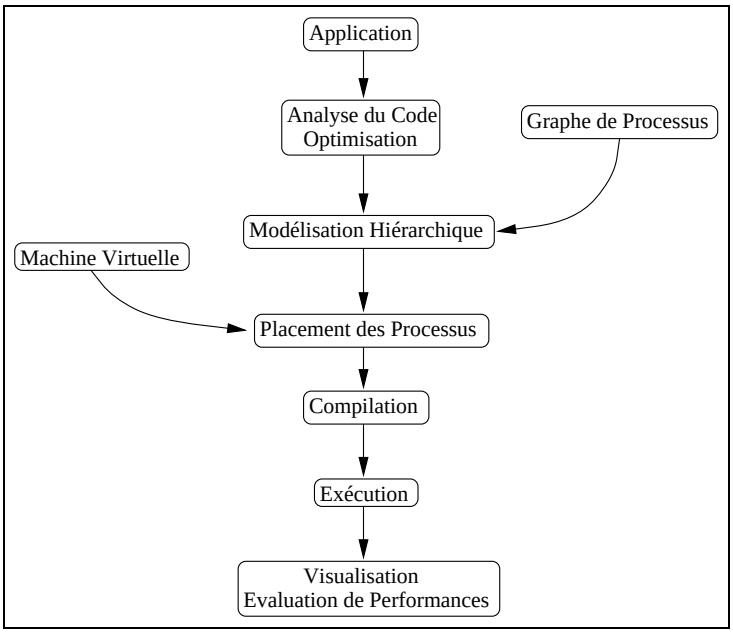


FIG. 7.1 - *L'environnement de programmation : prise en charge de l'utilisateur de la conception du code jusqu'à l'évaluation des performances*

7.1.1 Transparence de la programmation parallèle

Gestion des objets partagés

L'analyse du code permet l'extraction de l'ensemble des objets partagés définis par l'utilisateur. Ces objets sont regroupés et sauvegardés dans une structure de données indépendante : la table des objets. Cette table sera nécessaire à l'utilisateur pour la définition hiérarchique des groupes de processus (voir section 7.2.3).

Cette table offre à l'utilisateur une meilleure appréhension du partage et du découpage des objets auxquels il accède dans son application. Elle permet également au système de lier les objets utilisateurs, accédés par leur nom, avec les objets systèmes accédés à l'aide d'un numéro logique.

Accès aux objets partagés

Pour accéder à des objets partagés l'utilisateur n'utilise pas de primitives DOSMOS. Il accède à ces objets de la même manière qu'à des variables locales. La transformation de ces accès est prise en charge par l'environnement. Celui-ci détecte et analyse tous les accès aux objets partagés et génère les appels aux primitives DOSMOS correspondantes. Les deux primitives principales d'accès à des objets en mémoire partagée sont la fonction get (pour la lecture d'un objet) et la primitive put (pour l'écriture). Ces primitives comportent divers paramètres :

- Lecture : **get(objet, ligne, colonne)** : le paramètre **objet** désigne le numéro logique de l'objet accédé. Lors de l'utilisation d'un objet complexe (Vecteur ou Matrice), les paramètres

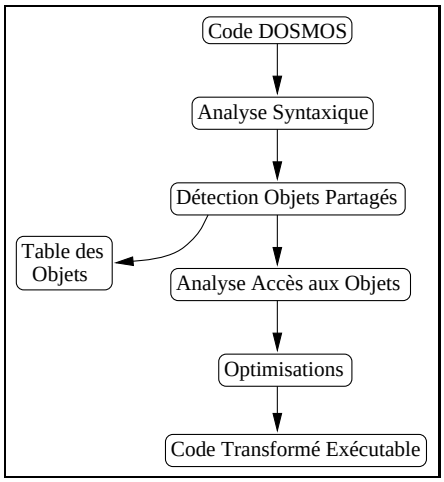


FIG. 7.2 - *Etapes dans l'analyse d'un code d'une application DOSMOS*

```
shared int Maximum;
int i,j;
shared float Matrice[100,200](5,5);
shared double Vecteur[100] by row;

Use_Dosmos();
...
...
End_Dosmos();
```

FIG. 7.3 - *Exemple de code DOSMOS avec la déclaration d'objets partagés*

int	Maximum	-	-	-	-	-
float	Matrice	100	200	BLOCK	5	5
double	Vecteur	100	-	ROW	-	-

FIG. 7.4 - *La table des objets partagés extraite du code DOSMOS*

ligne et colonne sont utilisés comme coordonnées de déplacement à l'intérieur de l'objet de taille importante ;

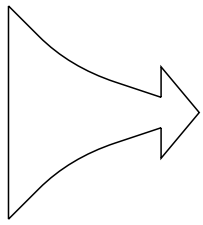
- Ecriture : la primitive **put (objet, ligne, colonne, valeur)** comporte des paramètres identiques à la primitive de lecture avec un paramètre supplémentaire qui permet d'indiquer la valeur à écrire dans l'objet.

L'analyseur de code transforme donc les accès aux données partagés en utilisant des variantes de ces deux primitives en fonction du type des objets : **put_int**, **put_float**, **get_char**, **get_double**... Ces primitives se présentent sous la forme **dm_*instruction*** pour les différencier de primitives existantes sur les architectures-cible.

La figure 7.7, exhibe un produit de matrices réalisé au-dessus de DOSMOS. On peut noter que le code programmé par l'utilisateur est très proche du code séquentiel.

```
shared int x;
shared int y;
int local;

main()
{
  use_dosmos();

  x=10;
  y=20;

  local=x;
  x=y;
  y=local;

  end_dosmos();
}
```

FIG. 7.5 - *Echange de valeur entre deux objets partagés à l'aide d'une variable locale : code utilisateur*



```
int local;

main()
{
 dm_use_dosmos();

 dm_put_int(0,0,0, (10));
 dm_put_int(1,0,0, (20));

 local=dm_get_int(0,0,0);
 dm_put_int(0,0,0, (dm_get_int(1,0,0)));
 dm_put_int(1,0,0, (local));

 dm_end_dosmos();
}
```

FIG. 7.6 - *Le même code analysé, transformé et prêt à être compilé*

7.1.2 Détections d'erreurs et optimisations

Cette phase d'analyse nous permet d'appliquer un contrôle de haut niveau sur les applications. L'ajout d'une cinquantaine de règles à la grammaire standard du langage C, permet une analyse poussée des accès aux objets. Ainsi cet outil dédié à de la programmation par mémoire partagée scrute le code de l'utilisateur et détecte les éventuelles erreurs de conception et optimisations possibles. La détection d'erreurs porte sur l'accès aux objets de grande taille. L'analyse détecte les débordements sur les objets et les accès illicites aux tableaux partagés.

L'optimisation du code porte sur les détections d'entrée dans une section à accès restreint. Lorsque le programmeur utilise dans son code les primitives *acquire* et *release* et qu'il n'y aucune écriture effectuée, l'analyseur de code le signale à l'utilisateur. De plus, toute détection d'entrée dans une SAR non suivie d'une libération déclenche une alarme pou l'utilisateur. Celui-ci pourra ainsi, si nécessaire, modifier le code de son application.

Par la suite, nous généraliserons cette optimisation en proposant des accès optimisés aux objets. Cette technique, appelée *prefetching*, est dérivée des méthodes de gestion de caches répartis : l'analyseur parcourt le code de l'application et essaie de limiter les accès distants aux objets. Pour réaliser cela, il génère des accès en avance afin de rapatrier l'objet dans la mémoire locale.

```
#define TAILLE_MAT 5

shared float mat1[TAILLE_MAT][TAILLE_MAT](1,TAILLE_MAT),
             mat2[TAILLE_MAT][TAILLE_MAT](TAILLE_MAT,1);
shared float result[TAILLE_MAT][TAILLE_MAT](1,TAILLE_MAT);

void main() {
    int i;
    int l, c;
    float val;

     use_dosmos();

    /* synchronisation */
     dm_sync(result);

    /* le process DN commence a la ligne DN-1 de la matrice result*/
    l = DN -1;
    c = 0;

    /* puis fait le calcul pour toutes les valeurs de la ligne */
    while( l < TAILLE_MAT ) {
        acquire(result[l][0]);
        for(c = 0; c <mat1:row; c++) {
            val = 0.0;
            for(i = 0; i<mat1:col; i++)
                 val +=  mat1[l][i]*mat2[i][c];
            result[l][c] = val;
}

/* MAJ de la ligne de la matrice resultat */
       release(result[l][0]);

/* incremente le numero de la ligne dans result
   pour le process courant */
      l+=DN_max;
    }
    dm_sync(result);

    if(DN == 1) {
        printf("Resultat :\n");
        affiche_matrice(3);  }

    end_dosmos(1);
}
```

FIG. 7.7 - *Exemple de code DOSMOS: Une multiplication de matrices à l'aide d'objets partagés*

```
void main() {
 int i;
 int l, c;
 float val;

 dm_use_dosmos();


 dm_synchro_barrier(2);


 l = DN -1;
 c = 0;


 while( l < 5 ) {

 dm_acquire(2,l, 0 );

 for(c = 0; c <5; c++) {
 val = 0.0;
 for(i = 0; i<5; i++)
 val += dm_get_float(0,l, i) *dm_get_float(1,i, c) ;
 dm_put_float(2,l, c  , ( val));
 }


 dm_release(2,l, 0 );



 l+=DN_max;
 }


 dm_synchro_barrier(2);

 if(DN == 1) {
 printf("Resultat :\n");
 affiche_matrice(3); }

 dm_end_dosmos(1);
}
```

FIG. 7.8 - *Code de la multiplication de matrices analysé et transformé*

7.2 Création des applications DOSMOS

L'utilisateur va utiliser différents outils qui vont l'assister dans la modélisation de ses applications DOSMOS. L'environnement de programmation regroupe l'ensemble de ces outils et les intègre dans une interface graphique standardisée.

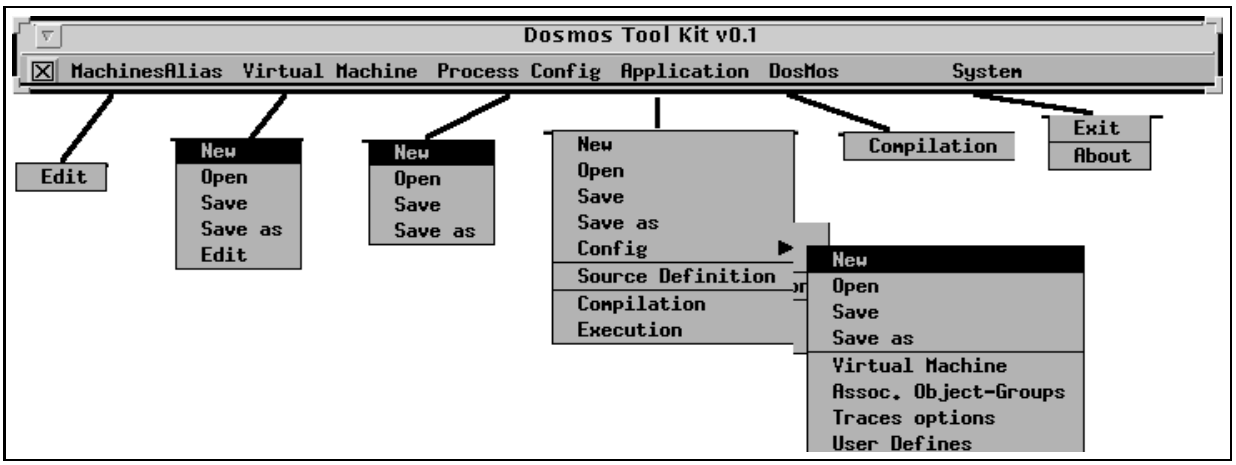


FIG. 7.9 - *Vue globale des menus de l'environnement de programmation*

L'utilisateur est ainsi assisté de bout en bout dans la modélisation des ses applications. L'environnement lui permet de créer des machines virtuelles à base de stations de travail ou d'architectures parallèles, de construire le graphe de processus spécifiant la modularité de son application et de structurer hiérarchiquement son application. En outre, l'environnement facilite toutes les opérations de compilation et d'exécution des applications sur un ensemble de sites distribués. Enfin, l'ensemble des opérations nécessaires à l'installation du système de MDVP DOSMOS sur les machines distantes sont prises en charge. Pour une visualisation complète de l'environnement de programmation, le lecteur pourra se reporter à la figure 7.32 située à la fin de ce chapitre.

7.2.1 Définition de machines virtuelles

La modélisation d'une application DOSMOS est fondée sur l'utilisation d'une machine virtuelle. Une machine virtuelle est un ensemble de sites reliés à l'aide d'un réseau de communication. Ces sites peuvent être des stations de travail locales ou des machines parallèles. La gestion de l'hétérogénéité est prise en charge par l'environnement qui va aider l'utilisateur à définir des machines virtuelles tout en n'ayant qu'un faible nombre de renseignements à fournir.

Ajout d'une machine

Ajouter un nouveau site à la configuration déjà disponible est une fonctionnalité importante de cet environnement de programmation même si elle est peu usitée (la composition du parc de machines varie souvent peu). Cela permet néanmoins d'ouvrir l'environnement à d'autres configura-

tions parallèles ou réparties. En effet, pour notre environnement, une machine peut être une simple station de travail ou une machine massivement parallèle. La généricité et la portabilité se paient par la fourniture de nombreux renseignements : paramètres globaux concernant l'architecture de la machine et paramètres d'implémentation logicielle pour le portage du système DOSMOS.

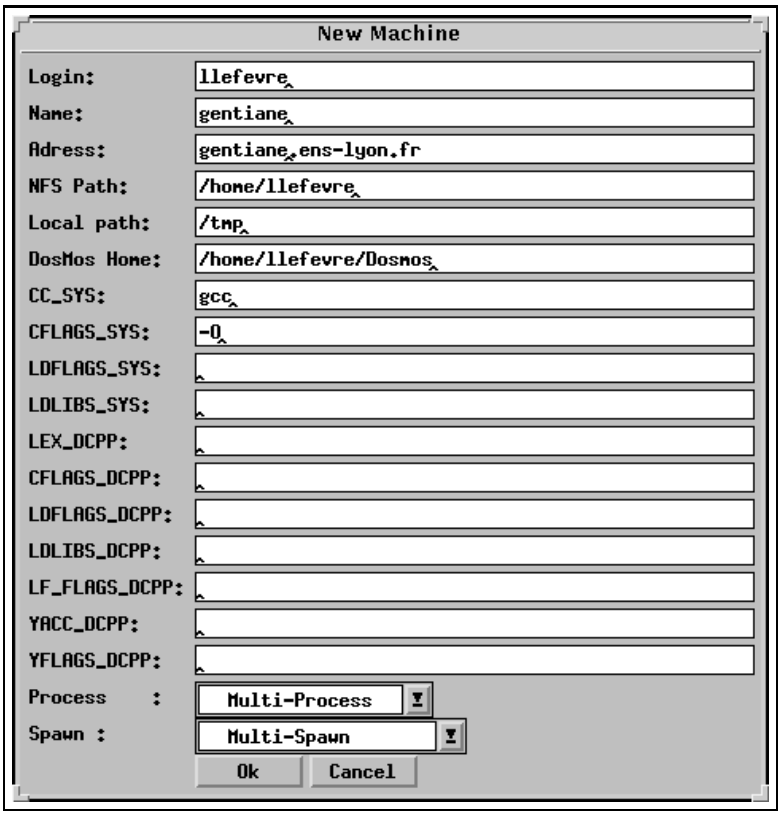


FIG. 7.10 - *L'ajout d'une nouvelle machine*

- **L'architecture de la machine** : l'environnement aide l'utilisateur en retrouvant un maximum d'informations de manière automatique (figure 7.13). L'utilisateur doit, cependant fournir quelques paramètres indispensables tels que le nom de la machine (**Name**) et son adresse complète (**Address**). Afin d'adapter le système à des architectures parallèles particulières, l'utilisateur doit préciser si la machine permet le lancement de processus multiples (**Mono/Multi-Process**). En effet, certaines machines parallèles (comme le Cray T3D) ne permettent qu'une programmation S.P.M.D.[1]. Ces architectures ne supportent donc qu'un seul type de processus pour une application. Le choix de l'option **Mono-Process** permet une fusion des différents processus présents dans DOSMOS en un seul processus générique. Enfin, l'option **Mono/Multi-Spawn** permet de spécifier qu'une machine n'accepte qu'un seul lancement de processus pour toute l'exécution (c'est le cas de la machine Matra Capitan) ;

- **Paramètres utilisateur** : ils sont éventuellement nécessaires lors de l'ajout de machines provenant de sites différents ou de machines parallèles sur lesquelles l'utilisateur est connecté

1. S.P.M.D. : Single Program Multiple Data

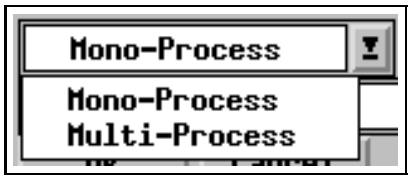


FIG. 7.11 - *Adaptation à des architectures parallèles*

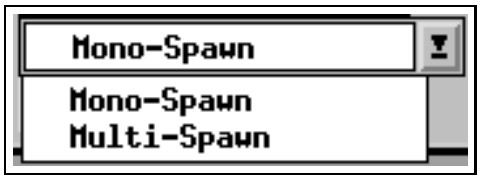


FIG. 7.12 - *Pour différentes configurations matérielles*

sous un autre nom (**Login**). Le chemin d'accès à ses fichiers partagés par NFS (**NFS Path**) et le chemin d'accès aux données présentes sur son disque local (**Local Path**) doivent également être renseignés ;

– **Paramètres logiciels** : le système DOSMOS pouvant être installés sur des architectures et des arborescences différentes, il importe de fournir les renseignements indispensables à son utilisation. Ainsi les caractéristiques des compilateurs installés et leurs options sont indispensables pour la compilation du système et des applications (**CC_SYS**, **CFLAGS_SYS**, **LDFLAGS_SYS**...). De même les renseignements concernant l'installation de l'analyseur de code doivent être fournis par l'utilisateur (**LEX_DCPP**, **YACC_DCPP**, **CFLAGS_DCPP**...).

Tous ces paramètres sont contrôlés ce qui permet de vérifier la validité des informations entrées par l'utilisateur et éventuellement de les compléter automatiquement. Ainsi, l'architecture du processeur (**Processor Type, Architecture**), le serveur de fichiers distant (**File**), le nombre de processeurs de la machine (**Number of Processors**) et la possiblité de placer manuellement les processus (**Mapping on Processors**) sont instanciés automatiquement par l'environnement. Celui-ci dialogue avec le système distant à l'aide de primitives standard (UNIX) pour récupérer les informations nécessaires.

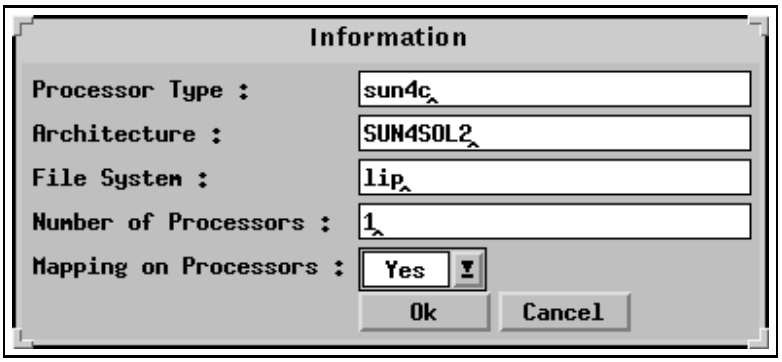


FIG. 7.13 - *Informations sur la machine ajoutée*

Bien entendu, si des informations se révèlent inexactes par suite d'une modification du système de la machine cible, l'utilisateur peut modifier manuellement les options détectées automatiquement.

Cet environnement est aussi conçu pour faciliter la portabilité du système DOSMOS entre différentes architectures logicielles et matérielles.

Gestion des alias de machines

Ces machines doivent être ensuite virtuellement fusionnées pour offrir une machine parallèle virtuelle (inspirée des travaux sur PVM [GBD+93]). Pour réaliser cela, nous utilisons un regroupement fondé sur des alias de machines qui regroupent un ensemble de machines.

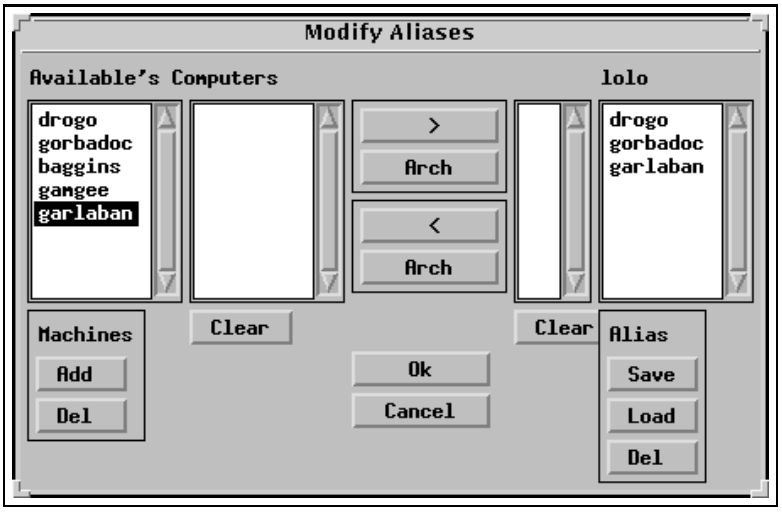


FIG. 7.14 - *La gestion des alias : ajouts et suppressions de machines*

Les alias permettent de regrouper facilement des ensembles de machines qui disposent de caractéristiques communes. L'utilisateur peut ainsi grouper les machines qui disposent d'une même architecture, les stations de travail dont la localisation géographique est proche ou les machines qui dépendent du même serveur de fichiers.

Ces alias sont gérés à l'aide de l'environnement qui permet de les sauvegarder et des utiliser (figures 7.15 et 7.16) pour la création de la machine virtuelle.

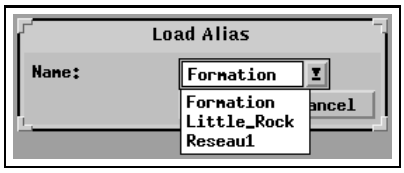


FIG. 7.15 - *Chargement d'un alias en vue d'une modification*

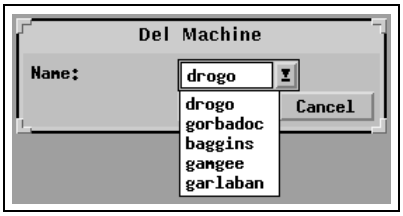


FIG. 7.16 - *Effacement d'un alias*

Création d'une machine virtuelle

Une machine virtuelle est un ensemble d'alias ou de machines indépendantes. Cet ensemble peut ainsi contenir des stations de travail locales, des réseaux indépendants de stations de travail ou des machines parallèles. Cette machine est représentée sous forme d'arbre visualisable par l'utilisateur

(figure 7.17). Cet arbre est construit graphiquement par l'utilisateur en fonction des machines dont il dispose. Il peut ensuite sauver cette arborescence pour l'utiliser dans diverses applications

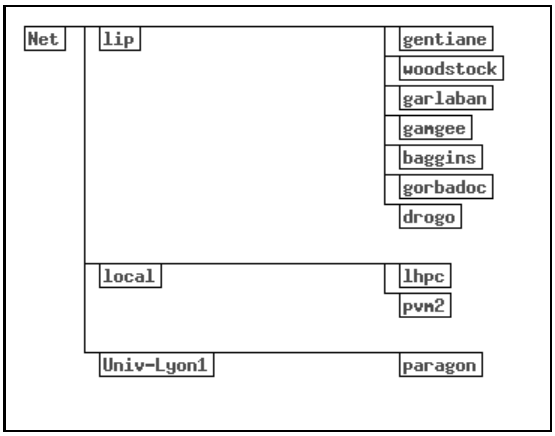


FIG. 7.17 - *Arborescence de la machine virtuelle*

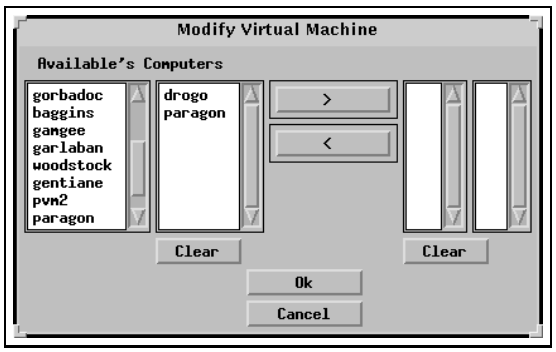


FIG. 7.18 - *Création d'une machine virtuelle*

La machine virtuelle étant créée, l'utilisateur peut définir l'ensemble des processus qu'il va utiliser au cours de son exécution.

7.2.2 Construction du graphe de processus

La conception d'une application DOSMOS repose sur l'utilisation de la modularité du système. Cette modularité est proposée à l'utilisateur sous la forme d'un graphe de processus. Le programmeur doit spécifier les processus utilisés par son application : Processus Application, Processus Mémoire, Processus Passerelle, Processus Gestionnaire d'Evènements. Les trois premiers types de processus ont déjà été définis dans le chapitre précédent. Le Processus Gestionnaire d'Evenements (E.M.P.) sera défini dans la section 7.3; il est utilisé pour la collecte des traces d'exécution des applications. L'utilisateur construit ainsi le graphe des processus et définit les relations entre ces processus (liens entre les PM et les PA, nombre de PA gérés par un PM, nombre de PP, liens entre les PM et les EMP...).

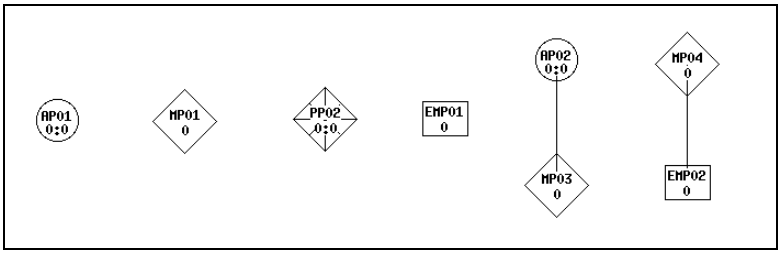


FIG. 7.19 - *Les types de processus disponibles pour la construction du graphe de processus*

Les notations graphiques utilisées permettent une construction intuitive du graphe de processus (figure 7.19). Les quatre types de processus utilisés pour la conception d'une application DOSMOS sont représentées dans le graphe : les Processus Application (**AP**), les Processus Mémoire (**MP**), les Processus Passerelle (**PP**) et les Processus Gestionnaire d'Evènements (**EMP**). Les relations

entre ces processus obéissent aux règles énoncées dans notre modèle de Mémoire Distribuée Virtuellement Partagée (section 5). Ainsi, un Processus Application ne peut être connecté qu'à un seul Processus Mémoire. Les Processus Gestionnaire d'Evènements ne dialoguent qu'avec des Processus Mémoire. Les valeurs numériques présentes dans les PA et les PP (**x:y**) identifient ces processus. La première valeur précise le numéro du groupe hiérarchique de plus haut niveau dans lequel est placé le processus. La deuxième valeur désigne le numéro logique qui identifie la machine sur laquelle sera exécuté le processus. Les Processus Mémoire et les Processus Gestionnaires d'Evènements ne sont réellement associés à aucun groupe. L'utilisateur doit seulement fournir le numéro de la machine sur laquelle ils seront exécutés. Des numéros de groupes ou de machines qui ont comme valeur 0 signifient que ces valeurs n'ont pas encore été instanciées par l'utilisateur.

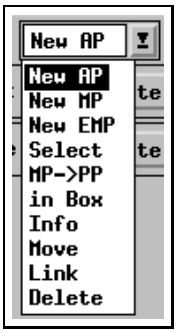


FIG. 7.20 - *Choix possibles pour la construction du graphe de processus*

L'utilisateur dispose de différents outils qui lui permettent de construire graphiquement le graphe de processus, de définir et de modifier la configuration des processus (figure 7.20) :

- Création de processus : cette option est utilisé pour la construction du graphe de processus : ajout de Processus Application (**New AP**), de Processus Mémoire (**New MP**) et de Processus Gestionnaire d'Evènements (**New EMP**) ;
- Configuration des processus : cela permet d'affecter des rôles à certains processus tels que la transformation d'un Processus Mémoire en Processus Passerelle (**MP->PP**) dédié à la gestion des groupes, et la vérification de la configuration des processus (**Info**) ;
- Liens inter-processus : l'option **Link** permet de vérifier la cohérence de la configuration proposée par l'utilisateur. Des validations sont effectuées pour vérifier la correction du graphe de processus (un PA n'est relié qu'à un PM, un PM communique avec plusieurs PA...)
- Fonctions de placement : elles permettent de gérer la visualisation du graphe de processus : déplacement des processus (**Move**), sélection individuelle (**Select**), sélection en groupe (**in Box**) , et effacement (**Delete**).

Les figures 7.21 et 7.22 présentent deux graphes de processus dédiés à des architectures différentes. La première figure propose un graphe de petite taille où la répartition des processus est parfaitement adaptée à une exécution distribuée ou répartie. La seconde figure laisse entrevoir les possibilités d'utilisation de notre environnement pour des machines parallèles. Le graphe, qui

comprend une soixantaine de processus, permet d'avoir une approche visuelle assez intuitive de la structure de l'application.

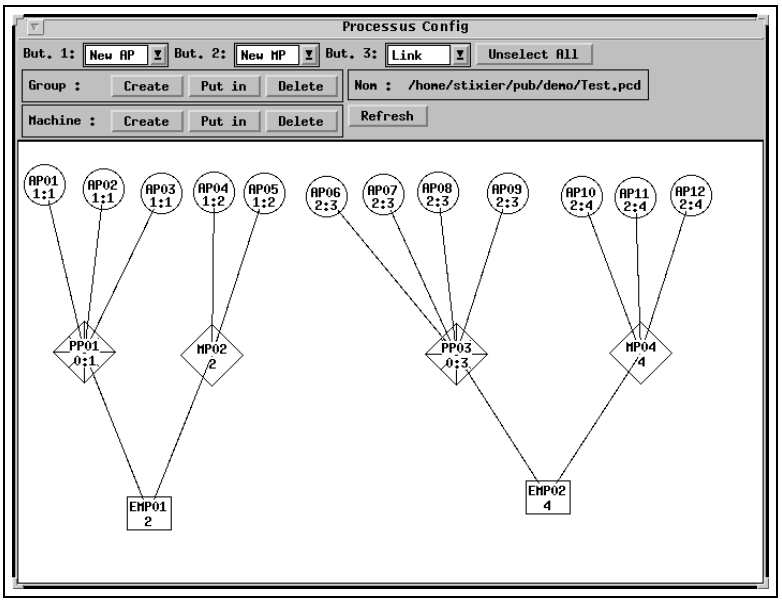


FIG. 7.21 - *Un graphe de processus dédiés à une exécution répartie*

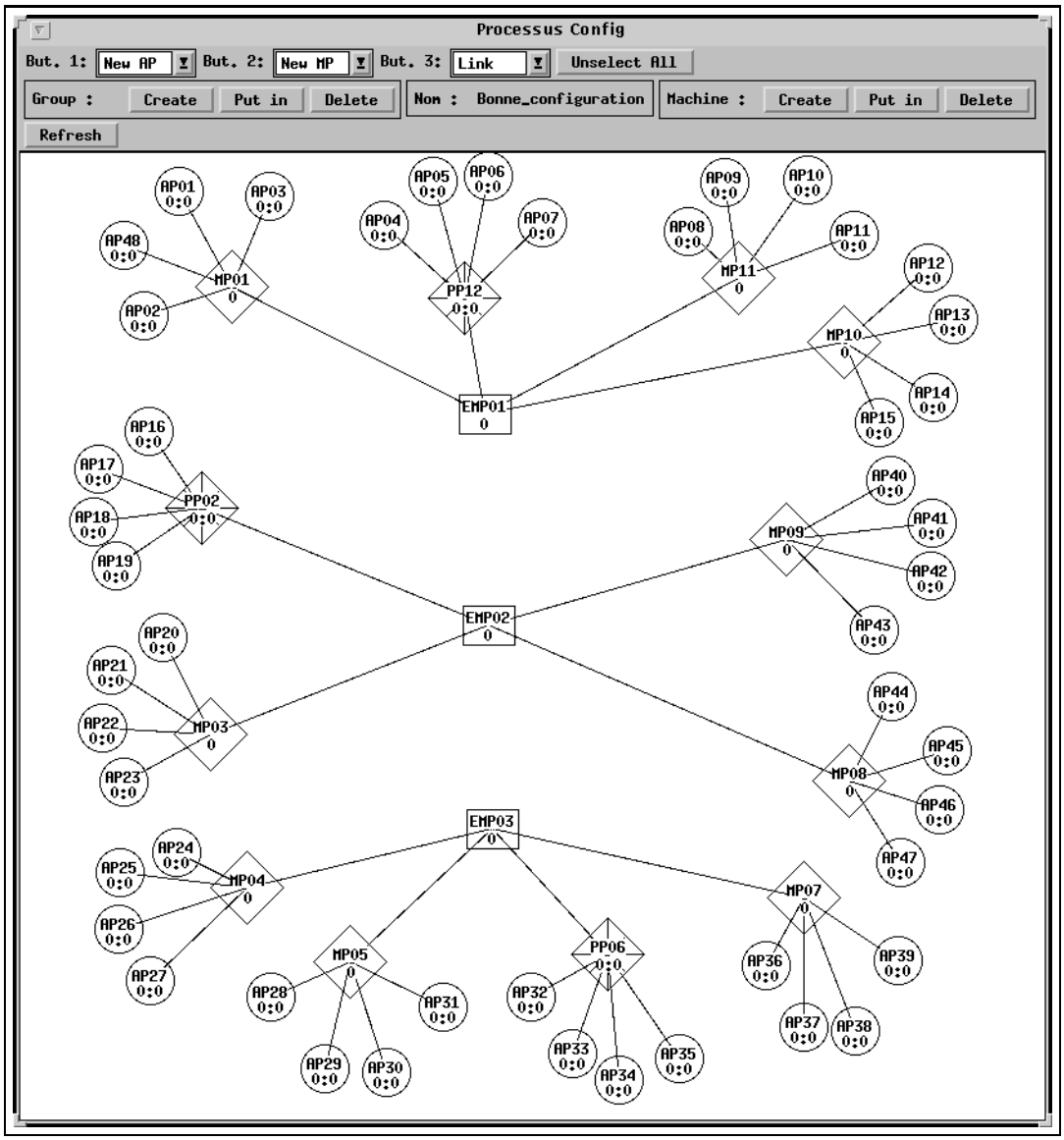


FIG. 7.22 - *Un graphe de processus dédié à une machine parallèle*

7.2.3 Représentation hiérarchique

La création des groupes hiérarchiques est une des étapes essentielles de la conception d'une application DOSMOS. La génération des groupes s'articule autour de deux étapes qui vont permettre à l'utilisateur de faire un lien entre le graphe de processus et l'application.

- Regroupement hiérarchique des processus : dans cette étape, l'utilisateur regroupe les processus qui ont des comportements de partage identiques. La construction des groupes hiérarchiques est réalisée de manière graphique et permet une vérification de la bonne hiérarchisation des groupes de partage (figure 7.23).

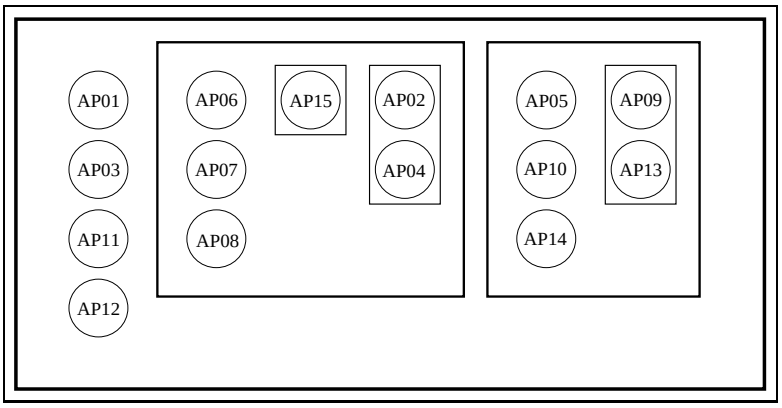


FIG. 7.23 - *Représentation des groupes hiérarchiques*

- Association d'objets partagés aux groupes : pendant cette étape, le code de l'application est analysé. La table des objets partagés en est extraite. Elle permet ainsi à l'utilisateur de contrôler le bon découpage de ses objets. L'utilisateur associe les groupes de processus identifiés par un numéro logique à des ensembles d'objets partagés. Des vérifications sont effectuées afin de contrôler la cohérence de cette structuration hiérarchique. Ainsi, un groupe ne peut exister sans contenir au moins un processus. De même, un groupe est obligatoirement lié à un (ou plusieurs) objets partagés.

7.2.4 Placement des processus

L'environnement propose à l'utilisateur de réaliser le plongement du graphe de processus dans la machine virtuelle.

L'utilisateur définit le placement de tous les éléments du graphe de processus sur les différents nœuds de la machine virtuelle. Une validité du placement des processus est effectuée en tenant compte des caractéristiques matérielles fournies par l'utilisateur (lors de l'ajout de machines; section 7.2.1).

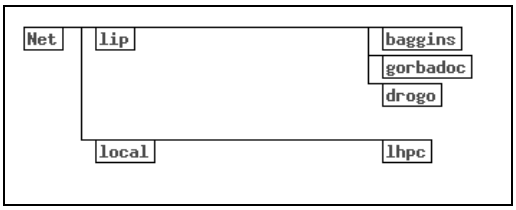


FIG. 7.24 - *Arborescence de la machine virtuelle*

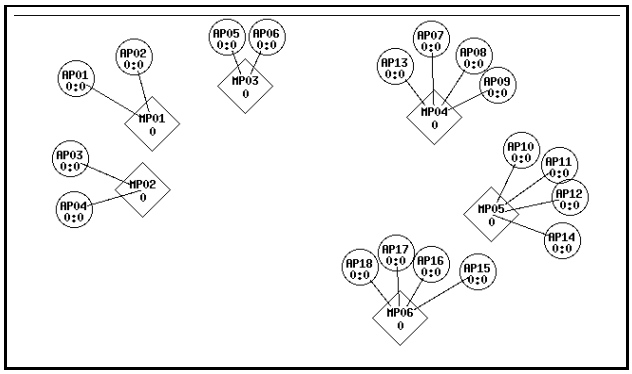


FIG. 7.25 - *Graphe des processus*

Prenons l'exemple d'une machine virtuelle regroupant 3 stations de travail (*baggins*,*gorbadoc*, *drogo*) et une machine parallèle (ici : la machine Matra Capitan (*LHPC*)). L'utilisateur a aussi défini le graphe des processus de son application (figure 7.25). il y a 18 Processus Application gérés par 6 Processus Mémoire. Certains PM sont associés avec une paire de PA, alors que les autres PM gèrent quatre PA chacun. Pour obtenir de bonnes performances, l'utilisateur désire placer ces derniers PM (et leurs PA associés) sur la machine parallèle, les autres processus étant répartis entre les stations de travail.

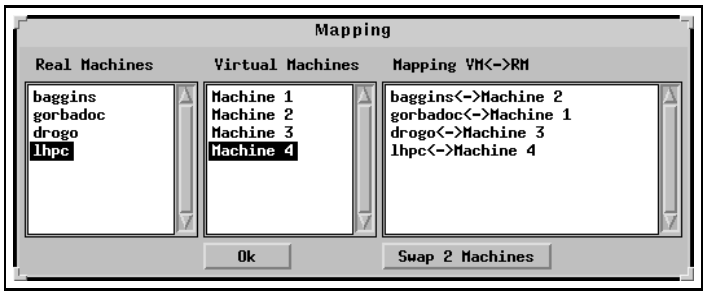


FIG. 7.26 - *Choix de la localisation effective des processus*

Afin d'automatiser cette étape, l'environnement propose des configurations de placement standard prenant en compte les spécificités de la machine virtuelle (un processus mémoire par processeur, un couple PM/PA par station par site...).

7.2.5 Compilation et exécution

Les informations collectées lors de l'ajout des machines sont utilisées pour compiler et exécuter les applications. Une application DOSMOS est, tout d'abord, transmise aux noeuds qui font partie de la machine virtuelle (figure 7.27). Puis, la compilation s'effectue de manière distribuée afin de respecter les contraintes logicielles et architecturales des différents sites composant la machine virtuelle. Ainsi, une même application DOSMOS peut être construite en parallèle à l'aide de plusieurs compilateurs.

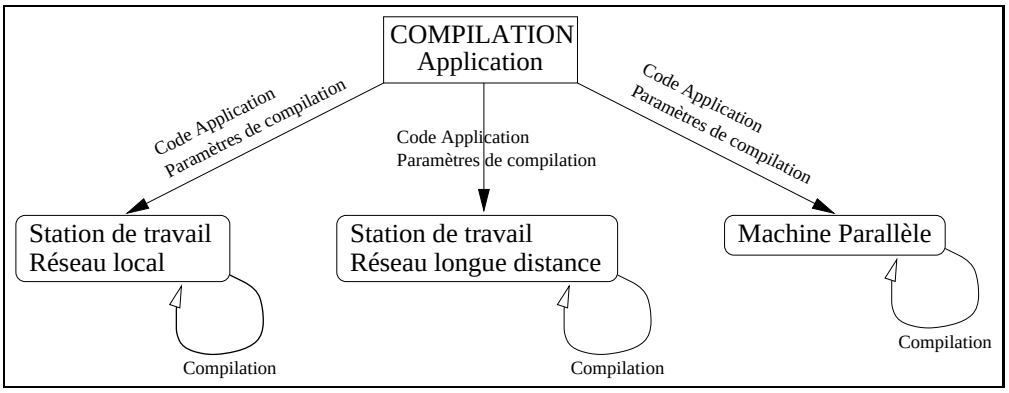


FIG. 7.27 - *Compilation Parallèle grâce à l'environnement*

Le lancement de l'exécution est réalisée de manière transparente pour l'utilisateur qui a l'impression de n'utiliser qu'une seule machine parallèle unifiée et homogène. L'environnement génère les processus sur les noeuds de la machine virtuelle. Le lancement de l'application est fondée sur l'utilisation de la couche PVM qui garantit la communication entre les sites distants.

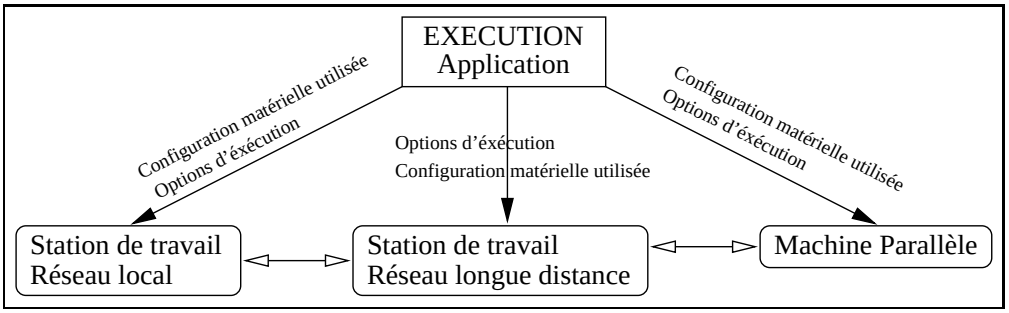


FIG. 7.28 - *Exécution Parallèle de l'application*

7.3 Visualisation et évaluation de performances

L'un des risques engendrés par l'utilisation d'un système de mémoire distribuée-partagée est que la simplification d'utilisation ne soit trop forte ! En effet, le système masquant toutes les communications de l'application, celle-ci peut apparaître comme une boîte noire. C'est pour cette raison que nous avons ajouté un environnement d'analyse d'exécution à notre système par l'intermédiaire de

l'outil DOSMOS-Trace. Celui-ci permet d'appréhender graphiquement le comportement des applications ainsi que leurs performances. Fondé sur un ensemble de processus collecteurs d'informations (les **Processus Gestionnaire d'Evènements : PGE**), cet outil sauvegarde les informations de traçage dans des fichiers distribués dans le système (utilisation optimisée des disques locaux existants (section 7.2.1)). La collecte et la visualisation des traces peut se faire de manière post-mortem ou en temps réel pendant l'exécution de l'application.

DOSMOS-Trace propose ainsi diverses vues de l'application (Figures 7.30 et 7.29) :

- Traçage des objets : le système fournit les informations relatives à la gestion des données partagées (propriétaire, taille des sous-objets...) afin de mieux appréhender les accès effectuées par le système DOSMOS. En outre, cet outil permet d'afficher de nombreuses statistiques sur les accès réalisés sur les objets de l'application (accès locaux, intra-groupe ou extra-groupe)(figure 7.29). Ainsi, l'utilisateur mesure globalement l'impact sur les performances de la répartition hiérarchique des processus.

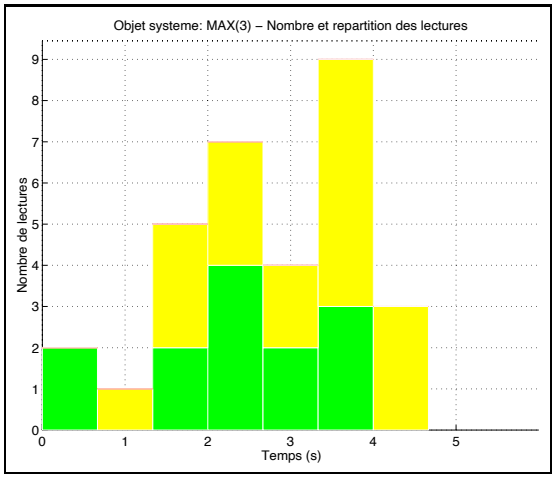


FIG. 7.29 - *Etude globale des opérations de lecture réalisées sur un objet : lectures locales, lectures intra-groupe et lectures extra-groupe.*

- Traçage des processus : DOSMOS-Trace permet d'afficher l'historique de l'exécution d'un processus donné afin de mieux comprendre son comportement pendant l'exécution de l'application (attentes sur des acquire, synchronisations...). Cet outil permet aussi de mesurer les performances de l'application en affichant des mesures de temps d'exécution précises (figure 7.30)

- Analyse des accès : ces informations permettent d'évaluer les performances de l'application et la répartition des accès sur les objets (accès intra ou inter-groupes, accès à l'aide d'*acquire*...) (voir figure 7.31).

L'outil graphique DOSMOS-Trace permet ainsi d'aider l'utilisateur à mieux appréhender le comportement de son application. L'ensemble des informations proposées permettent de détecter

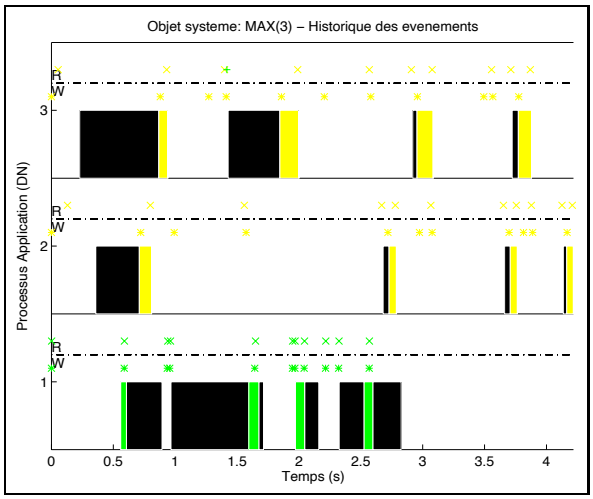


FIG. 7.30 - *Accès à un objet: chaque ligne représente les différents accès réalisés par un processus sur un objet donné: attente d'*acquire*, lecture locale, écriture locale, lecture intra-groupe...).*

```
Statistiques sur les lectures de l'objet systeme: MAX0(0)
total des lectures locales:              51 soit  85.00 %
total des lectures intra-groupe:          7 soit  11.67 %
total des lectures inter-groupe:          2 soit   3.33 %
total des lectures:                      60

Statistiques sur les ecritures de l'objet systeme: MAX0(0)
total des ecritures locales:              11 soit  18.03 %
total des ecritures intra-groupe:         30 soit  49.18 %
total des ecritures inter-groupe:         20 soit  32.79 %
total des ecritures:                      61

Statistiques sur les Acquire de l'objet systeme: MAX0(0)
Pas d'Acquire effectue sur cet objet.

Statistiques sur les Acquire en attente de l'objet systeme: MAX0(0)
Jamais d'Acquire en attente
```

FIG. 7.31 - *Statistiques d'exécution*

une vaste gamme des principaux problèmes rencontrés par un système de MDVP comme DOSMOS : goulots d'étranglement, mauvais découpage des objets, structuration hiérarchique inadaptée à l'application, inter-blocages...Cet environnement d'analyse d'exécution offre, de cette manière, des possibilités d'optimisation des applications facilement accessible à l'utilisateur.

Le traçage des applications, réalisé en collaboration avec O. Reymann [Rey94], n'étant pas le sujet principal de ce document, le lecteur pourra se reporter à l'article [BLR96] fourni en annexe.

7.4 Conclusion

L'utilisation de cet environnement pour une programmation parallèle à base de processus et d'objets partagés améliore la finesse de programmation des utilisateurs. Il permet de guider pas à pas les utilisateurs qui programment leurs applications au dessus du système DOSMOS (figure 7.32).

Les apports d'un tel environnement de programmation sont multiples et concernent toutes les étapes de la conception d'une application parallèle :

- La définition d'une interface unifiée permet une programmation assistée et intuitive de la structure de l'application à l'aide d'outils graphiques. Ainsi, cet environnement s'adresse à l'utilisateur débutant en lui permettant une parallélisation facile des applications DOSMOS ;
- Le portage du système est transparent pour l'utilisateur. Celui-ci peut ainsi facilement installer le système DOSMOS sur l'ensemble des plates-formes de développement mises à sa disposition L'utilisateur expert bénéficie ainsi des apports de cet environnement; il ne dépense plus inutilement son énergie à implémenter ses applications sur diverses architectures. En effet, le système est le garant de l'hétérogénéité de ses applications. ;
- La programmation d'une application fondée sur une MDVP s'apparente à une programmation séquentielle. Mais, pour atteindre des performances acceptables, l'utilisateur doit comprendre certains mécanismes sous-jacents qui peuvent briser le parallélisme de son application. En cachant toutes les communications, un système à MDVP peut ainsi apparaître comme une boîte noire. Afin d'aider les utilisateurs en quête de performances ou ceux désireux de comprendre les mécanismes internes d'une MDVP, un environnement d'analyse d'exécution est donc indispensable à un environnement de programmation complet. Pour compléter cette analyse d'exécution, des outils de déboggage sont actuellement en cours de réalisation.

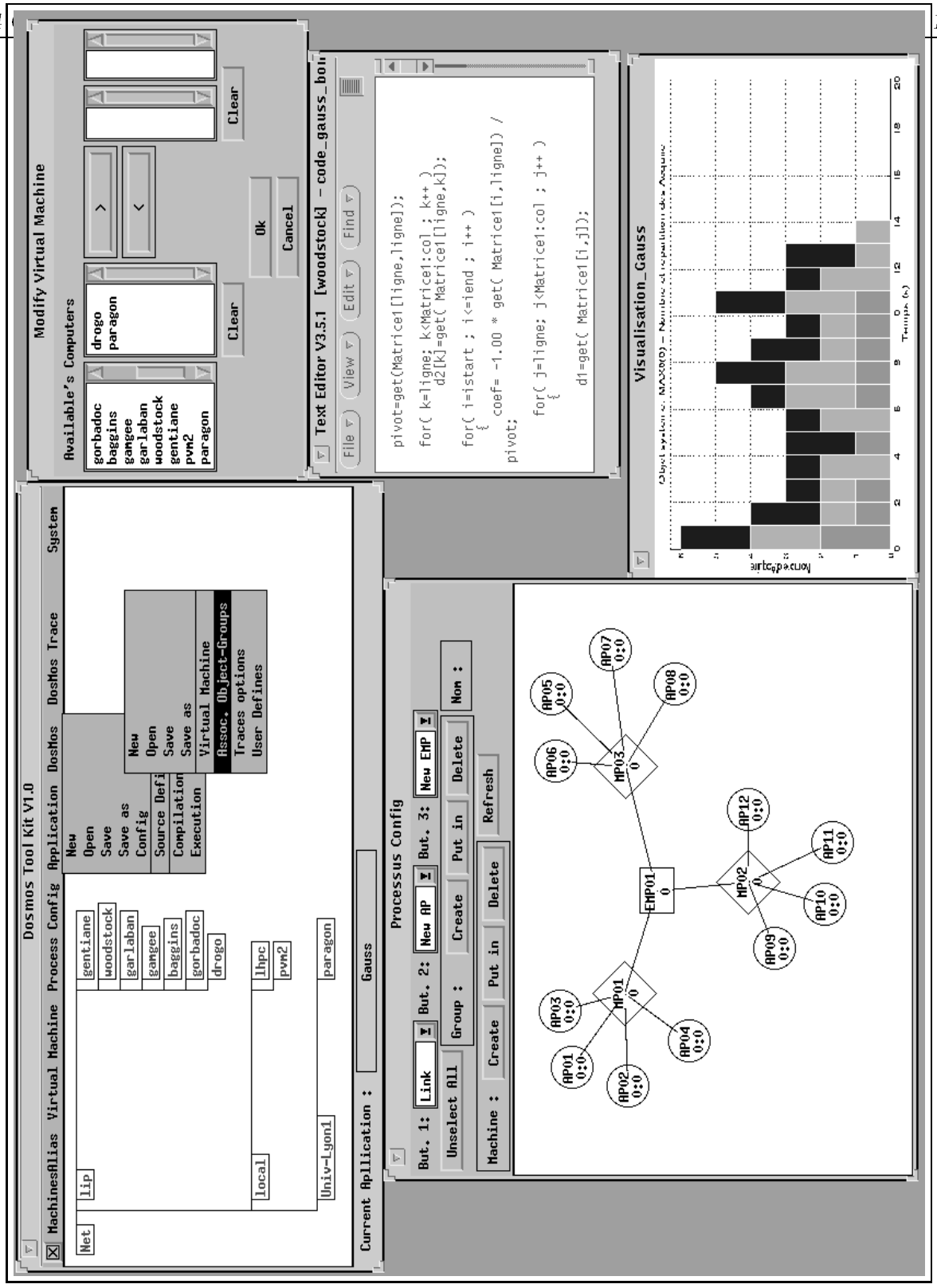


FIG. 7.32 - *Une vue complète de l'environnement de programmation*

CHAPITRE 8

Expérimentations

Différentes d'applications ont été expérimentées sur le système DOSMOS qui a été porté sur différentes architectures matérielles (réseau de stations, T3D, Matra Capitan...). Comme premières expérimentations, nous avons évalué notre système sur l'implémentation de nos concepts novateurs : les protocoles de cohérence faible, le découpage des objets et l'organisation en groupes. Ces diverses expérimentations nous permettent de tirer des conclusions sur l'efficacité et l'extensibilité des applications utilisant le système DOSMOS. Ce chapitre décrit une utilisation judicieuse de l'environnement de programmation, compare des applications utilisant le système DOSMOS ou PVM et présente les expérimentations en cours de développement.

8.1 Efficacité:

Pour évaluer l'impact de la cohérence faible à la libération, nous avons testé de petites applications sans utilisation de groupes hiérarchiques. Généralement, les principaux problèmes rencontrés par les systèmes de MDVP ont lieu quand plusieurs processus modifient en même temps la même donnée. Ainsi dans les exemples 8.1 et 8.2 nous utilisons jusqu'à 8 Processus Application qui accèdent au même objet partagé pour écrire une nouvelle valeur. Ces processus sont répartis sur 8 stations de travail SUN (Sparc Station 5) reliées par un réseau Ethernet.

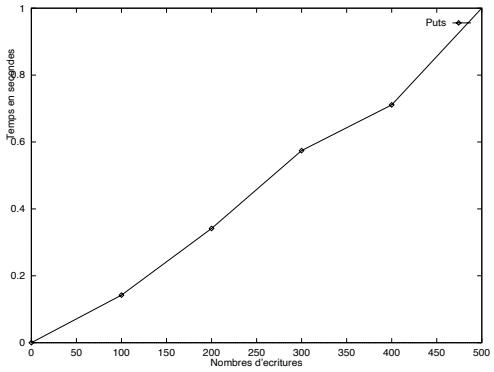


FIG. 8.1 - *Accès en écriture aux objets partagés*

Dans la figure 8.1, nous voyons que la courbe est quasi-linéaire. Ces résultats montrent que le

système DOSMOS n'est pas surchargé quand beaucoup d'accès sont réalisés en parallèle (on fait 500 écritures en moins d'une seconde).

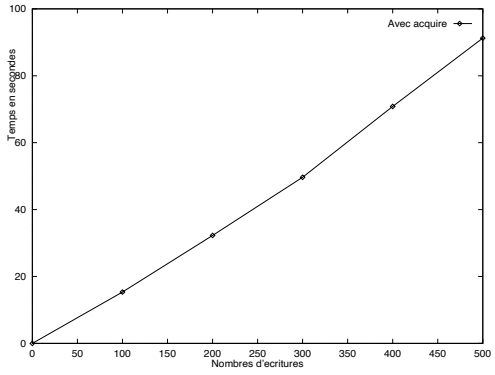


FIG. 8.2 - *Ecritures avec appels* acquire/release

Nous réalisons le même test pour la figure 8.2, mais avant chaque écriture sur l'objet les processus appellent la primitive *acquire* pour verrouiller l'objet au processus écrivain. Le Processus Application modifie la valeur de l'objet et appelle la primitive *release* pour le relâcher. Dans la courbe 8.2, on peut observer que la progression des résultats est aussi linéaire, le système continue à traiter les accès les uns après les autres. On peut se rendre, tout de même, compte que malgré la faiblesse de la cohérence à la libération *release* , le fait d'acquérir un objet a un coût non-négligeable dont l'utilisateur doit tenir compte quand il programme ses applications.

8.2 Extensibilité :

Dans les deux exemples suivants, nous utilisons un réseau de stations de travail (jusqu'à 8 Sun Sparc station IPX et ELX) qui exécutent chacun un Processus Mémoire et un Processus Application accédant à un objet partagé complexe.

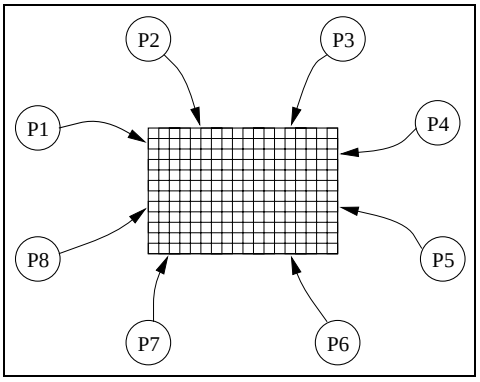


FIG. 8.3 - *Les processus accèdent au même objet*

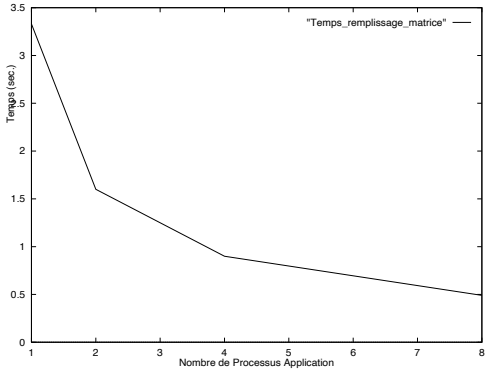


FIG. 8.4 - *Accès concurrents à un objet partagé : remplissage d'une matrice*

La figure 8.4 affiche les temps d'exécutions d'une application où les processus remplissent et

modifient les éléments d'une matrice distribuée. Ces modifications sont réalisées de manière concurrente en utilisant les protocoles de cohérence faible.

La matrice accédée est découpée en blocs afin d'éviter des goulots d'étranglement lors de la modification de l'objet. L'augmentation du nombre de processeurs (qui exécutent concurremment un PA et un PM) ne réduit pas fortement l'efficacité des processus. On observe encore un facteur d'accélération (speedup) proche de 6 lorsque la machine virtuelle comporte 8 Processus Application.

Dans l'expérimentation suivante (figure 8.5), chaque Processus Application accède en lecture à une partie d'un tableau de taille importante et partagé par tous les processus. Il parcourt le tableau et recherche un maximum local qu'il stocke dans une variable locale. Après le parcours le maximum local est placé dans un objet partagé afin qu'il soit connu par tous les processus.

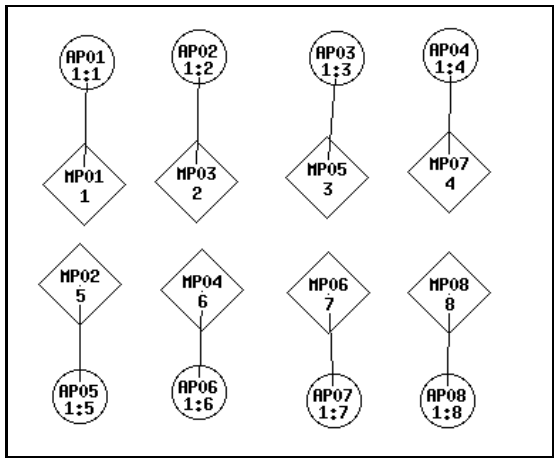


FIG. 8.5 - *Huit paires de processus réparties sur 8 stations de travail*

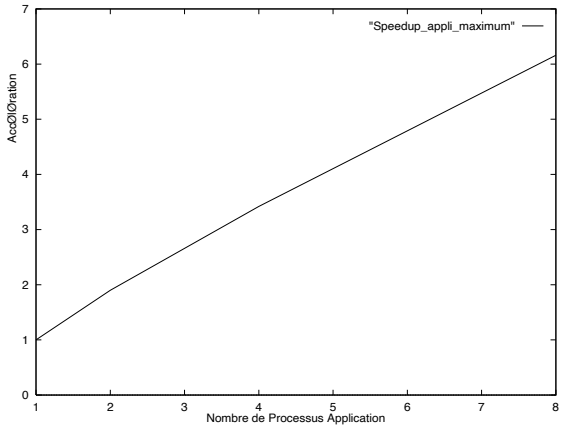


FIG. 8.6 - *Facteur d'accélération d'une application qui calcule l'élément maximum d'un tableau distribué*

L'expérimentation 8.6 décrit les facteurs d'accélération d'une application où les processus recherchent l'élément maximum d'un tableau distribué sur l'ensemble des processus. On obtient ainsi un très bon facteur d'accélération de 6.16 en utilisant 8 processeurs. L'accroissement de la courbe s'aplanit faiblement lorsque l'on augmente le nombre de Processus Application.

Ces différentes expérimentations, malgré la difficulté de l'accès à un unique objet partagé, permettent d'obtenir une extensibilité quasi-linéaire ce qui confirme l'utilité des concepts proposés par DOSMOS (modularité et cohérence faible).

8.3 Découpage d'objets

La possibilité de découper les objets volumineux est une fonctionnalité importante apportée par le système DOSMOS. Elle permet de manipuler de plus petits objets système et de limiter ainsi la taille des communications circulant dans le réseau. Les deux expérimentations suivantes (Cholesky et Gauss) s'interrogent sur l'influence du paramètre de découpage pour les performances

des applications.

8.3.1 Factorisation de Cholesky

Le propos de la factorisation de Cholesky est de factoriser une matrice symétrique positive de taille $n*n$ sous la forme $L.L^T$ où L est une matrice triangulaire inférieure. Ce calcul numérique est considéré comme un goulot d'étranglement dans les applications numériques parallèles.

Les tableaux 8.7 et 8.8 présentent les temps d'exécution d'une factorisation de Cholesky sur un ensemble de 8 Processus Application gérés par 4 Processus Mémoire, chaque processus étant placé sur une machine indépendante.

Le premier tableau présente une factorisation de Cholesky sur une matrice de taille modeste $(12*12)$. On observe que le découpage de cette matrice en 9 blocs de taille identique $(4*4)$ permet néanmoins d'améliorer les accès à la matrice en entrée. Ainsi, même sur une matrice de petite taille le découpage en sous-objets est bénéfique pour les performances d'exécution.

Taille Matrice	Nombre de blocs	Temps d'exécution (s)
$12 * 12$	1	9.3
$12 * 12$	9	2.4

FIG. 8.7 - *Temps d'exécution de Cholesky avec DOSMOS*

Le tableau 8.8 propose les résultats pour la même expérimentation mais avec une matrice en entrée de taille plus importante (100*100). D'une manière identique à l'expérimentation précédente, plus la matrice est découpée, plus les performances de l'application s'améliorent (45s avec des blocs de 33*33 éléments). Par contre, un trop fort découpage peut nuire aux performances de l'application. Si la taille des objets devient trop petite (un élément), les Processus Application ne profitent plus de la lecture groupée des éléments. Ils accèdent et réservent chaque élément les uns après les autres. Le système doit donc gérer de nombreux petits messages qui ralentissent les accès aux objets. Les expérimentations concernant la triangularisation de Gauss confirmeront ce problème du fort découpage.

Taille Matrices Entrée	Nombre de blocs	Temps d'exécution (s)
$100 * 100$	1	1800
$100 * 100$	4	111
$100 * 100$	9	45
$100 * 100$	100	336

FIG. 8.8 - *Temps d'exécution de Cholesky avec DOSMOS*

La figure 8.9 présente un extrait du code DOSMOS utilisé pour calculer la factorisation de Cholesky en parallèle.

```
shared double Matrice[100,100](10,10);
shared int Schedule_map[2,2];
main()
{
  use_dosmos();
  ...
  for (j=0; j<Schedule_map:col; j++)
    for (i=0; i<Schedule_map:row; i++)
      {if (i<j)
         { acquire(Matrice[i*taille_bloc,j*taille_bloc]);   /* J'écris 0 dans le bloc */
           for (r=0; r<taille_bloc; r++)
             for (c=0; c<taille_bloc; c++)
              Matrice[i*taille_bloc+r,j*taille_bloc+c]=0;
           release(Matrice[i*taille_bloc,j*taille_bloc]);
         }
       else
         { acquire(Schedule_map[i,j]);            /* Je cherche un bloc à calculer */
           if (get(Schedule_map[i,j])==0)
             { Schedule_map[i,j]=1;         /* J'indique que je calcule ce bloc */
              release(Schedule_map[i,j]);
              Transfer_Block(i,j,bloc);
              Compute_Block(i,j,bloc);
              Write_Block(i,j,bloc);            /* Met à jour l'objet Matrice */
              acquire(Schedule_map[i,j]);  /* Indique que le bloc est calculé */
              Schedule_map[i,j]=2;
              release(Schedule_map[i,j]);
             }
           else
              release(Schedule_map[i,j]);
         }
      }
  if (DN==1)
    { for (i=0; i<Matrice:row; i++)         /* Affichage du resultat par le processus 1 */
       { for (j=0; j<Matrice:col; j++)
           printf("%6.3f ",Matrice[i,j]);
         printf("\n"); }
      for (i=0; i<Schedule_map:row; i++)
       { for (j=0; j<Schedule_map:col; j++)
           printf("%2d ",Schedule_map[i,j]);
         printf("\n");
       }
    }
  end_dosmos();
}
```

FIG. 8.9 - *Code DOSMOS pour le calcul de Cholesky*

8.3.2 Triangularisation de Gauss

Le tableau 8.10 présente les temps d'exécution d'une triangularisation de Gauss réalisée sous DOSMOS à l'aide 5 Processus Application. Cette application est basée sur un code DOSMOS bien programmé (voir paragraphe 8.7) qui minimise les appels aux primitives de cohérence faible.

On peut se rendre compte que le temps d'exécution de la triangularisation de Gauss augmente d'une manière similaire à la taille du problème. L'application ne ressent pas les effets de goulots d'étranglement, dûs à l'accès à la matrice unique, grâce au découpage de la matrice d'entrée en sous-objets indépendants.

Taille de la matrice	Temps d'exécution
$10 * 10$	1.05
$20 * 20$	6.6
$30 * 30$	13.1

FIG. 8.10 - *Temps d'exécution d'une triangularisation de Gauss programmé sous DOSMOS*

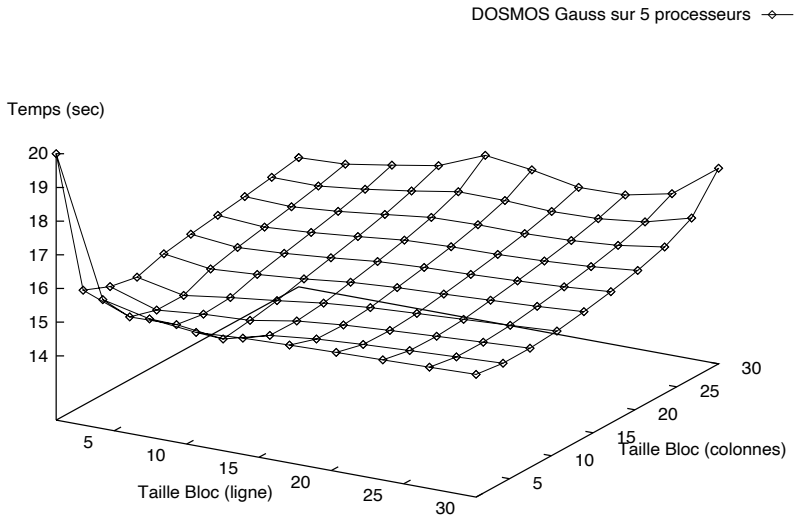


FIG. 8.11 - *A la recherche de la taille de bloc optimale pour un Gauss* $30 * 30$

Par contre, obtenir un temps de 13s pour un calcul de Gauss $30 * 30$ éléments nécessite un découpage des objets approprié. La figure 8.11 décrit les grandes variations de temps d'exécution suivant le découpage de la matrice d'entrée. Si la matrice n'est pas découpée, le temps d'exécution augmente car les accès à la matrice se traduisent par la manipulation et l'envoie de la matrice complète (temps de 19.5s). A l'inverse, choisir une taille de bloc trop petite est aussi lourde de conséquences sur les performances (temps de 20s). Dans cet exemple, le compromis se situe dans le choix de blocs de taille moyenne. Ainsi, des blocs de taille $2 * 5$ éléments permettent d'atteindre

les meilleurs temps d'exécution (13s) pour la triangularisation de Gauss.

Enfin, une structuration des Processus Application doit être effectuée pour bénéficier des groupes hiérarchiques de processus fournis par DOSMOS et qui permettent de réduire les coûts du maintien de la cohérence.

8.4 Structuration en groupes :

8.4.1 Calcul de π

Comme application test, nous parallélisons le calcul de la valeur Π sur un réseau de stations de travail. Une manière commune de calculer π peut être la suivante. Par définition :

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

Une méthode simple pour estimer cette intégrale est de discrétiser l'intervalle [0,1] en n petits intervalles de largeur $\frac{1}{n}$. Ainsi, la contribution de l'intervalle est approximée par $\frac{1}{n} * \frac{1}{1+x_i^2}$ où x_i représente le centre de l'intervalle i.

Nb. Processus	Nb. groupes	PA / Processeur	Temps de PI
12	0	1	3.95
12	2	1	2.56
12	2	6	1.85
12	2	3	0.78

FIG. 8.12 - *Resultats de π (en sec.)*

La figure 8.13 présente le code de π sans adjonction de groupes; chaque processus calcule sa propre partie de la valeur de π et réalise un acquire sur l'objet π pour ajouter sa contribution. Mais nous avons aussi implementé cette application avec différentes configurations de groupes et de nombre de processus par processeurs, pour tester l'impact de l'ajout de groupes sur ce type d'application. Dans nos expérimentations (Figure 8.12), nous utilisons 12 processus application, avec leur PM dédié, qui calculent concurrement la valeur de π.

Comme premier test nous lançons le calcul de π sans ajout de groupes sur 12 processeurs (temps de 3.95 secondes). Avec la même configuration, en ajoutant simplement une organisation en deux groupes, on obtient un meilleur temps de 2.56 s. En combinant le nombre de groupes avec le nombre de processus par processeur nous avons obtenu l'excellent temps de 0.78 s (avec 2 groupes de processus répartis sur 4 processeurs). Mais on se rend aussi compte que les performances peuvent chuter si les processeurs sont surchargés de travail (on obtient ainsi un temps d'exécution de 1.85s lorsque l'on place 6 PA et leurs PM associé sur chaque processeur).

Ainsi, en utilisant les groupes hiérarchiques, nous pouvons obtenir de meilleures performances, mais l'utilisateur doit faire un compromis entre la charge du processeur et le regroupement des

processus pour ne pas dimininuer les performances de ses applications.

```
#include ''Dosmos.h''
main()
{ int i,j,k,t;
  float s,n,inc,a;
  shared float pi;
  Use_Dosmos();
  pi=0.0;
  s=0.0; n=999.0; t=n/3;
  a=1.0/(2.0*n);
  for (i=t*(DN-1);i<(t*DN-1)+t;i++)
      { inc =(float)(i/n)+a;
        s=s+float(1.0/(1.0+(inc*inc)))/n;
      }
  acquire(pi);
  pi=4.0*s+pi;
  release(pi);
  proc_synchro();
  if (DN==0)
    printf(''La valeur de Pi est : %f\n'',pi);
  End_Dosmos();
}
```

FIG. 8.13 - *Application DOSMOS π. La variable* DN *représente le numéro du processus courant.*

8.4.2 Sous-réseaux

L'expérimentation suivante a pour but de tester le groupement hiérarchique des processus avec une application s'exécutant sur deux sous-réseaux de processus. Ces expérimentations ont été réalisées sur une machine virtuelle de 10 processeurs (figure 8.14) répartis sur deux sous-réseaux contenant chacun 5 processeurs (Sun Sparcstation 5, IPX, ELX). Chaque processeur exécute concurremment un Processus Application et son Processus Mémoire.

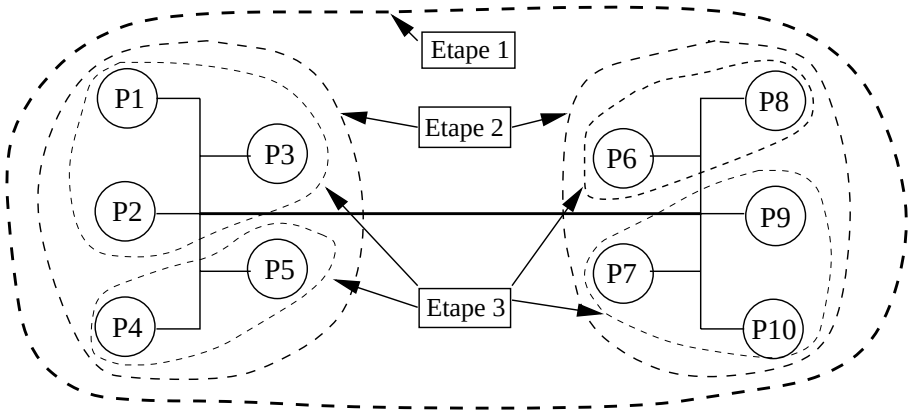


FIG. 8.14 - *Groupes de processus placés entre deux sous-réseaux*

Dans chaque test, les processus accèdent intensivement les objets partagés en réalisant 200 opérations de lecture et écriture sur les objets qu'ils partagent. Les expérimentations suivantes ont lieu selon trois étapes. A la première étape, tous les processus partagent le même objet (une matrice

d'entiers). Ils sont donc tous inscrits au même groupe. A l'étape 2, l'utilisateur se rend compte, qu'en fait la moitié des processus partagent une moitié de l'objet alors que les autres processus partagent l'autre moitié. Les processus sont donc groupés en deux groupes distincts placés sur chacun des sous-réseaux. A la troisième étape, l'utilisateur sépare encore plus finement ses processus en créant quatres de processus qui partagent chacun un quart de la matrice.

Accès simples

Le tableau 8.15 décrit les temps d'exécution lorsque les processus accèdent aux objets sans utiliser la cohérence à la libération. Cette utilisation est possible si l'utilisateur connaît bien son application et s'il peut s'assurer que les modifications ne se recouvrent pas. La'jout de groupes permet d'améliorer les performances de l'application (temps d'exécution diminué de moitié lorsque l'on divise par 2 le nombre de processus dans le groupe).

	Nb. Processus	**Nb. groupes**	**Temps Exécution**
Etape 1	10	1	3.8
Etape 2	10	2	1.45
Etape 3	10	4	0.8

FIG. 8.15 - *Accès simples avec des groupes de processus*

Accès avec Acquisition

Dans les expérimentations suivantes (figure 8.16), chaque processus réalise ses accès à la matrice partagée en utilisant le modèle de cohérence à la libération. Il entoure chacune des écritures à la matrice d'une paire de primitives *acquire* et *release*. On se rend compte ici, que le fait de localiser les verrouillages d'objets sur peu de processus à la fois améliore fortement les accès aux objets. Les temps d'exécution sont diminués par un facteur 3, lorsque l'on sépare les processus en groupes placés sur chaque sous-réseau.

	Nb. Processus	**Nb. groupes**	**Temps Exécution**
Etape 1	10	1	15
Etape 2	10	2	6.2
Etape 3	10	4	3.1

FIG. 8.16 - *Accès en cohérence faible avec des groupes de processus*

8.5 Objets partagés et échange de messages

Pour choisir son modèle de programmation, l'utilisateur doit deviner celui qui s'adaptera le mieux à son application : mémoire distribuée virtuellement partagée ou échange de messages. Nous allons, tout d'abord, comparer les modèles de programmation DOSMOS et PVM sur les accès qu'ils proposent en lecture et en écriture. Puis, nous généralisons cette comparaison sur de petites applications. Le système DOSMOS étant mis entre les mains de divers utilisateurs (débutants en parallélisme ou habitués de PVM), nous en tirons les conséquences nécessaires quant à l'amélioration du système.

Lectures et écritures de données

Dans le système DOSMOS, chaque Processus Application communique avec son Processus Mémoire pour accèder aux objets partagés. La procédure équivalente en PVM serait la suivante : un processus envoie un message à l'un de ses voisins et attend qu'il lui renvoie une valeur. La figure 8.17 compare les accès en lecture en DOSMOS et en PVM. On se rend compte que les lectures avec DOSMOS sont legèrement plus lentes avec DOSMOS à cause de la couche logicielle de la gestion de la mémoire virtuellement partagée. Mais les différences entre les deux systèmes sont faibles face aux différences d'effort de programmation. En PVM, il faut une vingtaine de lignes de code pour réalisr une lecture (empaqueter les données, gérer les buffers, envoyer, recevoir...), alors qu'avec DOSMOS la lecture est réalisée en deux lignes de code.

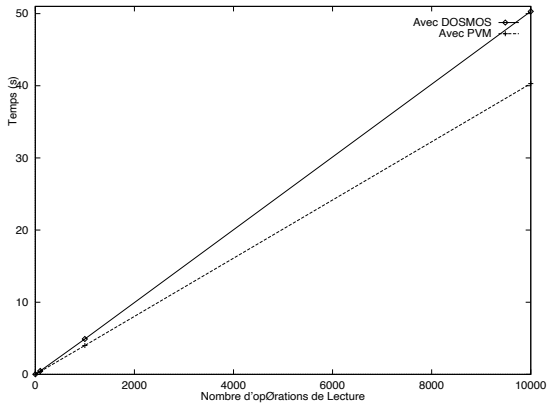


FIG. 8.17 - *Opérations de lectures DOSMOS et PVM avec 5 processus*

Lors d'écritures dans une donnée distante réalisée par DOSMOS, le processus profite du fait qu'il dispose d'une copie locale pour paralléliser les modifications d'objets (*écrivains multiples*). Une procédure équivalente en PVM ne bénéficie pas de la localité d'une copie et doit générer un message pour chaque accès en écriture. C'est la raison pour laquelle les écritures sous DOSMOS sont largement plus performantes que celles sous PVM (figure 8.18). La modification relâchée des objets permet aux applications DOSMOS de réaliser 1000 écritures en moins de deux secondes.

Ces expérimentations montrent que les deux modèles de programmation peuvent être comparés

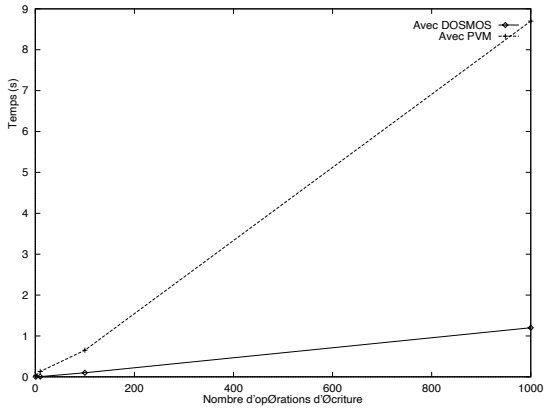


FIG. 8.18 - *Opérations d'écriture DOSMOS et PVM avec 5 processus*

sur les accès basiques aux donnés distantes.

Calcul de π en DOSMOS ou en PVM

L'expérimentation 8.20 permet de comparer les facteurs d'accélération de l'application π (Figure 8.13) implémentée en échange de messages avec PVM puis en DOSMOS. Les résulats montrent que le surcoût d'exécution engendrée par la couche DOSMOS est négligeable face à la difficulté d'implémentation en PVM pur. De plus avec un facteur d'accélération de 6.4 pour 8 processeurs, on s'aperçoit que les performances du système DOSMOS ne s'écroulent pas lorsque l'on augmente le nombre de processeurs.

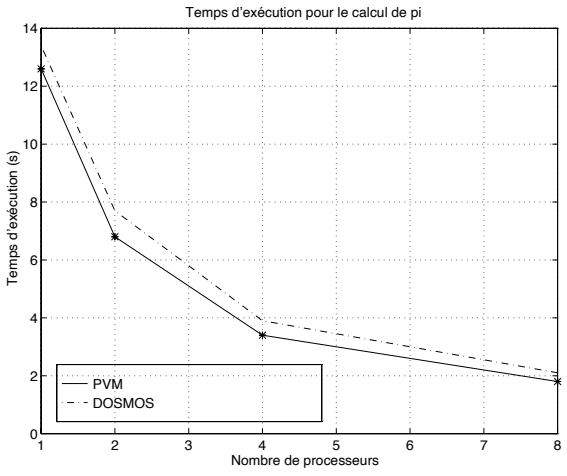


FIG. 8.19 - *Temps d'exécution de l'application DOSMOS π*

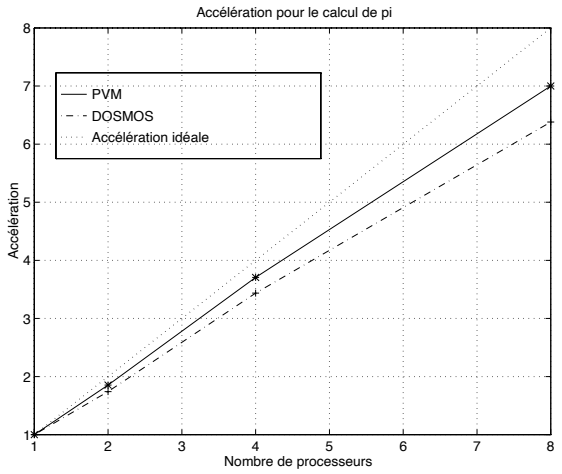


FIG. 8.20 - *Comparaison des facteurs d'accélération de l'application π en échange de messages avec PVM ou en mémoire partagée avec DOSMOS avec 8 processeurs.*

Zbuffer

Nous avons utilisé notre système dans un cas de ré-engineering d'une application PVM existante : une application d'imagerie parallèle à base d'échange de messages : un algorithme de Zbuffer parallèle et équilibré [CLM95]. Cette parallélistion équilibrée est fondée sur des techniques d'équilibrage de charge proposées par [MR91]. Cette ré-écriture partielle du code à porté sur l'ajout d'objets partagés pour le calcul d'équilibrage de charge.

Chaque processeur exécute en parallèle le Zbuffer programmé en PVM de la figure 8.21.

L'ajout de code DOSMOS porte sur la partie de code concernant les calculs et communications nécessaires à l'équilibrage de charges (figure 8.22). Il faut plus de 70 lignes de code pour réaliser le calcul de charge en échange de messages (réservation des zones mémoire, synchronisation, codes des séquences d'émission et de réception des données...) Or le calcul de la charge représente moins de 1% du temps d'exécution total. Il faut donc réaliser un effort considérable d'implémentation avec la programmation de communications globales (diffusion, réduction). L'insertion de code DOSMOS permet un apport à deux niveaux. Grâce à l'utilisation d'objets partagés le calcul de charge est considérablement facilité (avec moins de 10 lignes de code). Les performances et le comportement de l'application sont respectés car le calcul de la charge n'est pas une opération coûteuse comparée à l'ensemble des calculs et des communications nécessaires à un Zbuffer parallèle. De plus, l'insertion de primitives DOSMOS au milieu du code à base d'échange de messages ne remet pas en cause le reste du code qui peut rester inchangé.

En proposant de mélanger de l'échange de messages et de la MDVP au sein d'une même application, notre système s'adapte ainsi au comportement et aux habitudes de l'utilisateur en vue de faciliter la programmation de grosses applications.

Programmation en échange de messages (PVM) :

Distribution des facettes et des sommets
Pour chaque image à générer **Faire**
Début

Projection des sommets
Multi-diffusion des sommets projetés
Multi-réduction *pour le calcul des charges de travail de l'image*
Multi-diffusion *des charges locales aux autres processus*
Synchronisation *des processus*
Calcul *des charges*
Multi-diffusion des nouveaux indices de répartition
Multi-distribution des facettes pour les répartir entre les processus
Zbuffer séquentiel sur la partie de l'image dont s'occupe le processeur
Sauvegarde de l'image

Fin

FIG. 8.21 - *Zbuffer Parallèle programmé en échange de messages*

Programmation en échange de message + objets partagés (PVM + DOSMOS) :

Déclaration d'un objet partagé de charge globale et vecteur partagé des charges locales
Distribution des facettes et des sommets
Pour chaque image à générer **Faire**
Début

Projection des sommets
Multi-diffusion des sommets projetés
Ecriture dans l'objet charge globale après acquisition libération
Ecriture dans le vecteur des charges locales
Calcul parallèle de la charge
Synchronisation des processus
Multi-distribution des facettes pour les répartir entre les processus
Zbuffer séquentiel sur la partie de l'image dont s'occupe le processeur
Sauvegarde de l'image

Fin

FIG. 8.22 - *Programmation avec échange de messages et objets partagés*

Choisir entre DOSMOS et PVM

Après ces quelques expérimentations, nous avons pu constater différents réactions en provenance des utilisateurs de notre système (figure 8.23) :

- La programmation sous DOSMOS est plus facile qu'avec PVM et le code parallèle est proche de la version séquentielle ;
- Concevoir et programmer une application parallèle avec une librairie d'échange de messages telle que PVM prend souvent 4 fois plus de temps qu'avec un modèle de MDVP comme DOSMOS ;
- Des applications déjà programmées avec PVM peuvent être facilement portées et transformées pour benéficier d'objets distribués virtuellement partagés en profitant de la couche d'implémentation (à base de PVM) sur laquelle est fondé DOSMOS ;
- Le code d'une application est, en moyenne, réduit de 50% en utilisant le système DOSMOS, comparé à PVM. La maintenance et la recherche d'erreurs en sont facilités ;

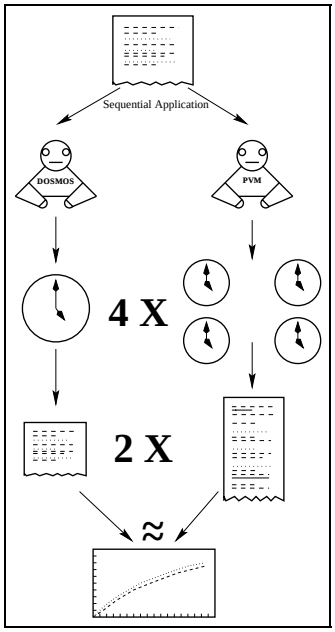


FIG. 8.23 - *La programmation d'une application parallèle: DOSMOS ou PVM?*

- La compréhension du comportement et l'optimisation des performances des applications sont plus faciles avec PVM pour un programmeur expérimenté. L'utilisateur débutant apprécie les services fournis par l'environnement de programmation de DOSMOS plutôt que la quête de performances ;
- DOSMOS est plus adapté à des applications à gros grain ou irrégulières tandis que PVM s'adapte mieux à des algorithmes parallèles réguliers ;

- L'environnement de programmation améliore fortement le temps de développement grâce à l'apport des outils graphiques qui permettent un placement rapide des tâches ;
- Les différences de performances entre PVM et DOSMOS sont légères si l'utilisateur s'investit autant dans sa programmation pour les deux modèles de programmation.

En analysant le comportement des utilisateurs implémentant des applications DOSMOS nous avons pu détecter les avantages et les inconvénients de notre système. Nous avons donc pu compléter notre environnement de programmation par l'ajout d'outils nécessaires à l'utilisateur pour développer ses applications.

8.6 Parallélisation de réseaux de neurones

Les réseaux de neurones artificiels sont de plus en plus utilisés dans les applications courantes. Bien implémentés en séquentiel, leur parallélisation à l'aide d'échange de messages est une tâche difficile avec des résultats incertains. Cette contrainte réduit le développement des applications parallèles dans le domaine des réseaux de neurones. Un environnement de programmation associé à une MDVP pourrait permettre une parallélisation facile d'applications à base de réseaux de neurones. Nous étudions actuellement diverses techniques pour implémenter facilement des réseaux de neurones sur des machines parallèles ou réseaux de stations. Ce paragraphe présente les travaux préliminaires mis en place pour la parallélisation de réseaux de neurones avec le système DOSMOS.

Parallèlisme intrinsèque

Les réseaux de neurones naturels sont de facto parallèles. Toutes les cellules travaillent simultanément. Ce parallèlisme intrinsèque pourrait s'appliquer aux réseaux de neurones artificiels. Il faudrait disposer d'architectures parallèles à grain très fin, c'est à dire de machines parallèles fournies avec de nombreux petits processeurs travaillant de manière synchrone et reliés par des réseaux d'interconnexion rapides (machines SIMD).

Parallélisation à gros grain

L'utilisation de machines MIMD nécessite une parallélisation à gros grain. En résumé on peut considérer un réseau de neurones comme un ensemble de cellules (qui contiennent un poids). Une base d'exemples à reconnaître est présentée à ce réseau qui modifie les poids de ses cellules en fonction des exemples (figure 8.24).

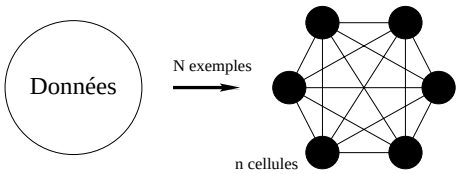


FIG. 8.24 - *En séquentiel*

Nous discernons deux grandes méthodes applicables à la parallélisation de réseaux de neurones :

- Les cellules peuvent être réparties parmi les processus. Cependant, les cellules sont généralement interconnectées et les communications sont nombreuses entre les cellules. Une telle parallélisation est difficile.

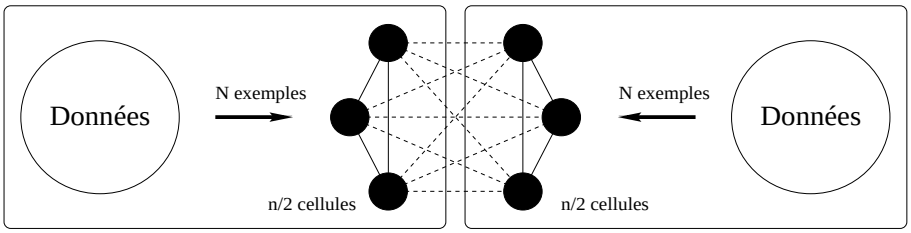


FIG. 8.25 - *Les cellules sont réparties entre les processus*

- On préfère modéliser une parallélisation basée sur un partage de la base de données d'exemples. Chaque processus simule le réseau de neurones complet, il contient contient une copie du réseau (ensemble de poids des cellules). La difficulté provient du fait du maintien de la cohérence des copies des réseaux.

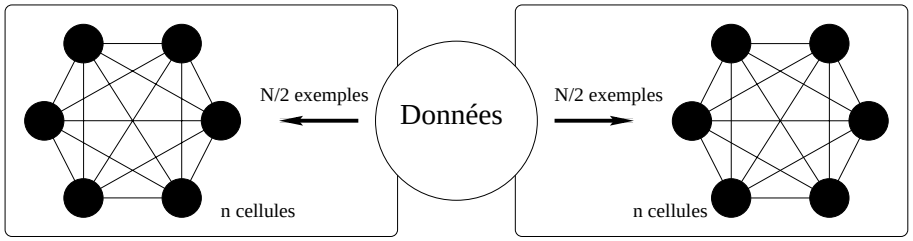


FIG. 8.26 - *On découpe les données, chaque processus contient le réseau complet*

Nous nous intéressons à la dernière méthode (figure 8.26) que nous paralléliserons à l'aide du système DOSMOS :

- Chaque processus contenant l'ensemble du réseau, l'application parallèle est proche du code séquentiel.
- L'ajout de deux types d'objets permet une parallélisation avec une MDVP. Il y a, tout d'abord, un objet complexe accessible seulement en lecture et qui contient la base d'exemples qui sera lue par chacun des processus (chaque processus n'accède qu'à un sous-ensemble des exemples). Le réseau (une matrice de poids) est lue et modifiée simultanément par tous les processus. Chaque processus doit donc disposer d'une copie locale du réseau.
- La cohérence des poids des cellules n'est pas nécessairement forte et peut être maintenue à l'aide de modèles de cohérene faibles. De temps en temps, un point d'arrêt (synchronisation) doit être effectué afin de moyenner les poids du réseau.

L'utilisation d'un réseau de neurones est généralement basée sur deux grandes phases. Tout d'abord, le réseau apprend à résoudre un problème à partir de la base d'exemples. Cet apprentissage occasionne la modification des poids des cellules du réseau. Puis une phase de généralisation peut être appliquée sur une autre base d'exemple où le réseau essaie de reconnaître les exemples de la base. Ces deux phases doivent être parallélisée :

- Même si l'apprentissage n'est réalisé qu'une fois, l'obtention de poids des cellules performants nécessite de nombreuses expérimentations. Une parallélisation peu coûteuse est donc nécessaire pour accélerer cet affinage des paramètres du réseau ;
- Seule la généralisation est utilisée par la suite. Sa parallélisation est facile (type S.P.M.D.) et doit être performante pour améliorer les temps de reconnaissance.

Ces travaux préléminaires sont en cours de développement et aboutiront prochainement à leur expérimentation sur différents types de réseaux afin de proposer une méthode de parallélisation facile fondée sur un système de Mémoire Distribuée Virtuellement Partagée.

8.7 Conclusion

Les expérimentations précédentes ont montré que la création de petites applications à l'aide de notre environnement de programmation basé sur DOSMOS est assez aisée. Le système permet d'atteindre de bonnes performances en fournissant un accès aux objets rapide. Mais atteindre de bonnes performances avec de plus grosses applications peut être difficile. Comme la MDVP masque toutes les optimisations et les communications, les performances peuvent fortement varier en fonction du niveau de connaissance de l'utilisateur.

Dans les figures 8.27 et 8.28, on peut appréhender la facilité de programmation d'une application DOSMOS. Ces codes permettent aussi de mieux comprendre les différences qui peuvent exister entre un utilisateur débutant et un expert en parallélisme.

```
for( i=istart ; i<=iend ; i++ )
    {
         dm_acquire( Matrice1[ligne,ligne]);
         pivot=dm_get_float(Matrice1[ligne,ligne]);
         dm_release( Matrice1[ligne,ligne]);
         dm_acquire( Matrice1[i,ligne]);
         coef= -1.00 * dm_get_float( Matrice1[i,ligne]) / pivot;
         dm_release( Matrice1[i,ligne]);
         for( j=ligne; j<Matrice1:col ; j++ )
         {
            double d1,d2;
            dm_acquire(Matrice1[i,j]);
            d1=dm_get_float( Matrice1[i,j]);
            dm_release(Matrice1[i,j]);
            dm_acquire(Matrice1[ligne,j]);
            d2=dm_get_float( Matrice1[ligne,j]);
            dm_release(Matrice1[ligne,j]);
            if( !EstNul( coef ) )
              buf=d1+ coef * d2;
            else
              buf=d1;
            dm_acquire(Matrice1[i,j]);
            dm_put_float( Matrice1[i,j],buf);
            dm_release(Matrice1[i,j]);
        }
    }
```

FIG. 8.27 - *Première version d'un code calculant la triangularisation de Gauss*

```
  pivot=dm_get_float(Matrice1[ligne,ligne]);
  for( k=ligne; k<Matrice1:col ; k++ )
          d2[k]=dm_get_float( Matrice1[ligne,k]);
  for( i=istart ; i<=iend ; i++ )
    {
      coef= -1.00 * dm_get_float( Matrice1[i,ligne]) / pivot;
      for( j=ligne; j<Matrice1:col ; j++ )
        {
          d1=dm_get_float( Matrice1[i,j]);
          if( !EstNul( coef ) )
            buf=d1+ coef * d2[j];
          else
            buf=d1;
          dm_put_float( Matrice1[i,j],buf);
        }
    }
```

FIG. 8.28 - *Code DOSMOS Gauss optimisé*

Comme DOSMOS propose différentes sémantiques de cohérence, les utilisateurs débutant préfèrent généralement garantir leurs accès aux objets partagés en choisissant une cohérence leur permettant de vérouiller les accès (code de la figure 8.27). Cela a aussi pour effet de réduire les performances de l'application puisque tous les accès sont synchronisés. Ce genre de code pourrait être fortement simplifié en utilisant des variables locales et en relâchant certains accès à la matrice partagée. En effet, dans ce genre d'application, les objets partagés préalablement découpés, peuvent être accédés de manière indépendante. Ainsi, de même qu'avec l'échange de messages l'utilisation

performante d'un système de MDVP passe une formation des utilisateurs aux concepts manipulés lors du développement de l'application parallèle.

CHAPITRE 9

Discussion

Nous avons présenté, dans ce document, à la fois un nouveau modèle de programmation et un nouveau système de MDVP fondé sur l'introduction de concepts originaux.

9.1 Un nouveau système de MDVP

Le système DOSMOS (et le modèle sous-jacent) s'articule autour d'un certain nombre de concepts et choix d'implémentation :

La gestion d'un ensemble d'objets partagés permet de s'affranchir des contraintes matérielles souvent prépondérantes dans les systèmes existants (à l'inverse des MDVP à base de pages). Le découpage des objets permet, on l'a vu dans les expérimentations, d'améliorer la disponibilité des objets et d'offrir une bonne capacité d'adaptation aux spécificités des applications. Ce découpage fait disparaître tout risque de faux partage qui pourrait aussi apparaître avec de gros objets partagés par plusieurs processus. La transparence d'accès aux objets (quel que soit le mode de découpage choisi) simplifie le code et permet à l'utilisateur d'affiner son application sans mettre en oeuvre d'importantes modifications.

Les sémantiques de cohérence variées fondées sur des protocoles d'invalidation optimisés permettent de prendre en compte la majorité des comportements d'accès aux objets auxquels l'utilisateur doit faire face. Les expériences précédentes ont montré qu'il est inutile de proposer à l'utilisateur une vaste gamme de primitives différentes pour accéder à ses données partagées. Trop de modèles de cohérence déroutent l'utilisateur. C'est la raison pour laquelle, le système DOSMOS ne propose que quelques modèles de cohérence bien différenciés, dont le comportement est facilement compréhensible par le programmeur.

Structuration hiérarchique

La structuration en groupes de processus permet de pallier les inconvénients des modèles de partage "*à plat*" dans lesquels tous les processeurs partagent tous les objets de l'application [1].

1. "Une mémoire partagée est une mémoire qui semble contenir un seul espace d'adressage et qui peut être accédée par n'importe quel processus du point de vue de l'utilisateur" [VW93].

Conçu à la fois pour des systèmes parallèles et distribués, DOSMOS propose une plate-forme de développement portable permettant de tester des applications sur une vaste gamme d'architectures. Les premières expérimentations ont montré l'efficacité de ce système et l'intérêt de la notion de groupes en terme de performances.

- Sur réseaux de stations de travail : comme les stations partagent physiquement le même média (Ethernet), il est important d'éviter des communications coûteuses entre les processeurs ce que permet de réaliser le partitionnement des processus en groupes.
- Sur machines parallèles : en limitant les diffusions globales, DOSMOS est bien adapté aux topologies (hypercube, grille...) et aux architectures (machines à base grappes de processeurs) qui tirent avantage du voisinage des processus.

La structuration hiérarchique en groupes de processus propose, en outre, une alternative intéressante aux traditionnelles listes de copies utilisée dans les systèmes de MDVP. Même les systèmes à base de pages peuvent tirer profit des groupes de processus. Ainsi, prenons l'exemple d'une machine futuriste à 1024 processeurs avec 128 MO de mémoire vive et des pages de données de 1Ko. Dans le pire cas, la taille mémoire nécessaire au stockage des listes de copies et égale à la taille maximum d'un *copy set* multiplié par le nombre de pages partagées. Cette machine futuriste devrait donc gérer une zone de mémoire correspondant à une table des pages de l'ordre de 128 millions d'entrée. Cette organisation n'est pas réaliste et risque de réduire à néant tous les efforts mis en oeuvre pour avoir une bonne extensibilité.

Portabilité

Le rêve de tout développeur de systèmes à MDVP est de pouvoir bénéficier de l'apport de couches matérielles dédiées pour la gestion de la cohérence des données, opération qui requiert des communications toujours coûteuses. Ainsi, de nombreux projets prometteurs sont en cours de développement (SCI [15992]...). Ces travaux semblent être la seule possibilité d'obtenir des performances comparables aux développements réalisés en échange de messages. En effet, la perte de performances due à la transparence des systèmes à MDVP est, alors, contrebalancée par une architecture dédiée où le ratio coûts de communications / coûts de calculs se rapproche d'un certain équilibre.

Mais toutes ces tentations pour l'obtention de performances alléchantes présentent un double inconvénient. Le développement de systèmes à mémoire Distribuée Virtuellement Partagée redevient très difficile car très proche de l'architecture sous-jacente. Cela risque d'empêcher l'intégration de nouvelles techniques de partage (cohérence, modèles de programmation...). La portabilité et la diffusion de ces systèmes sont clairement amoindries. Il y a donc un compromis difficile à trouver puisque d'un côté la généralisation des systèmes de MDVP passe par une offre de performances alléchante pour l'utilisateur; et d'un autre côté, les utilisateurs restent réticents à l'achat de nouveaux systèmes, préférant se contenter d'ajout de couches logicielles.

L'implémentation de DOSMOS au-dessus de PVM présente à la fois des avantages et des incon-

vénients. La couche PVM utilisée dégrade légèrement les performances des applications. La force de PVM est de fournir une plate-forme de développement robuste et portable sur de nombreuses configurations matérielles. Nous avons ainsi bénéficié de la portabilité de PVM pour installer DOSMOS sur différentes machines parallèles (CRAY T3D, Capitan...)

Adaptation

Nos discussions avec les utilisateurs potentiels de ce système nous ont amenés à la considération suivante. Les utilisateurs sont souvent plus intéressés par les services proposés par un modèle de programmation que par les performances brutes. De plus, dans le cas de parallélisation de codes existants, ce qui leur importe le plus, c'est la portabilité du système. Il ne veulent surtout pas avoir à réécrire les applications.

En proposant d'intégrer différents modèles de programmation, notre système s'adapte ainsi au comportement et aux habitudes de programmation de l'utilisateur et garantit une bonne portabilité.

Comparaisons avec les travaux précédents

En l'absence de bancs d'essai (*benchmarks*) standards, il est relativement difficile de comparer les différents systèmes de MDVP[2]. En effet, chaque MDVP dispose de primitives qui lui sont propres, de protocoles de cohérence adaptés, de méthodes dédiées... Le portage d'une application nécessite souvent la réécriture de la totalité du code. Il faudrait pour permettre un portage aisé que les systèmes de MDVP proposent des modèles de programmation les plus transparents possibles. Grâce à sa structure et à son environnement de programmation, DOSMOS va dans ce sens. On a vu, ainsi, que les codes DOSMOS étaient très proches des codes séquentiels.

Les études expérimentales que nous avons menées ont néanmoins permis d'évaluer la pertinence des notions et concepts que nous avons introduits. Ces mesures ont ainsi montré que la structure et les fonctionnalité du système permettent de garantir souplesse d'utilisation, efficacité et extensibilité.

9.2 Un nouvel environnement de programmation parallèle

Malgré l'effort de recherche considérable réalisé au cours des dix dernières années en ce qui concerne les systèmes de MDVP, on ne peut que constater que ce modèle de programmation n'est pas encore utilisé massivement par les programmeurs d'applications parallèles. Si on observe la diffusion de ces systèmes, on se rend compte que très peu de systèmes ont été largement diffusés auprès des utilisateurs si ce n'est LINDA et MUNIN/TreadMarks. La question que l'on peut alors se poser est "Mémoire virtuelle ou Utilisateur Virtuel?" En d'autres termes, les systèmes de MDVP sont-ils réellement utilisés?

2. Ce pourrait être l'objet d'une thèse que d'établir une batterie de tests représentatifs et de mener une étude comparée des différents systèmes de MDVP.

Nous pensons que le faible développement de ces systèmes provient de multiples facteurs. Nous en avons déjà cité quelques-uns dans nos motivations. L'un des principaux problèmes vient du fait que les systèmes de MDVP ne sont jamais présentés comme première approche du parallélisme aux utilisateurs débutants. Les systèmes de MDVP mettent souvent en avant des notions (sémantiques de cohérence...) qui n'ont aucun lien applicatif évident pour l'utilisateur qui désire paralléliser un algorithme.

De plus, le manque d'outils associés aux systèmes de MDVP se fait cruellement ressentir pour les utilisateurs. Très peu de travaux associent à un système de MDVP un environnement de programmation. Or nous avons vu tout l'intérêt de ce type d'outils.

Difficulté

Néanmoins, il ne faut pas le cacher à l'utilisateur, programmer de manière performante avec un système de MDVP nécessite aussi des efforts. Même si de nombreuses étapes de l'exécution sont cachées, l'utilisateur doit "penser parallèle". Nous l'avons vu dans nos expérimentations, s'il "pense séquentiel" (gros objet non-découpé, absence de groupes...), il obtiendra des performances décevantes ! L'utilisation d'un tel environnement de programmation est une aide à la "pensée parallèle" pour l'utilisateur qui peut prendre facilement en compte toutes les spécificités de ce modèle de programmation.

0 CHAPITRE 10 Conclusion et Perspectives

Nous avons essayé, dans cette thèse, de faire le point et d'apporter des idées novatrices sur un sujet à l'intersection de nombreux domaines de recherche : parallélisme, systèmes distribués, systèmes d'exploitation, environnements de programmation...

Fondée sur une analyse aussi large que possible de l'état de l'art en Mémoire Distribuée Virtuellement Partagée, la contribution de cette thèse se situe à un double niveau :

D'une part, nous avons proposé un nouveau modèle de mémoire distribuée virtuellement partagée. Ce modèle s'appuie sur une structuration hiérarchique de l'application en groupes de processus. Étendant et généralisant la notion de *copy set*, ce concept, à la fois, garantit une bonne extensibilité aux applications grâce à une minimisation des coûts de gestion et permet de s'adapter au plus près à la structure des applications et des architectures-cibles. Ce modèle améliore également la disponibilité des données et réduit la sévérité de certains goulots d'étranglements via le découpage et la gestion répartie des gros objets partagés et la mise en oeuvre de protocoles de cohérence relâchés. Afin de s'adapter à des contraintes d'efficacité et de maintenance, on autorise le mélange de plusieurs modèles de programmation. Cette approche a été mise en oeuvre dans le cadre du système DOSMOS, implémenté au-dessus de PVM. Dans un souci de portabilité et d'évolution, ce système s'appuie sur une approche modulaire découplant gestion de la mémoire partagée et exécution du code application. DOSMOS est associé à un langage de programmation qui permet une excellente transparence des mécanismes d'accès aux objets partagés.

En outre, ce système est complété par un environnement de programmation parallèle intégré. Cet environnement assiste l'utilisateur dans toutes les étapes de la programmation en proposant un ensemble d'outils graphiques puissants. Un analyseur-optimiseur de code traduit et optimise le code application. Une interface permet de spécifier des plates-formes virtuelles hétérogènes intégrant réseaux locaux et machines massivement parallèles. A l'aide d'autres outils, l'utilisateur définit le graphe de tâches de l'application et le plonge dans la machine virtuelle en fonction des objets partagés auxquels accèdent les processus. Un dernier outil graphique permet d'analyser le comportement des application et d'en évaluer les performances grâce à la mise en œuvre de mécanismes de traçage. Enfin, des procédures automatiques d'installation du système et d'exécution des applications complètent cet environnement.

Différentes expérimentations ont montré la pertinence des concepts à la base du système DOS-

MOS. Un premier retour d'informations de la part d'utilisateurs a confirmé l'intérêt majeur d'un environnement de programmation performant.

Une perspective d'amélioration à court terme du système DOSMOS est de s'affranchir, quand c'est possible, de PVM en profitant des spécificités du système-type. Ainsi, quand l'utilisateur programme une application DOSMOS sur un réseau de stations, le système DOSMOS peut tirer profit, pour la communication entre processus localisés sur une même station, des possibilités d'accès à des espaces mémoires partagés. DOSMOS pourrait également profiter de l'utilisation des primitives bas-niveau fournies par les machines parallèles. PVM, dans ce cadre, ne serait utilisé que pour le développement de systèmes hétérogènes. Ce type d'adaptabilité ne doit, bien sûr, pas remettre en cause la portabilité du système. C'est à ce dernier de mettre en oeuvre de telles optimisations en fonction de l'architecture-cible.

Au cours des expérimentations réalisées avec le système DOSMOS, nous nous sommes aperçus que ce type de système est parfaitement adapté à des applications à gros grain. Dans un proche avenir, nous comptons, dans ce cadre, associer une notion de persistence aux objets de DOSMOS. Cela nous permettra ainsi d'appliquer les concepts de DOSMOS dans le cadre d'applications coopératives qui se développent de plus en plus sur des architectures distribuées.

Enfin, nous poursuivons la coopération industrielle engagée avec la société Matra Cap Systèmes afin d'adapter le système DOSMOS aux spécificités de certaines applications industrielles (ex: gestion de bases de données vidéo).

Un des objectifs de DOSMOS est de fournir au programmeur d'applications parallèles, un modèle et un environnement de programmation simples à utiliser. D'utilisation intuitive, c'est un environnement de programmation facilement accessible qui permet de débuter dans la programmation d'applications parallèles sur réseaux de stations ou sur machines parallèles.

L'environnement de programmation associé à DOSMOS a été conçu dans un but purement applicatif. Le fait d'avoir proposé cet environnement comme outil de développement à des utilisateurs (étudiants) nous a été très bénéfique. Sa conception a été entièrement orientée dans une perspective d'assistance à l'utilisateur. Cet environnement de programmation peut être encore amélioré. Nous étudions actuellement la possibilité de lui adjoindre des outils de déboggage qui aideront l'utilisateur à détecter les erreurs et inter-blocages et à optimiser son code lors de l'exécution.

Annexe

Article "**Execution Analysis of DSM Applications: A Distributed and Scalable Approach**" avec L. Brunie et O. Reymann présenté à SPDT'96 : SIGMETRICS Symposium on Parallel and Distributed Tools, ACM Press editor FCRC, Mai 1996.

Execution Analysis of DSM Applications: A Distributed and Scalable Approach

Lionel Brunie, Laurent Lefèvre and Olivier Reymann
Laboratoire de l'Informatique du Parallélisme
Ecole Normale Supérieure de Lyon
69364 LYON Cedex 07 , France
(lbrunie, llefevre, oreymann)@lip.ens-lyon.fr

Abstract

In the last five years, Distributed Shared Memory (DSM) systems have received increasing attention. Indeed, by releasing the programmer from the management of inter-process communications, they offer a very intuitive and easy-to-use programming paradigm. In compensation, such systems often appear, from the programmer point of view, as a "black box" since no information about the actual communications is available. Consequently, in the absence of visualization and monitoring tools, optimizing, debugging or evaluating the performance of DSM applications is very difficult. In that framework, this paper proposes an original monitoring model based on two new concepts: *meta-objects* and *event manager processes*. This model constitutes the basis of an actual monitoring system, called DOSMOS-Trace, that has been designed and implemented to monitor applications developed on top of the DOSMOS DSM system[1]. This monitoring environment is analysed in terms of functionalities, protocols and user interface. Experiments show the efficiency and the robustness of the underlying model as well as the pertinence, for the programmer, of such a monitoring tool.

Keywords: distributed shared memory, monitoring, performance evaluation, program visualization.

1 Introduction

In the last five years, Distributed Shared Memory (DSM) systems have received increasing attention. Indeed, by releasing the programmer from the management of inter-process communications, they offer a very intuitive and easy-to-use programming paradigm. In compensation, such systems often appear, from the programmer point of view, as a "black box" since no information about the actual communications is available. Consequently, in the absence of visualization and monitoring tools, optimizing, debugging or evaluating the performance of DSM applications is very difficult.

Until now, most of monitoring tools have been designed for message passing applications. However, in spite of evident points of convergence, monitoring DSM applications differs from monitoring message-passing applications because pertinent information to be traced is clearly different. In that framework, this paper proposes an original monitoring model based on two new concepts: *meta-objects* (*i.e.* specific distributed data-structures designed for the storage of monitoring data) and *event manager processes* (*i.e.* specialized distributed processes in charge of the on-the-fly collection and management of execution traces).

This paper is organized as follows. In section 2, we recall some basic points about DSM systems and analyse the functionalities that should be provided by DSM-oriented monitoring tools. Previous works on parallel monitoring are studied in section 3. Then, the DOSMOS system is briefly presented in section 4. The basics of the monitoring model we propose are described in section 5 while protocols are analysed in section 6. Section 7 presents the DOSMOS-Trace monitoring environment which implements the concepts introduced in this paper. An analysis of the intrusion generated by the monitoring is proposed in section 8. Section 9 discusses the pertinence and the effectiveness of the model presented in this paper. Finally, section 10 concludes this paper and analyses the main perspectives of this work.

2 Monitoring DSM applications

2.1 What is a DSM System?

Basically, Distributed Shared Memory systems (DSM) allow, above a distributed memory architecture, the manipulation of shared data in a transparent way. In other words, in such systems, a programmer can make the processes of his application share data without explicitly programming the inter-process communications which are actually handled by the system.

Two basic approaches have been studied:

virtual sharing of memory pages: the environments of this type [Li88, LP92, FP89, CBZ91, HS92] merge various memory pages distributed in the system into a single address space.

virtual sharing of variables or objects: such systems [RAK89, TKB92, CG89, BL94, BL96], more programming oriented, allow the user to define *shared variables* (or *shared objects* for object-oriented systems) which

[1] DOSMOS is the acronym of Distributed Objects Shared MemOry System. The DOSMOS system has been developed on top of PVM in our laboratory.

will be accessible in a transparent way from any node in the network.

The purpose of this paper is not to discuss the respective advantages and drawbacks of these two kinds of systems. Indeed, from the programmer point of view (and thus for the monitoring point of view), all these systems implement the same basic functionalities, *i.e.* transparent manipulation of shared data.

2.2 Monitoring DSM applications

The goal of any monitoring tool is to collect information about the execution of the application (called execution traces) and to display it in a pertinent way in order to allow the programmer to understand the behaviour of his application. In the framework of DSM applications, execution traces can be grouped in two classes:

Information about the DSM system administration: creation, destruction, migration of processes and, to speak more generally, of system "entities" (*e.g.* group of processes for the DOSMOS system (cf. section 4)).

Information about the shared data: duplication of shared data (evolution of the number of copies distributed in the system), migration, number and types of accesses (*e.g.* read-only, exclusive write, concurrent write, ...), a list, for each shared datum, of the processes that frequently modify it, *etc.*

Analysing such traces will allow (section 7) the programmer to detect and correct most critical situations, *e.g.*:

bottlenecks: a bottleneck occurs when a shared datum is too frequently accessed in an exclusive way. A possible solution consists in splitting the shared datum into several sub-variables in order to distribute the accesses over several objects;

ping-pong effects: this occurs when a variable is, for a long while, concurrently accessed by two, or more, processes. Possible solution: splitting of the variable;

no-sharing: when a variable is declared as shared but only one process actually accesses it. Solution: declaration of the variable as a local variable in order to circumvent the DSM system layer.

specific features: for instance, in the DOSMOS system, a bad group structure.

The purpose of this paper is twofold: first, proposing and studying a software architecture able to efficiently implement such monitoring functionalities (section 5); second, illustrating the relevance of this model by analysing the facilities provided by the DOSMOS-Trace environment which has been designed and implemented according to these principles (section 7).

3 Previous Works

Basically, monitoring an application requires dealing with three main problems: first, the collection of execution traces; second, the management and storage of the traces; finally, the analysis and visualization of the traces. Until now, most monitoring tools have been designed to trace message-passing applications. In that framework, the trace collection is usually performed by instrumenting the application code in order to monitor the most important events occurring during the execution. The instrumentation can be placed at various levels:

- operating system level (*e.g.* Tapestry [MR90]): tracing of events like communications, creation of processes, memory accesses, system calls;
- run-time environment level: (*e.g.* IPS [MY87]): entries into and exits from parallel sessions, use of barriers, procedure calls;
- application level (*e.g.* IVE [FLK+91] or PVVT [Str90]): working at this level allows focusing the monitoring on the most interesting parts of the source code. The systems of this type are all based on the same principle: inserting additional code into the user program in order, as previously, to trace the most important events.

Once the traces are collected, it is necessary to manage them, to organize them in the memory. On the other hand, post-mortem analysis requires the storage of traces on disk. Though a few monitoring tools (*e.g.* SIEVE [SG92] or The Belvedere system [HC87]) implement database techniques, most monitoring systems use "ordinary" files. Recently, in order to increase the scalability and the efficiency of monitoring tools, some works (*e.g.* PIMSY [TV94a, TV94b]) have proposed the use of distributed trace files. The DOSMOS-Trace system lies within this approach.

Last point: the visualization of traces. Basically, all systems allow the visualization of inter-processes communications [KS93]. Furthermore, each system proposes its own specific features: hierarchical visualization (*i.e.* grouping of processes), performance analysis, traffic analysis, processors activity,...

DSM-oriented monitoring tools are not over-abundant. Most tools have been designed for virtual shared memory systems (*i.e.* page-based systems (section 2.1)). In that framework, attention has mainly been focused on *system* events: accesses to pages[Ede93], false sharing, cache misses. However, for the basic programmer, such information is very difficult to use (*e.g.*, what measure is to be taken if a page is over-used ?). In other words, such monitoring systems provide pertinent data for DSM system designers, but not for end-users.

Nevertheless, some works have proposed focusing on higher-level features. Thus, Brorsson and Strenstrom [BS92], propose an analysis tool based on the study of four parameters: the spatial granularity, the degree of sharing, the access mode and the temporal granularity. These parameters allow the comparison of various coherence strategies in order to choose the most pertinent one (for the target application...). SHMAP [DBKF90] provides a visualization of memory access patterns, cache strategies and processor assignment. In a different context, designed to trace applications developed on top of shared-memory multiprocessors, Robinson, David and Enbody [RCE92] propose the observation of causal dependencies between events. In the same framework, [LMCF90] implements a specialized library in which every access to a shared data is assigned a logical sequence number used to infer a partial ordering of the execution. MTOOL [GH91]) tries to isolate memory bottlenecks by comparing the actual execution with an ideal execution performed on a perfect shared memory machine.

However, most of these approaches suffer from important drawbacks. First, most of them provide information hardly

usable by the end-user. Second, tools designed to monitor shared-memory multiprocessors use specific features provided by these machines and so are not portable. Thus, like most message-passing monitoring tools, MPTrace [EKKL90] instruments the application code. However, this is not realistic for software DSM systems. Indeed, no internal information is available at the application level, *e.g.* it is impossible, from the application, to know where a shared variable is located, which process manages it, etc. So, working at the application level would require a complete modification of the interface between the DSM system and the applications, which would deeply affect the actual behaviour of the applications. Finally, most existing systems (and specifically DSM-oriented systems) are intrusive (*e.g.* systems instrumenting the DSM management routines) and not scalable (*e.g.* systems using a centralized trace file).

4 The DOSMOS System

The DOSMOS-Trace monitoring environment (section 5) has been designed to monitor applications programmed on top of the DOSMOS DSM system developed in our laboratory. The purpose of this paper is not to study this DSM system (see *e.g.* [BL94, BL96]). However, to better understand the functionalities provided by the DOSMOS-Trace environment, it may be necessary to say a few words about the DOSMOS system.

DOSMOS is a variable-oriented DSM system (section 2.1) developed on top of PVM. The user can declare either basic type variables (*e.g.* integers, floats,...) or arrays that will be split into several "system objects". Various splittings are provided: by row, by column and by block. Basically, a DOSMOS application is composed of two kinds of processes:

Application Processes (AP) which execute the user's code;

Memory Processes (MP) which manage the shared variables and handle the access requests issued by Application Processes.

To avoid expensive synchronizations and useless communications which break down the efficiency of the applications, DOSMOS allows grouping together the processes which actually share a common set of variables. These groups can be hierarchically structured into groups and subgroups. Furthermore, to maintain the coherence of the shared data, the DOSMOS system implements a weak consistency protocol called *release consistency*. This protocol is based on two primitives: *acquire* which allows obtaining an exclusive access to a variable and *release* which unlocks this variable (for a complete discussion on consistency protocols, see [RM93]).

Though Application Processes and variables can be structured in groups, any shared variable is accessible from any Application Process. Link Processes (LP) are specialized MP devoted to inter-group operations on shared variables. Thus, thanks to LPs, an Application Process can access variables managed by other groups than its own. However such inter-groups accesses are, of course, more expensive than intra-group accesses. Figure 1 shows an example of software configuration including 6 APs, 3 MPs and 2 LPs.

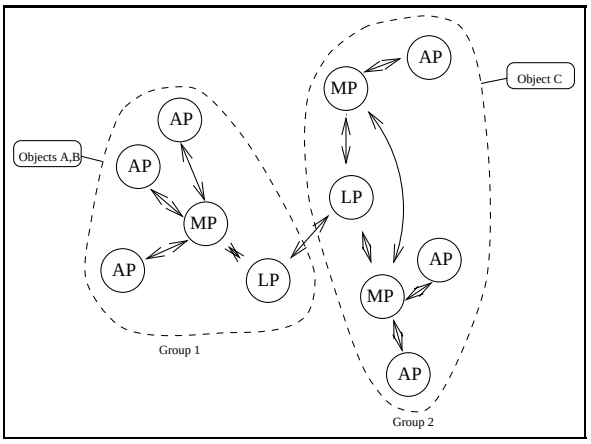


Figure 1: DOSMOS system: an example of software configuration with two groups and three objects A, B and C

5 A Model for DSM Application Monitoring

5.1 Trace Detection and Collection: Event Manager Process

As previously discussed, instrumenting the user's code is not realistic. Consequently two approaches are possible: first, modifying the DSM code, *i.e.*, in the DOSMOS context, modifying the code of the memory processes (MP). This approach presents important drawbacks: as in any DSM system, the whole efficiency of DOSMOS relies on the memory processes. Thus, loading MPs with the monitoring would deeply affect the behaviour of the system. So, in order to minimize the monitoring intrusion, we propose to introduce a new kind of system processes called *Event Manager Processes* (EMP). An EMP is linked to one or more memory processes. Once an MP detects an event, it sends a message to the EMP it depends on. EMPs are in charge of the whole management of the execution traces. Protocols defining the coordination procedures between memory processes and EMPs are described in section 6. This approach presents important advantages. First, it minimizes the work requested from MPs, and consequently, the intrusion due to monitoring[2]. Furthermore, in the case of post-mortem utilization, traces have to be stored on disk. However, as traces are managed by EMPs, which are distributed in the whole network, the storing on disk is not performed by a single process but by all the EMPs, which is clearly more scalable and efficient. The scalability of the system can even be increased if several distributed trace files, located on several disks, are used. In fact, the best (but the most expensive...) solution is to attach a local disk to each processor on which an EMP runs. Indeed, by increasing the I/O bandwidth, such an architecture allows reducing the bottleneck constituted by the transfer of traces to disk. Figure 2 shows an example of the monitoring environment. This configuration uses three processors, two logical groups and two shared variables A and B.

Finally, experimentally it appears that tracing realistic applications generates a huge amount of traces which affects the intrusion. That is why, to reduce the volume of traces, the DOSMOS-Trace system allows the user to specify which information he is interested in.

[2] This intrusion will be even reduced if EMPs are located on dedicated processors (in order not to "steal" CPU time from memory processes).

5.2 Trace Management: the Meta-Object Concept

In contrast to post-mortem analysis, on-line monitoring tools require keeping execution traces in memory. To manage these traces, we propose to introduce new data structures, called *meta-objects*. A meta-object is a tuple (record) with as many fields as different monitoring informations.

However, in the DOSMOS system, for efficiency purposes, a variable can be duplicated, *i.e.* several read-only copies of a variable can be distributed (within the group of processors sharing the variable). Therefore, it is necessary to distinguish between two types of meta-objects:

a *primary* meta-object is attached to each shared variable. It contains information about the variable such as the number and the type (read, write, acquire) of the accesses performed on the variable. It also maintains the list of the processes that recently accessed the variable, the origin (local, intra-group, inter-group) and the characteristics of the accesses they requested (*i.e.* type (read/write/acquire)). This information is very useful to analyse the behaviour of the application and to propose optimizations. The primary meta-object of a variable is managed by the EMP monitoring the MP owner of the variable.

***secondary* meta-objects** are attached to each copy of a shared variable. Because a copy can only be accessed in a read-only fashion, a secondary meta-object does not have to store as much information as a primary meta-object does. In practice, secondary meta-objects record the identification of the MP that owns the copy, the identification of the EMP that manages the primary meta-object and the number of read operations performed on this copy. A secondary meta-object is managed by the EMP attached to the MP owner of the copy.

Secondary meta-objects allow the user to know the actual distribution of the read accesses among the processes. This information is important because it deals with the group structure of the application and with the efficiency of the implemented consistency protocol. Indeed, to be efficient, DSM applications should perform as many local accesses as possible (because remote accesses are more expensive).

Remark: write accesses require bringing invalidation protocols into play. These protocols are triggered by the Memory Process owner of the variable. This Memory Process is connected with the EMP which manages the primary meta-object. Consequently, all the write accesses are traced in this primary meta-object.

5.3 Analysis and Visualization of Execution Traces

Whether they work in an on-line or post-mortem fashion, analysis tools must interact with EMPs which are the only processes able to access monitoring data. This argues for implementing a client-server architecture in which EMPs act as servers and tools as clients.

The DOSMOS-Trace system implements the following approach (figure 3): a Visualization Process (VP) is started at the beginning of the execution (on-line monitoring) or after the execution (post-mortem analysis). The user submits queries to this process which passes them to all the EMPs concerned. These latter return the requested information to the VP which is in charge of the fusion of these data. Finally, the VP displays the results.

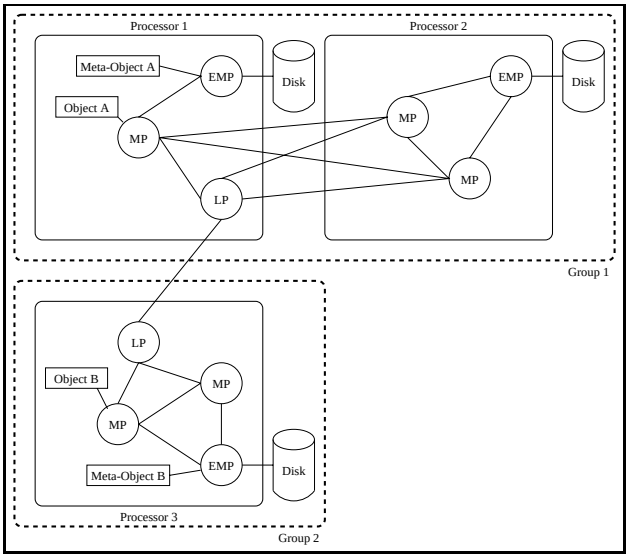


Figure 2: DOSMOS-Trace: example of monitoring environment

6 Implementation and System Architecture

6.1 Meta-Objects

As described in section 5.2, meta-objects are designed to store and manage the traced information in memory. However, as several copies of the same variable can be distributed in the network, several kinds of meta-objects must be distinguished.

Thus, in the DOSMOS-Trace system, a *primary meta-object* is associated with the main copy of a variable[3]. This meta-object contains general information such as:

- Variable identification (name, system identication)
- Group: this field contains the identification of the group the shared variable belongs to. It is used to analyse and visualize the group structure.
- Number of copies of the variable distributed in the system
- Memory process owner of the variable
- Number of read operations performed on this main copy
- Total number of read accesses performed on all the copies (see below)
- Number of write accesses
- Number of acquire and release operations
- List of last acquire operations
- List of last write operations
- List of delayed acquire operations

[3] In the DOSMOS system, a shared variable is managed by one Memory Process (Distributed Static Owner protocol ([LH89])). This Memory Process controls the duplication of the shared variable, handles the copies invalidations and manages the write accesses.

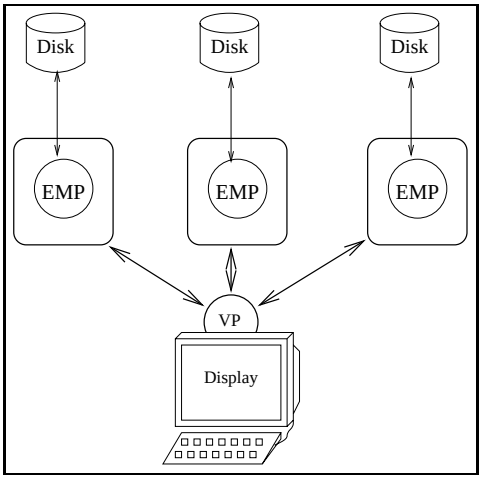


Figure 3: DOSMOS-Trace: the Visualization Process (VP) communicates with all the Event Manager Processes distributed in the network.

These last three lists store triplets containing the identifier of the Application Process from which the operation was issued, the identifier of the Memory Process that received this query and the group the Application Process belongs to.

Variable copies are monitored using *secondary meta-objects*. A secondary meta-object is attached to each copy of a variable. It contains the following information:

- Variable identification
- Memory process owner of this copy[4]
- EMP which manages the primary meta-object
- Number of read operations performed on the copy

Secondary meta-objects permit a very accurate view on the execution. More precisely, secondary meta-objects allow knowing the actual distribution of the read accesses among the processes. This information is important because it concerns the group structure of the application. Indeed, the efficiency of DSM applications is largely determined by the ratio of the number of local accesses to the number of remote accesses[5]. So, analyzing the read access distribution is extremely important to understanding the behavior of DSM applications well.

Moreover, using secondary meta-objects allows the EMP managing the primary meta-object to be discharged from the management of the traces generated by the copies. As a consequence, it increases the scalability of the monitoring system (both from a CPU point of view and an I/O point of view (if, of course, EMPs use several disks)).

6.2 System Architecture

Figure 2 shows an example of process configuration during a monitored execution. This architecture follows a few rules:

- One Event Manager Process at most can be run on one processor;
- Each EMP must be connected to at least one Memory Process;
- A Memory Process sends its trace information only to its dedicated Event Manager Process;
- A Memory Process must deal with at least (and eventually more than) one Application Process;
- An Application Process communicates with only one Memory Process.

Event Manager Processes can switch between two modes. During execution, EMPs receive messages from the memory processes concerning the various operations performed on the shared objects. They store these traces in memory (meta-objects) and/or on disk (trace files). At the end of the execution, EMPs remain alive in order to answer to the queries issued by the user.

6.3 Protocols

This section describes the protocols implemented for passing traces information from MPs to EMPs.

During a variable access, two kinds of memory processes must be distinguished:

Primary Memory Process (PMP): the Memory Process that owns the requested variable;

Secondary Memory Process (SMP): any Memory Process that received an access request from one of its Application Processes but does not own the requested variable. It possibly has one copy of that variable.

The management of shared memory is based on four standard operations: write access, read access, acquire and release.

6.3.1 Write Operation Protocol

The protocol used for a write access is the simplest one. Two cases are possible:

Local write access: (figure 4.a) The PMP directly receives the AP write request (1) ; it modifies the variable and informs its EMP (2) in order to store the operation on disk and update the meta-object.

Remote write access: (figure 4.b) The SMP receives the AP write request (1), informs its EMP (2) to store the operation on disk and forwards the request to the PMP of the variable (3) which performs the write access. Then the PMP sends a message to its EMP (4) to update the variable's primary meta-object.

6.3.2 Read Operation Protocol

Local read access: (figure 5.a) The MP receiving the AP read request (1). It owns either the variable itself or a valid copy of this variable. It then returns the requested value to the AP (2) and informs its EMP (3) to store the operation on disk and update the meta-object.

Remote read access: (figure 5.b) In this case, the MP receiving the AP read request (1) does not own a valid copy of the variable. This request is then forwarded to the variable PMP (2) which returns the value of the variable (3) and sends a message to its EMP (4) to

[4] This information is mandatory because an EMP can monitor several Memory Processes (see section 6.2).

[5] Remote accesses are much more expensive.

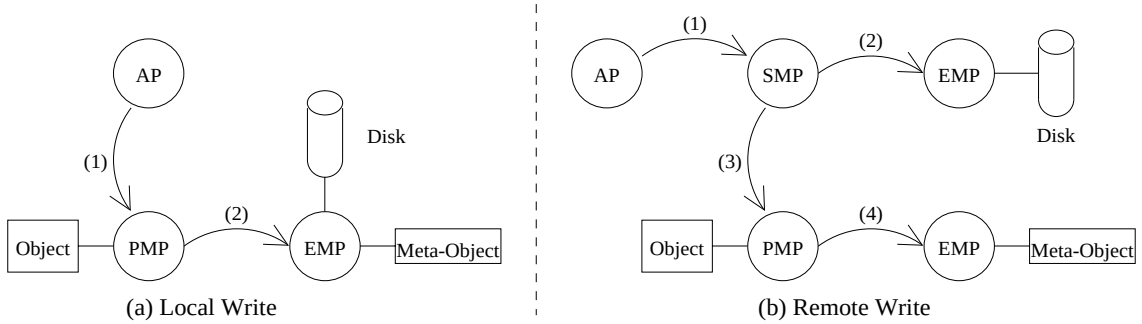


Figure 4: Protocol implemented to collect the trace information about a write operation.

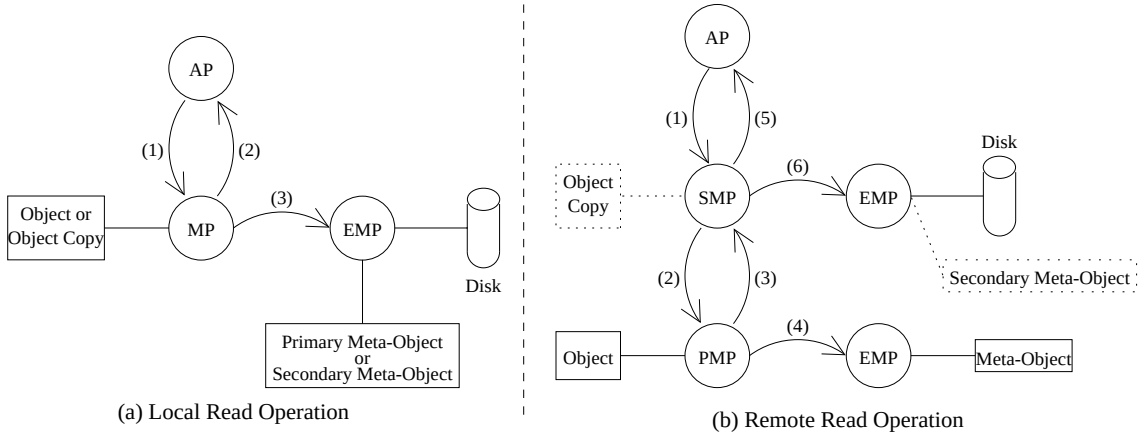


Figure 5: Protocol implemented to collect the trace information about a read operation.

update the meta-object attached to it. The secondary MP forwards the value to the calling AP (5), creates a variable copy and notifies its EMP to store the operation on disk and to generate a secondary meta-object.

6.3.3 Acquire Operation Protocol

To obtain an exclusive write access right on a variable (figure 6), an AP must generate an Acquire request message and sends it to its MP (1). Two cases must be distinguished:

This MP is the PMP of the variable: (figure 6.a) If the variable is free (*i.e.* not acquired by another AP), it gives the exclusive write access right to the AP (2) and sends information to its EMP (3) in order to store the operation on disk and update the meta-object. If the variable is already acquired by another AP, the MP informs the EMP that a new AP is waiting for the variable (3). When it is released, the PMP gives the exclusive write access right to the AP (2) and reports it to its EMP (3).

This MP is not the PMP of the variable: (figure 6.b) The SMP forwards the request to the PMP of the variable (2). This latter verifies if the variable is free. In this case, it returns the exclusive write access write to the SMP (3) which forwards it to the AP (5). The EMP attached to the PMP updates the primary meta-object (4) while the EMP attached to the SMP stores the operation on disk (6). If the variable was already acquired, the PMP informs its EMP that a new AP is waiting for the variable (4). When it is released, the same action sequence is performed as in the case where the variable was immediately available.

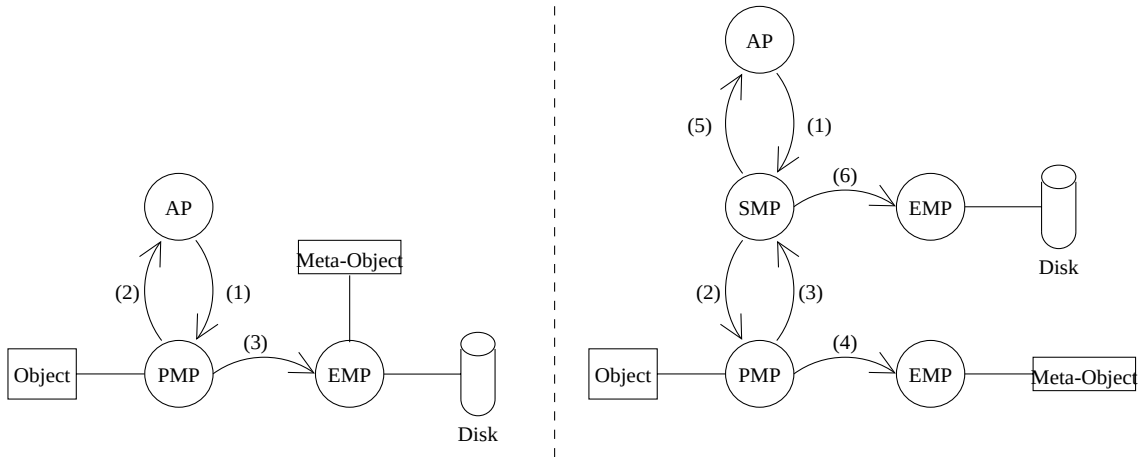
6.3.4 Release Operation Protocol

The management of a Release operation requires a lot of communications. Indeed, we must guarantee the consistency of all the object copies but also update the primary meta-object by sending to it all the data contained in the secondary meta-objects. This generates additional communications between EMPs.

Figure 7 shows a diagram of the protocol used by a Release operation in the most general case, *i.e.* when the release request is sent by an AP to a SMP (1). This latter forwards the request to the variable's PMP (2) which performs either an invalidation or an update of all the copies distributed in the system (3). Each SMP that has a copy informs its EMP (4) that it must send the data stored into the secondary meta-object to the EMP attached to the PMP in order to update the primary meta-object (5). When the PMP receives all acknowledgement messages issued by the SMPs (6), it updates the primary meta-object (7) and requests the SMP linked to the calling AP to inform this AP (8 and 9) and the EMP attached to this SMP (10) that the release operation is finished.

7 Interface and Experiments

Designed to trace applications developed on top of DOSMOS, the DOSMOS-Trace system actually implements all



(a) Local Acquire Operation (b) Remote Acquire Operation

Figure 6: Protocol implemented to collect the trace information about an Acquire operation.

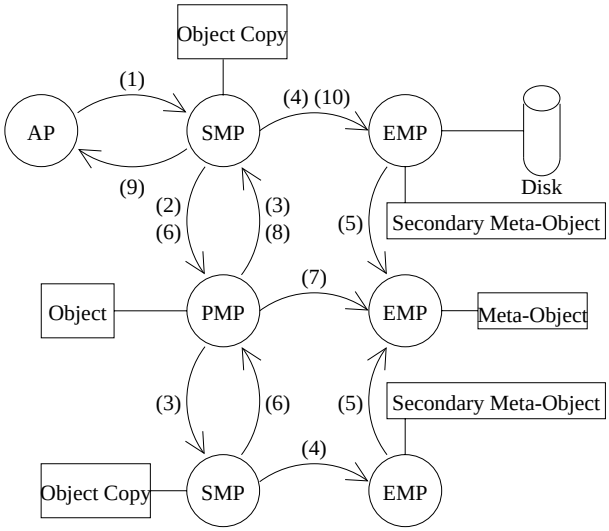


Figure 7: Protocol implemented to collect the trace information about a Release operation.

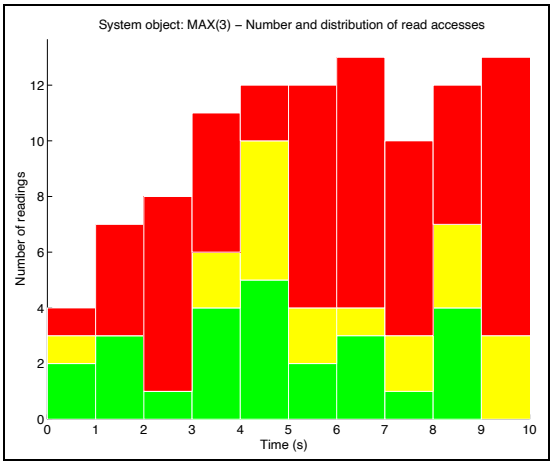


Figure 8: Number and origin of the read accesses performed on an object vs execution time (in black: inter-group accesses)

the concepts developed in the above sections: EMPs, meta-objects, distributed traces files, visualization process. The aim of this section is to illustrate its functionalities by presenting two examples of information provided by this system[6].

7.1 Accesses to shared Variables

Figures 8 and 9 display the histogram of the read accesses performed on a variable during the execution of an application. Various colors[7] are used in order to differentiate the origin of the accesses: local accesses are represented in green, intra-group accesses in yellow and inter-group accesses in red.

Such diagrams allow the user to detect a bad group structure. Thus, in figure 8, the predominance of inter-group accesses shows clearly that the group structure is not pertinent.On the contrary, in figure 9, one can verify that no inter-group accesses are performed.

In the same way, it is possible to visualize write and acquire accesses.

7.2 Histories

This functionality provides an analysis of the "history" of any shared variable (figure 10) or any application process (figure 11). In other words, it allows the visualization of all the accesses performed on a variable or, reciprocally, all the accesses performed by an application process.

On these figures, a "*" represents a write operation (under the dotted line), an "x" symbolizes a read access (above the dotted line) and a green "+" represents an optimized read access (*i.e.* a read access performed on a local copy of the variable). Black boxes are used to represent the amount of time that a process was waiting before it can perform either an acquire or a release.

[6] Figures are displayed using Matlab.

[7] Grey level correspondence: green=dark grey, yellow=light grey, red=black

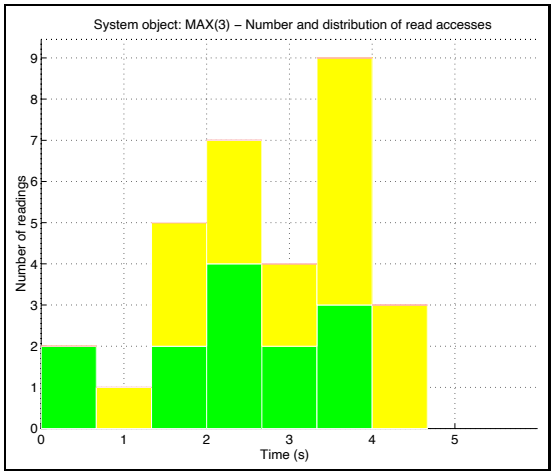


Figure 9: Number and origin of the read accesses performed on an object vs execution time (note this execution does not include inter-group accesses)

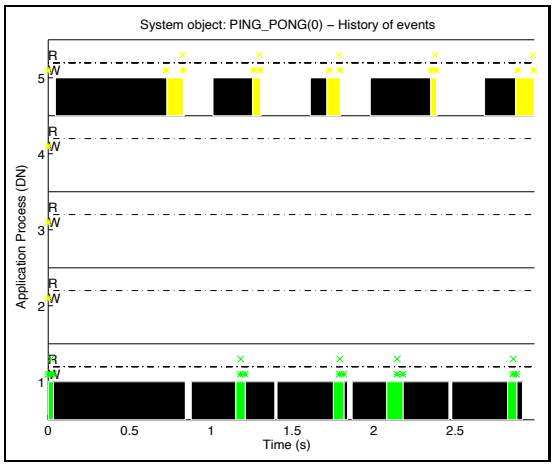


Figure 10: Object activity vs execution time

Such diagrams are extremely useful for the user in analysing problematical situations. Indeed they allow the very easily isolating ping-pong effects (*e.g.* figure 10), over-accessed variables, bottlenecks, not actually shared variables, etc.

8 Estimation of the Intrusion

This section presents the methodology we have followed to estimate the overhead time introduced by the monitoring.

The experiments were made on a network of SUN Sparc workstations. They are based on a sample application which consists of a sequence of exclusive write and read accesses applied to a variable without any computation. In terms of overhead time, this application is especially unfavorable. Indeed:

- This application does not perform any computation. The ratio overhead/execution time is consequently the worst possible;

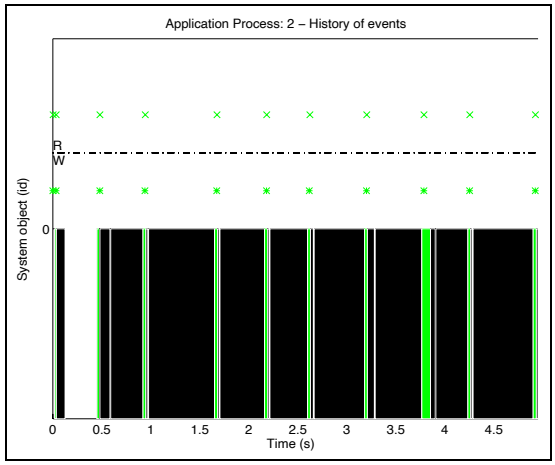


Figure 11: Process activity vs execution time

- All requests are made on the same variable. So one cannot take advantage of the variable's owner distribution;
- Only acquire and release operations are performed. However these operations are, as we have seen, the most exepensive ones.

In other words, results obtained using this sample application can be considered as an upper bound.

Four system configurations were used:

1. 4 workstations each one holding one Application Process and one Memory Process. This is the reference configuration.
2. 4 workstations each one holding one Application Process, one Memory Process and one Event Manager Process.
3. 4 workstations each one holding one Application Process and one Memory Process. Furthermore, one of them also runs one Event Manager Process which collects the events from all the MPs.
4. 4 workstations each one holding one Application Process, one Memory Process plus another workstation holding only an Event Manager Process which collects the events from all the MPs.

Table 1 shows the execution time obtained on these various system configurations. Obviously, the three monitoring configurations do not provide the same results. It is clear that an architecture containing dedicated processors only executing an EMP is the best solution. In this case, an overhead of 29% was obtained.

Consider now a "real" application, *i.e.* an application which not only makes accesses to shared data but also performs some computations. Let R be the ratio between the computation time and the access data time. The sample application represents the case where R equals zero. Its execution time, without monitoring, is equal to 21.90 seconds. For a given value of R, the execution time without monitoring can therefore be estimated as $21.90 \times (1 + R)$. In case of monitoring, it can be estimated as the time for the monitored execution without computation plus $R \times 21.90$.

Configuration	1	2		3		4	
Execution time	21.90	40.00	(+83%)	48.20	(+120%)	28.30	(+29%)

Table 1: Execution time (in seconds) for several configurations

Ratio \ Configuration	1	2		3		4	
R=1	43.80	61.90	(41%)	70.10	(60%)	50.20	(15%)
R=2	65.70	83.80	(28%)	92.00	(40%)	72.10	(10%)
R=3	87.60	105.70	(21%)	113.90	(30%)	94.00	(7%)

Table 2: Calculated execution time (in seconds) for different (computation/shared data access) ratios

Table 2 shows the estimated execution times (and the corresponding overhead (in percentage)) for 3 values of R. Thus, it appears that as soon as the computation time is higher than the access data time (which seems reasonable), the intrusion falls below 15%.

One question remains open: how many MPs must be managed by one EMP in order to keep the intrusion below a predefined limit? We are currently making tests in order to collect more experimental data.

9 Discussion

In comparison with previous approaches, the model presented in this paper presents several important advantages:

weak intrusion, due mainly to the introduction of dedicated distributed processes (EMPs).

scalability, due to the distributed architecture on which relies the model. Thus, EMPs are distributed as well as trace files;

flexibility: meta-objects are very flexible data structures. Adding a functionality to the monitoring environment only requires adding fields to the meta-object structure and specifying the protocol between EMPs and memory processes;

user-orientation: as illustrated in section 7, by working at the variable level, the DOSMOS-Trace system allows the user to clearly understand the behaviour of his application, especially to detect the most important problems: bottlenecks, ping-pong effects, bad group structure, activity imbalance,...

independence towards the shared data type: though designed for variable-oriented DSM systems, this model allows to deal with page-oriented systems. Thus, tracing the accesses to shared pages can be very simply handled by associating one meta-object to each page.

10 Conclusion and Future Works

This paper has described a novel model for the monitoring of DSM applications. This model relies on two original concepts: Event Manager Processes and meta-objects. In comparison with previous systems, this approach, based on a distributed architecture, has shown it was weakly intrusive and scalable. Implementing these concepts, the DOSMOS-Trace monitoring system has proved their efficiency and robustness.

Based on this model, further developments are mainly focused on the definition and implementation of an on-line automatic optimization tool (data migration, load balancing), *i.e.* on the automatic detection and correction, at runtime, of typical problematical situations (*e.g.* bottlenecks, ping-pong effects, bad group structure).

Acknowledgment

We would like to thank Mr Robert Halstead for his proofreading of this paper. His pertinent advices were a great help for us for improving the quality of this paper.

References

[BL94] Lionel Brunie and Laurent Lefèvre. DOSMOS : A distributed shared memory based on PVM. In *First european PVM users group meeting*, Universita di Roma, October 1994.

[BL96] Lionel Brunie and Laurent Lefèvre. New propositions to improve the efficiency and scalability of DSM systems. June 1996. to be published in the proceedings of the IEEE ICA3PP'96 conference (Singapore).

[BS92] Mats Brorsson and Per Strenstrom. Visualizing sharing behaviour in relation shared memory management. In *International Conference on Parallel and distributed systems*, Hinschu Taiwan ROC, December 1992.

[CBZ91] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of MUNIN. *ACM - Operating Systems Review*, 25(5):152–164, 1991.

[CG89] Nicholas Carriero and David Gerlenter. LINDA in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[DBKF90] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg. A tool to aid in the design, implementation and understanding of matrix algorithms for parallel processors. *Parallel Distributed Computing*, 9(2):185–202, June 1990.

[Ede93] Daniel R. Edelson. Fault interpretation: Fine-grain monitoring of page accesses. In *Winter USENIX*, pages 395–403, San Diego, CA, January 1993.

[EKKL90] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. *Performance evaluation review*, 18(1):37–47, May 1990.

[FLK+91] M. Friedell, M. LaPolla, S. Kochhar, S. Sistare, and J. Juda. Visualizing the behavior of massively parallel programs. In *Supercomputing*, pages 472–480, Albuquerque, November 1991.

[FP89] Brett D. Fleisch and Gerald J. Popek. MIRAGE: a coherent distributed shared memory design. In ACM PRESS, editor, *Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, volume 23, pages 211–223, The Wigwam Litchfield Park, Arizona, December 1989.

[GH91] Aaron Goldberg and John Hennessy. MTOOL: a method for isolating memory bottlenecks in shared memory multiprocessor programs. In *International Conference on Parallel Processing*, volume 2, pages 251–257, 1991.

[HC87] A. A. Houch and J. E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *International conference on parallel processing*, pages 735–738, August 1987.

[HS92] Abdelsalam Heddaya and Himanshu Sinha. An overview of Mermera: a system and formalism for non-coherent distributed parallel memory. Technical report, Computer Science Department, Boston University Boston, MA 02215, September 1992.

[KS93] E. Kraemer and J. T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, volume II, pages 94–101, August 1988.

[LMCF90] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analysing parallel program execution using multiple views. *Parallel Distributed Computing*, 9(2):203–217, June 1990.

[LP92] Zakaria Lahjomri and Thierry Priol. KOAN: a shared virtual memory for the iPSC/2 hypercube. In Springer-Verlag, editor, *Parallel Processing : CONPAR 92-VAPV*, pages 441–452, September 1992.

[MR90] A. D. Malony and D. A. Reed. Visualizing parallel computer system performances. *Parallel computer systems*, 1990.

[MY87] B. P. Miller and C. Q. Yan. IPS: an interactive and automatic performance measurement tool for parallel and distributed programs. In *Seventh international conference on distributed computing systems*, University of Wisconsin, September 1987.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: unifying synchronization and data transfer. In *International conference on parallel processing*, volume II, pages 160–169, 1989.

[RCE92] David F. Robinson, Betty H. C. Cheng, and Richard J. Enbody. A transparent monitoring tool for shared-memory multiprocessors. *IEEE*, pages 227–232, 1992.

[RM93] Michel Raynal and Masaaki Mizuno. How to find his way in the jungle of consistency criteria for distributed object memories (or how to escape from minos' labyrinth). Technical Report 1962, INRIA, IRISA, Rennes, July 1993.

[SG92] S. R. Sarukka and D. Gannon. Peformance visualization of parallel programs using SIEVE. In *International conference on supercomputing*, pages 157–166, Washington D. C., July 1992.

[Str90] Per Strenström. A survey of cache coherence schemes for multiprocessors. *IEEE computer*, 23(6):12–24, June 1990.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE computer*, 25(8):10–19, August 1992.

[TV94a] Bernard Tourancheau and Xavier-François Vigouroux. Parallel trace file management on top of PVM. In *PVM UG*, Oak Ridge, TN, 1994.

[TV94b] Bernard Tourancheau and Xavier-François Vigouroux. PIMSy – a parallel trace file analyzer. In IEEE computer society press, editor, *Scalable High-Performance Computing conference*, Knoxville, Tennessee, May 1994.

Bibliographie Personnelle

Revues internationales

- "**Parallel Programming on top of DSM Systems : An Experimental Study**" - A paraître dans la revue Parallel Computing, Début 1997.
- "**Monitoring Distributed Shared Memory Applications: concepts, protocols and experiments**" avec L. Brunie et O. Reymann - Soumis à la revue Journal of Parallel and Distributed Computing, Septembre 1996

Conférences d'audience internationale avec comité de lecture et publication des actes

- "**DOSMOS : A Distributed Shared Memory based on PVM**", avec L. Brunie - First european PVM users group meeting - Rome 9-10 Octobre 1994.
- "**An optimized and load-balanced portable parallel zbuffer**", avec Henri-Pierre Charles et Serge Miguet - IST SPIE Symposium on Electronic Imaging - SPIE Proceedings Vol 2410 "Visual Data Exploration and Analysis II" - Pages 394-403 - ISBN : 0-8194-1757-2 - San Jose, Californie USA, 5-10 Février 1995.
- "**The LHPC Programming Environment**", avec L. Brunie, S. Chaumette, M. Cosnard, F. Desprez, M. Loi, M. Pourzandi, B. Tourancheau et X. Vigouroux - Proceedings of the Second Workshop on Environnments and Tools for Parallele Scientific Computing - SIAM - ISBN 0-89871-343-9 - 1995
- "**PVM implementation for low-level image processing systolic array designs**" avec S.A. Amin et D.J. Evans - Proceedings of the Second European PVM Users' Group Meeting - Lyon 14-15 Septembre 1995
- "**New propositions to improve the efficiency and scalability of DSM systems**" avec L. Brunie - 1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing - IEEE ICA3PP'96 - Pages 356-364 - ISBN: 0-7803-3529-5 - Singapoure Juin 1996.

- "**DOSMOS+ : Scalable Distributed Shared Memory Environment including Monitoring Facilities**" avec L. Brunie et O. Reymann - ESPPE : Parallel Programming Environments for High Performance Computing, pages 165-168 - Alpes d'Huez, Avril 1996.
- "**Execution Analysis of DSM Applications: A Distributed and Scalable Approach**" avec L. Brunie et O. Reymann - SPDT'96 : SIGMETRICS Symposium on Parallel and Distributed Tools, ACM Press editor FCRC, pages 51-60 - Philadelphie, Pennsylvanie, USA, Mai 1996.
- "**Monitoring and performance evaluation of distributed shared memory applications**" avec L. Brunie et O. Reymann - Second International Conference on Massively Parallel Computing Systems, Euromicro, pages 382-389 - Ischia, Italie, Mai 1996.
- "**Integration of Performance Evaluation Facilities into Distributed Shared Memory Based Programming Environments**" avec L. Brunie et O. Reymann - TDP '96: Telecommunication Distribution Parallelism - Agelonde, La Londe Les Maures, 26-28 Juin 1996
- "**A DSM-based Structural Programming Environment for Distributed and Parallel Processing**" avec L. Brunie - HiPC '96 : 3rd International Conference on High Performance Computing - A paraître aux éditions IEEE Computer Society - Trivandrum, Inde, Décembre 1996

Conférences d'audience nationale avec comité de lecture

- "**Une mémoire distribuée-partagée pour machine massivement parallèle**" - Journées 1995 du Site Experimental en Hyperparallelisme - Arcueil Janvier 1995

Conférences d'audience nationale avec comité de lecture et publication des actes

- "**Un modèle de mémoire distribuée-partagée pour machine massivement parallèle**" - Actes de conférence de Renpar 6 - Lyon 7-10 Juin 1994.
- "**Extensibilité et systèmes de mémoire distribuée virtuellement partagée**" avec L. Brunie et O. Reymann - MPR '96: Journées de Recherche sur La Mémoire Partagée Répartie - Bordeaux, 6-7 Mai 1996

Rapports de recherche

- "**An optimized and load-balanced portable parallel zbuffer**", avec H.P. Charles et Serge Miguet - Rapport de recherche LIP n 95-25

– "**Distributed Shared Memory (How to communicate without knowing it ?)**" A paraître.

– "**Un environnement de programmation pour systèmes distribués et parallèles basé sur une mémoire distribuée virtuellement partagée**" En cours de révision.

Bibliographie

[15992] 1596-1992 (IEEE Std). – The SCI standard. – 1992. IEEE Service Center.

[ABHN91] Ahamad (M.), Burns (J. E.), Hutto (P. W.) et Neiger (G.). – Causal memory. *In: 5th International Workshop on Distributed Algorithms*, éd. par Springer-Verlag, pp. 9–30. – Delphi, Greece, 1991.

[ABHN92] Ahamad (Mustaque), Burns (James E.), Hutto (Phillip W.) et Neiger (Gil). – Causal memory. *IEEE Computer*, 1992, pp. 9–29.

[ABLN85] Almes (G.T.), Black (A.P.), Lazowska (E.D.) et Noe (J.D.). – The Eden system: A technical review. *IEEE Transactions on Software Engineering*, vol. 11 (1), janvier 1985, pp. 43–59.

[ACD+96] Amza (Cristiana), Cox (Alan L.), Dwarkadas (Sandhya), Keleher (Pete), Lu (Honghui), Rajamony (Ramakrishnan), Yu (Weimin) et Zwaenepoel (Willy). – TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, février 1996, pp. 18–28.

[ACG86] Ahuka (S.), Carriero (N.) et Glernter (D.). – LINDA and friends. *IEEE Computer*, vol. 19, n° 8, août 1986, pp. 26–34.

[AH90] Adve (Sarita V.) et Hill (Mark D.). – Implementing Sequential Consistency in Cache-Based Systems. *In: International conference on parallel processing*, pp. 47–50.

[AJL92] Amaral (Paulo), Jacquemot (Christian) et Lea (Rodger). – A model for persistent shared memory addressing in distributed systems. *In: International Workshop of Object Oriented Operating Systems IWOOOS '92.* – IEEE Computer Society.

[AM96] André (F.) et Mahéo (Y.). – Programmation distribuée avec partage de tableaux: la bibliothèque cidre. *In: MPR '96: Journées de recherche sur la mémoire partagée répartie.* – Université de Bordeaux, mai 1996.

[BBLP91] Badouel (Didier), Bouatouch (Kadi), Lahjomri (Zakaria) et Priol (Thierry). – *KOAN: A versatile tool for parallelizing realistic rendering algorithms.* – Rapport technique, Institut de recherche en informatique et systemes aleatoires, juillet 1991.

[BBP94] Badouel (Didier), Bouatouch (Kadi) et Priol (Thierry). – Distributing data and control for ray tracing in parallel. *In: IEEE Computer graphics and applications*, pp. 69–77. – IEEE, juillet 1994.

[BCZ90] Bennet (J.K.), Carter (J.B.) et Zwaenepoel (W.). – Munin; distributed shared memory based on type-specific memory coherence. *In: 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel programming.*

[BEP93] Bodin (François), Erhel (Jocelyne) et Priol (Thierry). – Parallel sparse matrix vector multiplication using a shared virtual memory environment. *In: 6th SIAM Conference on parallel processing for scientific computing.* – Norfolk, Virginia (USA), mars 1993.

[BF88] Bisiani (R.) et Forin (A.). – Multilanguage parallel programming of heterogenous machines. *IEEE Transaction on Computers*, vol. 37, n° 8, Aug 1988, pp. 930–945.

[BGL+93] Berrendorf (Rudolf), Gerndt (Michael), Lahjomri (Zakaria), Priol (Thierry) et d'Anfray (Philippe). – *Evaluation of numerical applications running with shared virtual memory.* – Rapport technique n° KFA-ZAM-IB-9315, FORSCHUNGSZENTRUM J LICH GmbH Allemagne, octobre 1993.

[BGLP95] Berrendorf (Rudolf), Gerndt (Michael), Lahjomri (Zakaria) et Priol (Thierry). – *Shared Virtual Memory and Message Passing Programming on a Finite Element Application.* – Rapport technique n° 2355, INRIA, 1995.

[BH89] Borrmann (Lothar) et Herdieckerhoff (Martin). – Parallel processing performance in a Linda system. *In: International conference on parallel processing.* pp. 151–158. – Pennsylvania State University Press.

[BH90] Borrmann (Lothar) et Herdieckerhoff (Martin). – A coherency model for virtually shared memory. *In: International conference on parallel processing.* pp. 252–257. – Pennsylvania State University Press.

[BKP93] Bodin (F.), Kervella (L.) et Priol (T.). – Fortran-s: A fortran interface for shared virtual memory architectures. *In: Supercomputing 93.* – Portland, novembre 1993.

[BLR96] Brunie (Lionel), Lefèvre (Laurent) et Reymann (Olivier). – Execution Analysis of DSM Applications: A Distributed and Scalable Approach. *In: SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, éd. par Press (ACM). FCRC, pp. 51–60. – Philadelphia, Pennsylvania, USA, mai 1996.

[BMP93] Bouatouch (Kadi), Menard (Daniel) et Priol (Thierry). – Parallel radiosity using a shared virtual memory. *In: First Bilkent computer graphics conference on advanced techniques in animation ,rendering and visualization.* – Ankara, Turkey, juillet 1993.

[BPR96] Badouel (Didier), Priol (Thierry) et Renambot (Luc). – Svmview : a performane tuning tool for dsm-based parallel computers. *In: Euro-Par '96 Parallel Processing*, éd. par Springer, pp. 98–105.

[BT88] Bal (H. E.) et Tanenbaum (A. S.). – Distributed programming with shared data. *In: IEEE International conference on Computer Languages*, pp. 82–91.

[BZ91] Bershad (Brian N.) et Zekauskas (Matthew J.). – *Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors.* – Rapport technique n° CMU-CS-91-170, Pittsburgh, School of Computer science Carnegie Mellon University, 1991.

[BZS93] Bershad (Brian N.), Zekauskas (Matthew J.) et Sawdon (W. A.). – The midway distributed shared memory system. *In: 38th IEEE International Computer Conference (COMPCON Spring'93)*, pp. 528–537.

[CAL+89] Chase (J.S.), Amador (F.G.), Lazowska (E.D.), Levy (H.M.) et Littlefield (R.J.). – The Amber System : Parallel programming on a network of multiprocessors. *12th ACM Symposium on Operating Systems Principles*, décembre 1989, pp. 147–158.

[CCGST96] Calliet (G.), Cornilleau (T.), Gressier-Soudan (E.) et Toinard (C.). – Mémoire partagée répartie à cohérence stricte sur un service d'appartenance à des groupes. *In: MPR '96 : Journées de recherche sur la mémoire partagée répartie.* – Université de Bordeaux, mai 1996.

[CF89] Cox (Alan L.) et Fowler, Robert (J.). – The implementation of a coherent memory abstraction on a numa multiprocessor : Experiences with platinum. *In: 12th ACM Symposium on Operating Systems Principles*, pp. 32–44. – Litchfield Park, AZ, dec 1989.

[CF94] Corbel (Annie) et Fleter (Frank). – Une implémentation de linda dans un environnement hétérogène. *In: RenPar'6.* – Ecole Normale Supérieure de Lyon, France, juin 1994.

[CG89] Carriero (Nicholas) et Gerlenter (David). – LINDA in context. *Communications of the ACM*, vol. 32, n° 4, avril 1989, pp. 444–458.

[CG91] Cheriton (David R.) et Goosen (Hendrik A.). – Paradigm: A highly scalable shared-memory multicomputer architecture. *IEEE Computer*, vol. 24, n° 2, février 1991.

[CGCS91] Cohn (D. L.), Greenawalt (P. M.), Casey (M. R.) et Stevenson (M. P.). – Using kernel level support for distributed shared data. *In: SEMDS II Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 247–259. – Atlanta, GA, mar 1991.

[CIJ+91] Campbell (R. H.), Islma (Mayeem), Johnson (Ralph), Kougiouris (Panos) et Madany (Peter). – Choices, frameworks and refinement. *In: International Workshop on Object Orientation in Operating Systems*, pp. 9–15. – Palo Alto, C.A., oct 1991.

[CLM95] Charles (Henri-Pierre), Lefèvre (Laurent) et Miguet (Serge). – An optimized and load-balanced portable parallel zbuffer. *In: IST SPIE Symposium on Electronic Imaging*, éd. par SPIE Proceedings, Georges G. Grinstein (Robert F. Erbacher Eds). IST SPIE, pp. 394–403. – San Jose, février 1995.

[Com96] Comprendre (Guide). – Architectures parallèles. *In: Informatiques Magazine*, pp. 96–99. – février 1996.

[CPP94] Cabillic (G.), Priol (T.) et Puaut (I.). – *MYOAN: an implementation of the KOAN shared virtual memory system on the Intel Paragon.* – Rapport technique n° TR 812, Rennes, France, IRISA, mars 1994.

[DAM+90] Dasguta (Partha), Ananthanarayanan (R.), Menon (Sathis), Mohindra (Ajay) et Chen (Raymond). – *Distributed Programming with Objects and Threads in the Clouds System.* – Rapport technique, Georgia Institute of Technology, 1990.

[DCM+90] Dasgupta (P.), Chen (R. C.), Menon (S.), Pearson (M. P.), Ananthanarayanan (R.), Ramachandran (U.), Ahamad (M.), LeBlanc (R. J.), Appelbe (W. F.), Bernabéu-Aubán (J. M.), Hutto (P. W.), Khalidi (M. Y. A.) et Wilkenloh (C. J.). – The design and implementation of the Clouds distributed operating system. *Usenix Computing Systems*, vol. 3, n° 1, décembre 1990.

[Dio96] Dion (Michèle). – *Alignement et distribution en parallelisation automatique.* – Phd96-04, LIP ENS Lyon, janvier 1996.

[DLAR91] Dasgupta (Partha), LeBlanc (Richard J.), Ahamad (Mustaque) et Ramachandran (Umakishore). – The clouds distributed operating system. *In: IEEE Computer*, éd. par IEEE, pp. 34–44.

[DRDS+91] Delgado-Rannauro (Sergio A.), Dorochevsky (Michel), Schuerman (Kees), Véron (André) et Xu (Jiyang). – A shared environment parallel logic programming system on distributed memory architectures. *In: Distributed Memory Computing*, éd. par Bode (Arndt). – Munich, avril 1991.

[DSB88] Dubois (M.), Scheurich (C.) et Briggs (F. A.). – Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, vol. 21, n° 2, février 1988, pp. 9–21.

[EJ93] Engelmann (Curd) et Jörg (Keller). – Simulation-based comparison of hash functions for emulated shared memory. pp. 1–11.

[Esk96] Eskicioglu (M. Rasit). – A comprehensive bibliography of distributed shared memory. *ACM Operating Systems Reviews*, vol. 30, n° 1, jan 1996, pp. 71–96.

[For93] Forum (Hich Performance Fortran). – *High Performance Fortran Language Specification.* – Rapport technique, Rice University, janvier 1993.

[FP89] Fleisch (Brett D.) et Popek (Gerald J.). – Mirage: A coherent distributed shared memory design. *In: Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, éd. par PRESS (ACM), pp. 211–223. – The Wigwam Litchfield Park, Arizona, décembre 1989.

[FS94] Ferreira (Paulo) et Shapiro (Marc). – *Garbage Collection of Persistent Objects in Distributed Shared Memory.* – Rapport technique, INRIA, mai 1994.

[GBD+93] Geist (A.), Beguelin (A.), Dongarra (J.), Jiang (W.), Manchek (R.) et Sunderam (V.). – *PVM3 User's Guide and Reference Manual.* – Oak Ridge National Laboratory, Oak Ridge, Tennesse, mai 1993.

[GLL+89] Gharachorloo (K.), Lenoski (D.), Laudon (J.), Gibbons (P.), Gupta (A.) et Hennessy (J.). – Memory consistency and event ordering in scalable shared-memory multiprocessors. *In: 16th Annual Symposium on Computer Architecture*, pp. 15–26.

[HA90] Hutto (P. W) et Ahamad (M.). – Slow memory: weakening consistency to enhance concurrency in distributed shared memories. *In: 10th IEEE International Conference on Distributed Computing Systems*, pp. 9–30. – Paris, Frence, juin 1990. Also available as Georgia Institute of Technology report GIT-ICS-89/39.

[Hah93] Hahad (Mounir). – Factorisation de matrices creuses sur une mémoire virtuelle partagée. *Le lettre du transputer et des calculateurs parallèles*, no20, décembre 1993, pp. 25–43.

[Hel90] Hellwagner (Hermann). – *Survey of virtually shared memory systems.* – Rapport technique, Munich, septembre 1990.

[Hel92] Hellwagner (Hermann). – On the practical efficiency of randomized shared memory. *In: Parallel Processing: CONPAR 92-VAPV*, éd. par Springer-Verlay, pp. 429–440.

[HERV93] Heiser (Gernot), Elphinstone (Kevin), Russell (Stephen) et Vochteloo (Jerry). – *Mungi: A distributed address-space operating system.* – Rapport technique, School of computer science and engineering. The university of New South Wales, novembre 1993.

[HS92a] Heddaya (Abdelsalam) et Sinha (Himanshu). – *An implementation of MERMERA: a shared memory system that mixes coherence with non-coherence.* – Rapport technique, Boston University, Boston, MA 02215, Computer Science Department, octobre 1992.

[HS92b] Heddaya (Abdelsalam) et Sinha (Himanshu). – *An Overview of Mermera: A System and Formalism for Non-coherent Distributed Parallel Memory.* – Rapport technique, Boston University Boston, MA 02215, Computer Science Department, septembre 1992.

[HS93] Heddaya (Abdelsalam) et Sinha (Himanshu). – An Overview of MERMERA: A System and Formalism for Distributed Parallel Memory. *In: 26th Hawaiian International Conference on System Sciences (HICSS-26).* – Maui, Hawaii, janvier 1993.

[HT89] Hsu (M.) et Tam (V. O.). – *Transaction synchronisation in Distributed Shared Virtual Memory Systems.* – Rapport technique n° TR-05-89, Aiken Computation Laboratory, Harvard University, Cambridge, Jan 1989.

[Ive62] Iverson (K.). – *A programming language.* – New York, Wiley, 1962.

[JC89] Johnston (G. M.) et Campbell (R. H.). – An object-oriented implementation of distributed virtual memory. *In: Workshop on Experiences with Distributed and Multiprocessor Systems*, éd. par Association (USENIX), pp. 39–57. – Fort Lauderdale, oct 1989.

[JLHB88] Jul (E.), Levy (H.), Hutchinson (N.) et Black (A.). – Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, vol. 6 (1), février 1988, pp. 109–133.

[KCG+95] Kermarrec (A-M), Cabillic (G.), Gefflaut (A.), Morin (C.) et Puaut (I.). – A recoverable distributed shared memory integrating coherence and recoverability. *In: 25th International Symposium on Fault Tolerant Computer Systems.* – Pasadena, USA, juin 1995.

[KCZ92] Keleher (P.), Cox (A.L.) et Zwaenepoel (W.). – *Lazy release consistency for software distributed shared memory.* – Rapport technique, Rice University, mars 1992.

[Ken92] Kendall Square Research. – *KSR1 Principles of Operations*, 1992, waltham, mass édition.

[KLS+94] Koebel (Charles H.), Loveman (David B.), Schreiber (Robert S.), Steele (Guy L.) et Zosel (Mary E.). – *The High Performance Handbook.* – 1994.

[KMR90] Koebel (C.), Mehrotra (P.) et Rosendale (J. V.). – Supporting shared data structures on distributed memory architectures. *In: Second ACM SIGPLAN Symposium on principles and practices of parallel programming*, pp. 177–186.

[Lam78] Lamport (L.). – Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, vol. 21, n° 7, juillet 1978, pp. 558–565.

[Lam79] Lamport (L.). – How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, vol. C-28, n° 9, septembre 1979, pp. 690–691.

[LBR94a] Le Barazer (Vincent) et Roth (Gunter). – *Applications sur CRAY T3D.* – Cray Research France, septembre 1994.

[LBR94b] Le Barazer (Vincent) et Roth (Gunter). – *Applications sur Cray T3D. : Tome IV : Documentation SHMEM et PVM 3.3.* – Rapport technique, Cray Research France, 1994.

[LDCZ95] Lu (Honghui), Dwarkadas (Sandhya), Cox (Alan L.) et Zwaenepoel (Willy). – *Message Passing Versus Distributed Shared Memory on Networks of Worksstations.* – Rapport technique, Rice University, Houston, Texas, 1995.

[LH86] Li (K.) et Hudak (P.). – Memory coherence in shared virtual memory systems. *In: 1986 5th Ann. Symp. on principles of distributed computing*, pp. 229–239.

[LH89] Li (Kai) et Hudak (Paul). – Memory coherence in shared virtual memory systems. *In: ACM Transactions on Computer Systems*, pp. 321–359. – ACM PRESS, novembre 1989.

[Li86] Li (Kai). – *Shared Virtual Memory on Loosely-coupled Multiprocessors.* – Technical report yaleu-rr-492, Yale University, octobre 1986.

[Li88] Li (Kai). – IVY: A shared virtual memory system for parallel computing. *In: 1988 International Conference on Parallel Processing*, pp. 94–101.

[Lil93] Lilja (David J.). – Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, vol. 25, n° 3, septembre 1993, pp. 303–338.

[LP92] Lahjomri (Zakaria) et Priol (Thierry). – Koan : a shared virtual memory for the ipsc/2 hypercube. *In: Parallel Processing: CONPAR 92-VAPV*, éd. par Springer-Verlay, pp. 441–452.

[LS88] Lipton (R.J.) et Sandberg (J.S.). – *PRAM: a scalable shared memory.* – Rapport technique n° CS-TR-180-88, Princeton University Department of Computer Science, septembre 1988.

[LS89] Li (K.) et Schaefer (R.). – A hypercube shared virtual memory system. *In: 1989 International conference on parallel Processing*, pp. 125–131.

[LSB92] Le Sergent (Thierry) et Berthomieu (Bernard). – *Incremental Multi-threaded Garbage Collection on Virtually Shared Memory Architectures*, pp. 179–199. – 1992.

[MAT95] MATRA CAP SYSTEMS. – *Manuel d'utilisation de la machine CAPITAN*, 1995.

[MF92] Murer (Stephan) et Färber. – A scalable distributed shared memory. *In: Parallel Processing: CONPAR 92-VAPV*, éd. par Springer-Verlay, pp. 453–466.

[MR91] Miguet (S.) et Robert (Y.). – Elastic load-balancing for image-processing algorithms. *In: First International Conference of the Austrian Center for Parallel Computation.*

[NL91] Nitzberg (Bill) et Lo (Virginia). – Distributed shared memory: a survey of issues and algorithms. *IEEE computer*, vol. 24, n° 8, septembre 1991, pp. 52–60.

[OMW+92] Osmon (Peter), Murray (Kevin), Whitcroft (Andy), Wilkinson (Tim) et Williams (Nick). – *Network Shared Memory.* – Rapport technique, Systems Architecture Research Center, City University, octobre 1992.

[PBR91] Puaut (Isabelle), Banatre (Michel) et Routeau (Jean-Paul). – Early experiences with the gothic distributed operating system. *In: SEMDS II Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 271–282. – Atlanta, GA, mar 1991.

[PL92] Priol (Thierry) et Lahjomri (Zakaria). – *Trade-offs between shared virtual memory and message-passing on an iPSC2 Hypercube.* – Rapport technique, Institut de recherche en informatique et systemes aleatoires, février 1992.

[Rai90] Raina (S.). – Software controled shared virtual memory management on a transputer based multiprocessor. *In: 4th North American Transputers Users Group Conference.*

[RAK89] Ramachandran (Umakishore), Ahamad (Mustaque) et Khalidi (M. Yousef A.). – Coherence of distributed shared memory: unifying synchronization and data transfer. *In: 1989 International conference on parallel processing*, pp. 160–169.

[Ray92] Raynal (Michel). – *Gestion des données réparties: problèmes et protocoles.* – 1992.

[Rey94] Reymann (Olivier). – Dosmos-trace: outils de visualisation pour mémoire distribuée virtuellement partagée. – juin 1994. Rapport de stage de DEA.

[RM93] Raynal (Michel) et Mizuno (Masaaki). – *How to find his way in the jungle of consistency criteria for distributed object memories (or how to escape from Minos' labyrinth).* – Rapport technique n° 1962, IRISA, Rennes, INRIA, juillet 1993.

[RMN92] Raynal (Michel), Mizuno (Masaaki) et Neilsen (Mitch). – *Causality oriented shared memory for distributed systems.* – Rapport technique n° 656, IRISA, Rennes, IRISA, avril 1992. Publication interne.

[SF95] Shapiro (Marc) et Ferreira (Paulo). – Larchant-RDOSS: a distributed shared persistent memory and its garbage collector. *In: WDAG, Workshop on Distributed Algorithms*, éd. par Hélary (J.-M.) et Raynal (M.), pp. 198–214. – Le Mont Saint-Michel (France), septembre 1995.

[SH95] Shang (Shisheng) et Hwang (Kai). – Distributed hardwired barrier synchronization for scalable multiprocessor clusters. *In: IEEE Transactions on Parallel and Distributed Systems*, pp. 591–603. – IEEE, juin 1995.

[SPFA94] Shapiro (Marc), Plainfossé (David), Ferreira (Paulo) et Amsaleg (Laurent). – Some key issues in the design of distributed garbage collection and references. *In: Unifying Theory and Practice in Distributed Systems.* – Dagstuhl (Germany), septembre 1994.

[Str90] Strenström (Per). – A survey of cache coherence schemes for multiprocessors. *IEEE computer*, vol. 23, n° 6, juin 1990, pp. 12–24.

[Sys92] Systems (Trollius). – *Brenda User Guide.* – Rapport technique, Ohio Supercomputer Center, The Ohio State University, jul 1992.

[SZ90] Stumm (Michael) et Zhou (Songnian). – Algorithms implementing distributed shared memory. *IEEE computer*, vol. 23, n° 5, mai 1990, pp. 54–64.

[Tan91] Tanenbaum (Andrew). – *Architecture de l'ordinateur.* – InterEditions, 1991, intereditions édition.

[Thi90] Thinking Machine Corporation. – *Programming in C**, 1990.

[TKB92] Tanenbaum (Andrew S.), Kaashoek (M. Frans) et Bal (Henri E.). – Parallel programming using shared objects and broadcasting. *IEEE computer*, vol. 25, n° 8, août 1992, pp. 10–19.

[VW93] V. Wilson (Gregory). – A Glossary of Parallel Computing Terminology. *In : Parallel & Distributed Technology - Systems & Applications*, pp. 52–67. – IEEE Computer Society, février 1993.

[WCF+93] Wood (David A.), Chandra (Satish), Falsafi (Babak), Hill (Mark D.), Larus (James R.), Lebeck (Alvin R.), Lewis (James C.), Mukherjee (Shubhendu S.), Palacharla (Subbarao) et Reinhardt (Steven K.). – Mechanisms for cooperative shared memory. *In : 20th Annual International Symposium on Computer Architecture*, éd. par IEEE.

[WF90] Wu (K.L.) et Fuchs (W.K.). – Recoverable distributed shared virtual memory. *In : IEEE Transactions on computers.* – IEEE, avril 1990.

[WLJI+93] Wilson (Andrew W.), LaRowe Jr (Richard P.), Ionta (Robert J.), Valentino (Ralph P.), Hu (Beeching), Breton (Peter R.) et Lau (Pocheong). – *Update propagation in the Galactica Net distributed shared memory architecture.* – Rapport technique, Center for High performance computing Worcester Polytechnic institute, Malborough, MA 01752, juillet 1993.

[WP90] Winterbottom (P.) et P. (Osmon). – Topsy : an extensible UNIX multicomputer. *In : UK IT90 Conference (Southampton University)*, pp. 164–176.

[WSG+92a] Wilkinson (T.), Stiemerling (A.), Gull (A.), Whitcroft (P.E.), Osmon (P.), Saulsbury (A.) et Kelly (P.). – Angel : a proposed multiprocessor operating system kernel. *In : European Workshop on Parallel Computing.*

[WSG+92b] Wilkinson (T.), Stiemerling (A.), Gull (A.), Whitcroft (P.E.), Osmon (P.), Saulsbury (A.) et Kelly (P.). – *Angel: a proposed multiprocessor operating system kernel.* – Rapport technique n° TCU/CS/1992/10, City University, London, mars 1992.

[WY94] Wen-Yew (Liang). – *ADSMITH: A structure-based heterogeneous distributed shared memory on PVM.* – Thèse de PhD, Institute of computer science National tsing hua university, juin 1994.

[Zho92] Zhou (S.). – An heterogeneous distributed shared memory system. *In : IEEE Transactions on Parallel and Distributed Systems.*

[ZMS93] Zhou (James Z.), Mizuno (Masaaki) et Singh (Gurdip). – *A sequentially consistent distributed shared memory.* – Rapport technique, Kansas state university, 1993.