Rapport de Stage MIM 1
Fusion de deux environnements de programmation parallèle : Dosmos et PM^2

Aumage Olivier

11 septembre 1997

Table des matières

Chapitre 1

Introduction

Un formidable potentiel de parallélisme Durant les 15 dernières années, l'informatique du parallélisme a connu une évolution considérable. En effet, il y a encore peu de temps, la programmation parallèle n'était envisageable que sur des machines conçues dans ce but, chères, à courte durée de vie, et donc peu répandues. L'impressionnante — et récente — montée en puissance des stations de travail et des micro-ordinateurs a ouvert de nouvelles perspectives à l'informatique parallèle grâce à l'idée suivante : utiliser les stations de travail d'un réseau comme autant de processeurs d'une machine parallèle virtuelle. Une idée séduisante à plus d'un titre : la quasi totalité des universités, entreprises et institutions du monde entier possèdent un, voire plusieurs réseaux, les stations sont plus souples d'utilisation et bien moins onéreuses qu'une machine parallèle et les performances se révèlent plus que séduisantes (et ne devraient pas cesser d'augmenter dans les années à venir). Plus encore, l'apparition d'environnements multitâches, et surtout d'environnements multithreadés, offre une forme de parallélisme au sein même d'un seul processeur. L'informatique du parallélisme devient accessible.

Des outils pour maîtriser ce potentiel Dès lors, le problème de la maîtrise de ces possibilités se pose, car rares sont les programmeurs capables de concevoir seuls un logiciel applicatif utilisant plus de quelques processeurs. Il apparaît donc indispensable de seconder le travail de conception de l'informaticien, ce que proposent les environnements de programmation parallèle. Leur rôle est de fournir un ensemble de services visant à aider le programmeur à exprimer son algorithme de manière parallèle, tout en offrant une certaine abstraction (plus ou moins importante selon les environnements) vis-à-vis de la structure matérielle sous-jacente. Parmi ces environnements, deux nous intéressent particulièrement : PM^2 et Dosmos :

- PM^2 (Parallel Multithreaded Machine) est un système de programmation parallèle basé sur le multithreading et sur le principe de l'appel de procédure à distance léger créé par R.Namyst [`NAM97`] et J.F.Méhaut de l'équipe ESPACE de Lille. L'idée est ici d'exprimer le parallélisme en remplaçant les appels de fonctions traditionnels par des appels distants avec création de threads.
- Dosmos (Distributed Objects Shared MemOry System), lui, se classe parmi les systèmes de mémoire distribuée virtuellement partagée. Initialement conçu par L.Lefevre [`LEF97`] du LIP, il est actuellement développé par O.Reymann, également du LIP. Le but de cet environnement est de prendre en charge le contrôle de structures de données partagées par plusieurs processus, et ceci de manière transparente pour le programmeur.

L'union dans la complémentarité Les approches de ces deux environnements sont donc très différentes. Il est d'ailleurs important de remarquer ceci : si PM^2 offre de puissants services concernant le contrôle du programme, il délaisse totalement la question des données partagées. Et inversement, les compétences de Dosmos ne vont pas au-delà de la prise en charge du partage des objets ; il manque à cet environnement la possibilité d'agir sur l'exécution du programme. Or, s'il est une équation fondamentale en informatique, c'est bien celle-ci :

$$\boxed{\textbf{Programme} = \textbf{Contrôle} + \textbf{Données}}$$

L'idéal serait donc de disposer d'un environnement alliant les fonctionnalités de PM^2 en matière de contrôle de programme aux services de gestion de données partagées de Dosmos. C'est précisément l'objet de ce stage.

Chapitre 2

Prise en main des environnements

2.1 PM^2

L'environnement de programmation parallèle PM^2, développé dans le cadre du projet ESPACE du LIFL par R.Namyst [NAM97] et J.F.Méhaut de Lille, fournit un modèle de programmation reposant sur l'utilisation d'appels de procédures à distance légers (souvent appelés LRPC, pour Lightweight Remote Procedure Call). Voyons plus précisément en quoi cela consiste.

2.1.1 Threads et appels de procédures à distance légers

PM^2 permet de créer des procédures particulières nommées *services*, et pouvant ultérieurement être appelées par l'intermédiaire de primitives. Ces primitives prennent en paramètre un identificateur indiquant le processus de la machine virtuelle de PM^2 (c'est-à-dire la machine parallèle virtualisant la configuration physique réelle) sur lequel le service doit être exécuté. C'est là la différence majeure avec un appel de fonction classique, et la raison de leur nom d'*appel de fonction à distance*. Lors de l'appel à distance, le processus cible crée une thread chargée d'exécuter le service (Le qualificatif *léger* est dû au fait que l'appel distant génère ici une thread, par opposition aux appels distants classiques (services RPC) qui créent des processus, par définition bien plus lourds que des threads). Suivant la primitive d'appel utilisée, la thread appelante peut être bloquée en attente du résultat du service, ou bien continuer son exécution *en parallèle* et éventuellement récupérer le résultat plus tard. Au niveau interne, les LRPC sont directement pris en charge par PM^2 lui-même, par contre la gestion des threads est sous-traitée à la librairie multithread Marcel (elle-même développée par les concepteurs de PM^2), et les communications inter-processus sont gérées par la librairie PVM.

Afin de mieux comprendre toutes ces notions, regardons la réalisation concrète d'un programme sous PM^2, en l'occurence une implémentation de l'algorithme du tri fusion.

2.1.2 Application

Dans une implémentation classique, nous utiliserions une procédure récursive divisant le tableau en deux parties, triant les deux moitiés, et fusionnant celles-ci. Donc si nous suivons le principe de programmation de PM^2, la solution est tout simplement de remplacer cette procédure par un *service PM^2*, et de substituer une primitive de LRPC à l'appel récursif traditionnel. La réalisation effective de ce service comprend deux étapes :

- un appel distant peut — par définition — avoir pour cible n'importe quel processus de la machine virtuelle PM^2. Il n'est donc pas nécessairement local au processus appelant. Ceci implique l'utilisation d'un mécanisme de communication des paramètres d'entrée du service entre la source et la cible, et dans le sens inverse pour le transfert du(des) résultat(s). PM^2 propose pour cela d'exécuter automatiquement des procédures d'empaquetage et de dépaquetage des données définies par l'utilisateur, lors de l'appel distant et du retour des données. Ces routines sont écrites à l'aide de primitives PM^2 prenant en charge le transfert des types de données fondamentaux (même entre des architectures différentes). Dans le cadre de notre programme, ces fonctions doivent donc assurer le transfert de la zone du tableau à trier à l'appel, et de la même zone triée au retour. La méthode la plus évidente est d'empaqueter la longueur de la zone sous forme d'un entier, puis les données de la zone elle-même, élément par élément. La fonction de dépaquetage dispose ainsi de toutes les données nécessaires à la réception : la taille de la zone de mémoire à allouer, puis les éléments à charger dans cette zone mémoire.
- la deuxième étape est naturellement de coder le service proprement-dit, de manière identique à l'implémentation classique, exception faite, bien sûr, de l'appel récursif, remplacé ici par un LRPC. Si l'optimisation était recherchée, il faudrait prendre un soin tout particulier à l'algorithme déterminant la cible d'exécution de chaque LRPC. Le but de cette implémentation étant simplement d'aider à comprendre le modèle de programmation de PM^2, nous nous contenterons ici d'exécuter tous les LRPC sur le même processus. Cependant, il faut savoir que grâce à la première étape, notre service pourrait — sans modifications — être utilisé dans une implémentation plus travaillée, dotée d'un algorithme visant à équilibrer la charge des machines ou à minimiser le volume des communications, par exemple.

La réalisation d'un programme PM^2 est donc particulièrement simple, et on peut même s'étonner, dans le cas de notre exemple, de ne pas avoir parlé un seul instant de programmation parallèle ! Pourtant, notre programme est bel et bien un authentique *programme parallèle* : chaque exécution du service de tri (exception faite des services de la fin de la récursion) produit deux LRPC, un pour la première moitié de la zone à trier et l'autre pour la seconde moitié. Ces deux nouveaux services sont exécutés sur des threads distinctes créées spécialement. Ils fonctionnent donc concurremment, c'est-à-dire ... potentiellement *parallèlement*!

2.2 Dosmos

L'environnement Dosmos a été conçu et développé par L.Lefevre [LEF97], et plus récemment par O.Reymann, tous deux du LIP. Il propose la gestion d'objets (structures de données) partagés entre plusieurs processus en se fondant sur un système de mémoire distribuée virtuellement partagée (MDVP). Arrêtons-nous un instant sur cette caractéristique importante.

2.2.1 Mémoire distribuée virtuellement partagée

Le but des systèmes de mémoire distribuée virtuellement partagée est de rassembler les avantages des machines parallèles à mémoire partagée (où tous les processeurs accèdent à une même et unique mémoire vive) et des machines parallèles à mémoire distribuée (où chaque processeur possède sa propre mémoire). Le principe est de virtualiser la mémoire de manière à montrer au programme une mémoire distribuée sous la forme d'un seul bloc de mémoire partagé par tous les processeurs.

Dans le cas de Dosmos, la virtualisation est même plus élevée car la notion de mémoire partagée est affinée en notion d'*objet partagé*, plus intuitive. D'un point de vue plus technique, chaque processeur possède une partie de chaque objet partagé. Si un processus veut accéder à un élément d'un objet partagé, et que cet élément est la propriété de son processeur, cela ne pose pas de problèmes. Si — par contre — le processeur local ne possède pas l'élément, le système MDVP entre en communication avec le possesseur effectif de l'élément afin d'en obtenir une copie (voire d'en devenir propriétaire, suivant les options techniques choisies pour la conception du système MDVP). Cette communication est transparente au programme : pour lui, tout se passe comme ci l'élément était local.

2.2.2 Objets partagés, objets systèmes

Dosmos utilise le concept d'objets partagés. Ces objets peuvent, actuellement, être des variables de types simples fondamentaux (caractères, entiers, flottants), ou bien des tableaux (vecteurs ou matrices) d'éléments de ces types. Tandis que les variables sont *atomiques*, les tableaux peuvent être découpés en parties plus petites appelées *objets systèmes* qui seront les unités insécables de transfert entre les processus. Nous avons vu ci-dessus que lorsqu'un élément n'était pas disponible localement, il était rapatrié depuis un autre processus. L'intérêt des objets systèmes est de grouper les rapatriements, et donc de limiter les communications ; leur taille est en outre définie par l'utilisateur, ce qui confère à ce dernier un contrôle sur les performances. Celles-ci sont en effet sensibles au rapport entre le nombre d'objets systèmes (plus ils sont nombreux, plus ils demandent de traitement) et leur taille (plus ils sont grands, plus leur communication est coûteuse, et moins ils offrent de souplesse).

Afin de mieux nous familiariser avec cet environnement tout en gardant une certaine cohérence, voyons à présent une nouvelle implémentation de l'algorithme du tri fusion, mais sous Dosmos cette fois-ci.

2.2.3 Tri fusion sous Dosmos

Nous venons de voir que les communications de mise-à-jour des objets partagés fonctionnent sur la base des objets systèmes. Il est donc important de tenir compte de cette caractéristique dans l'implémentation de l'algorithme sous Dosmos, et c'est pourquoi le tri est découpé en deux phases :

1. au cours de la première, chaque objet système — ou plutôt la zone du tableau correspondante — est trié indépendamment des autres. L'exécution de chacun de ces tris est distribuée circulairement sur tous les processus : si nous supposons qu'il y ait n processus, alors le processus i trie les objets systèmes dont le numéro modulo n vaut i. Notons au passage l'utilisation (naturelle) du parallélisme : les processus se partagent le travail.

2. la seconde phase reçoit, en entrée, un ensemble d'objets systèmes triés. Son but étant de retourner, en sortie, un tableau complètement trié, elle doit donc fusionner le contenu des objets systèmes entre eux. Elle suit pour cela une structure pyramidale : les objets systèmes sont tout d'abord fusionnés par paires de zones consécutives. Puis les zones triées obtenues sont à nouveau fusionnées entre elles (c'est-à-dire par groupe de 4 objets systèmes), et ceci tant qu'il reste plus d'une zone triée. Cette approche pyramidale est intéressante, car elle tient compte, elle-aussi, du découpage en objets systèmes : à chaque étape de la pyramide, un seul processus accède à un objet système donné.

Plus généralement, à chacune des deux phases, et à chaque étape de la pyramide de la seconde phase, un objet système n'est utilisé que par un seul processus. Il serait donc inefficace que pendant ces phases, les différentes copies éventuelles d'un objet système soient mises-à-jour à chaque accès. Dosmos propose donc un mécanisme appelé *barrière de synchronisation*, qui permet d'une part de synchroniser les processus tournant en parallèle, et d'autre part de mettre à jour en une seule fois toutes les copies d'objets. Une barrière de synchronisation vient donc prendre place entre chaque étape du tri.

La programmation sous Dosmos est donc, tout comme PM^2, des plus simples. Elle est d'ailleurs encore plus facilitée par le préprocesseur DCPP de Dosmos qui traduit les instructions d'accès aux objets partagés (strictement identiques à celles utilisées pour des variables et tableaux classiques) en primitives Dosmos.

Chapitre 3

Réalisation

Maintenant que nous avons une idée plus précise des possibilités de Dosmos et PM^2, nous sommes quasiment prêt à aborder la réalisation du nouvel environnement. Toutefois, avant d'aller plus loin, il serait utile de disposer d'un outil permettant de visualiser le comportement des processus et l'état de leurs threads. PM^2 dispose d'une petite interface graphique OpenLook (utilisant la librairie XView) nommée XPM^2 et chargée justement de décrire les processus de la machine virtuelle. Cependant, celle-ci ne permet pas de voir les threads de plusieurs processus simultanément. Nous allons donc l'étendre pour qu'elle supporte cette possibilité.

3.1 L'interface graphique XPM^2

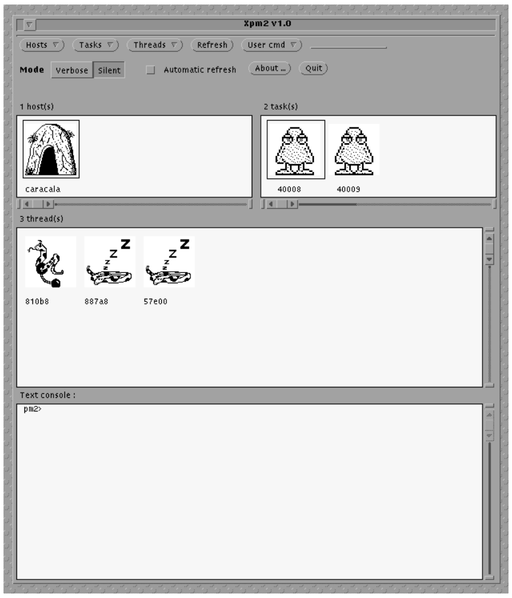


FIG. 3.1 - *L'interface XPM^2.*

L'image 3.1 montre une copie d'écran de l'aspect initial de l'interface XPM^2. L'écran comporte 4 cadres principaux. Le premier, en haut à gauche, contient les icones représentant les stations de la machine virtuelle. Le cadre suivant, à droite indique les processus tournant sur la station sélectionnée. Au milieu de la fenêtre apparaissent les threads du processus possédant le focus de sélection. Enfin, le cadre du bas est réservé à l'interaction avec la console PM². L'idée est donc de supprimer le cadre central et de le remplacer par des panneaux indépendants de la fenêtre principale pouvant être ouverts et fermés par l'utilisateur suivant ses besoins. L'ouverture d'un panneau de threads est commandée par une nouvelle option du menu contextuel des processus, visible sur l'image 3.2.

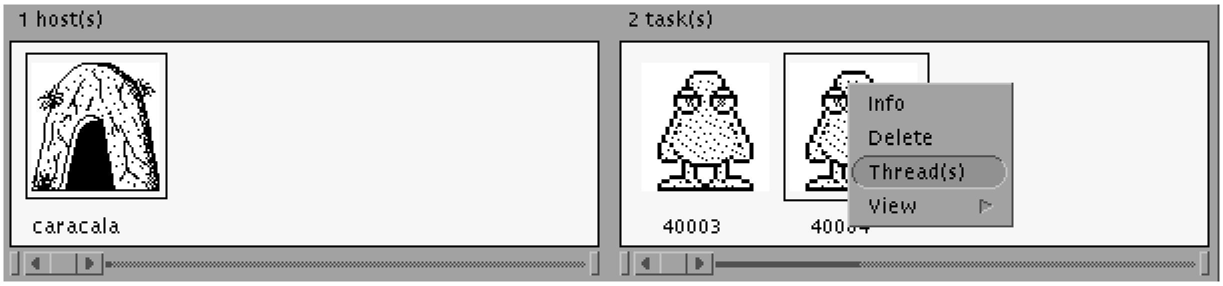


FIG. 3.2 - *Commande d'ouverture d'un panneau de threads.*

Entrer dans des détails plus techniques de la réalisation de cette extension ne présenterait que très peu d'intérêt, et nous n'irons donc pas plus avant. L'image 3.3 montre la version étendue de XPM², avec deux panneaux de threads ouverts autour de la fenêtre principale.

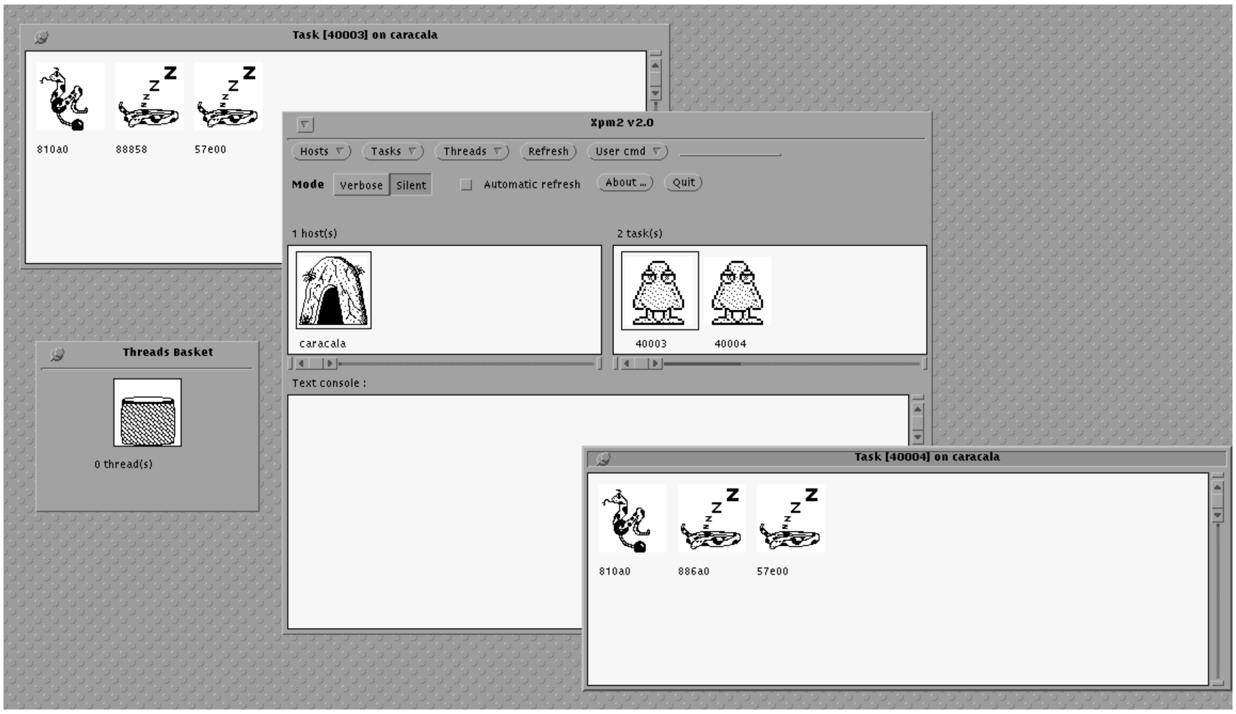


FIG. 3.3 - *Nouvelle version de XPM².*

Nous sommes prêts pour la construction de l'environnement. Cette construction, du fait du gabarit imposant des deux environnements de départ, va se dérouler en deux temps : pour commencer, nous verrons la fusion de Dosmos et de la librairie multithreads Marcel. Ceci nous permettra d'aborder la seconde partie, la fusion entre Dosmos et PM^2 proprement-dite, sur des bases solides.

3.2 Fusion de Marcel et de Dosmos

Marcel est la librairie responsable de la gestion des threads sous PM^2. Cette responsibilité est double : d'une part, Marcel doit fournir les primitives de contrôle des threads telles que la création ou la suppression, et d'autre part, il doit assurer la préemption de ces mêmes threads afin de leur garantir des temps d'exécution équitables (pondérés par les priorités des threads). L'objet de cette étape consiste donc à analyser les problèmes créés par le fonctionnement multithreadé d'un programme Dosmos (rappelons que Dosmos n'a pas été conçu pour être utilisé en environnement multithreads et donc qu'il n'est pas — c'est le terme employé — *MultiThreads-Safe*, ou plus couramment MT-Safe), et bien entendu à leur apporter des solutions.

3.2.1 La librairie PVM et les threads

Le principal responsable de la qualité 'MT-Unsafe' de Dosmos est la librairie de communication PVM. Les structures de données fondamentales utilisées par cette librairie, les buffers d'entrée et de sortie, sont globales à chaque processus. Or l'utilisation de données globales dans un environnement multithreads sans contrôle de l'accès partagé à celles-ci est une source potentielle de conflits. En effet prenons l'exemple d'une thread qui demande l'envoi d'un certain nombre de données. Elle exécute donc la primitive PVM correspondante et elle commence à remplir le buffer d'émission. Supposons qu'à cet instant elle soit préemptée par l'ordonnanceur et que la thread recevant le focus soit elle aussi sur le point d'envoyer des données. Elle appelle donc la primitive PVM adéquate (ce qui peut déjà être délicat car cette primitive n'est certainement pas réentrante), et elle écrit ses propres données dans le buffer, à la suite des précédentes données ou sur ces dernières si le pointeur du buffer a été réinitialisé. Dans les deux cas, le buffer est corrompu.

La solution doit donc, d'une manière ou d'une autre, garantir à chaque threads d'avoir un accès exclusif au buffer d'émission (ou de réception, le cas échéant) du début à la fin d'un envoi (ou d'une réception) de données. La méthode la plus adéquate serait d'utiliser un sémaphore d'exclusion mutuelle (couramment appelé *mutex*) pour chacun des buffers. Cependant, la librairie PVM ne fait pas partie intégrante de Dosmos, ni même de la distribution de Dosmos. Modifier PVM aurait donc pour double conséquence d'obliger à distribuer la nouvelle version avec Dosmos, et surtout de mettre en péril la portabilité de celui-ci. Toute modification au sein de la librairie de communication est donc à proscrire. Le problème doit donc être résolu à partir de Dosmos. Comme nous l'avons vu ci-dessus, la corruption des buffers a pour cause la préemption intempestive d'une thread au milieu d'une primitive PVM. Or, Marcel dispose d'une paire de fonctions appelées `lock_task()` et `unlock_task()`, qui, respectivement, bloquent et débloquent la préemption de la thread active. Notre problème est résolu ! Entourons dans Dosmos chaque primitive PVM à l'aide de ces deux commandes, et les threads ne sont plus préemptées intempestivement.

Tout n'est pas parfait cependant. En effet, les primitives PVM de réception utilisées par Dosmos sont pour la plupart bloquantes, ce qui signifie qu'elles attendent que la réception soit terminée pour rendre la main. Or, comme la préemption est bloquée pendant l'exécution de ces commandes, le programme est complètement arrêté dès qu'une thread attend une donnée. Ceci est particulièrement inefficace, car les autres threads pourraient effectuer des calculs pendant ce temps. Il faut donc retirer toutes les primitive PVM bloquantes de Dosmos, et les remplacer par une boucle d'attente active utilisant les fonctions non-bloquantes équivalentes de PVM, et de déplacer les fonctions `lock_task()` et `unlock_task()` à l'intérieur de la boucle, juste autour de la commande de réception non-bloquante. Tant que la donnée attendue n'est pas reçue, la boucle rend la main (par l'intermédiaire de la commande `pthread_yield()` de Marcel) à l'ordonnanceur qui lui-même passe le contrôle aux autres threads. Par cette méthode, nous recouvrons les phases d'attente de données par du calcul, ce qui est plus performant.

La préemption étant désactivée entre `lock_task()` et `unlock_task()`, tout se passe comme si le programme était monothreadé entre ces deux commandes. Dès lors, la librairie PVM n'est plus utilisée que dans un contexte monothread, dans lequel elle fonctionne parfaitement. Dosmos peut donc maintenant s'appuyer sur des communications sécurisées. Toutefois, il reste un point important à traiter avant que des objets Dosmos puissent être utilisés sans risque en environnement multithreads.

3.2.2 Inversion de données

Pour comprendre ce qui se passe, regardons l'exemple suivant : une thread, que nous numéroterons 1 demande l'élément x d'un tableau Dosmos à un processus distant. Il n'y a plus de problème de corruption de buffer avec PVM, donc la requête est envoyée correctement. A cet instant, l'ordonnanceur préempte la thread 1, et la main est donnée à la thread 2. Celle-ci demande alors l'élément y du tableau au même processus distant. Entre-temps, ce dernier a traité la requête de la thread 1, et renvoie l'élément x. La thread 2 étant en attente d'une valeur, elle reçoit donc cet élément x, en le prenant pour l'élément y. Enfin, après un certain temps, la thread 1 reprend la main, et reçoit l'élément y au lieu de l'élément x. Les éléments ont donc été échangés. Ce problème est nettement plus subtil que le précédent, car contrairement à lui, il ne génère pas de message d'erreur à l'exécution.

La cause profonde de ce disfonctionnement est que Dosmos n'a pas la notion de threads. Or, alors que le destinataire d'une communication devrait être défini au niveau 'thread', sa précision s'arrête au niveau 'processus'. Ce problème pourrait donc être résolu en ajoutant un identificateur de thread dans les communications, mais cette solution nécessiterait de très importants travaux au cœur de Dosmos. Cette approche pourrait être choisie dans un projet ultérieur. En ce qui nous concerne, nous nous contenterons d'une solution moins efficace, mais beaucoup plus simple dans sa mise en œuvre s'appuyant sur l'observation suivante : Dosmos inverse les données d'un objet car il traite les requêtes en supposant qu'elles sont séquentielles, ce qui est vrai en contexte monothread, mais ne l'est plus lorsque l'on fait du multithreading. Forçons les requêtes à être séquentielles, et comme PVM, Dosmos sera confiné dans un "micro-environnement" monothread artificiel, c'est-à-dire dans un contexte où il est prévu pour fonctionner. Concrètement, séquentialiser les requêtes, c'est imposer qu'à chaque instant et dans chaque classe de commandes (envoi, réception, synchronisation, ...), au plus une primitive soit en cours d'exécution. Munissons donc chacune des classes de commandes d'un sémaphore d'exclusion mutuelle. Toutes les primitives Dosmos verrouillerons désormais le mutex correspondant à leur classe avant tout traitement, et le déverrouillerons avant de se terminer, interdisant de ce fait toute nouvelle exécution simultanée d'une commande de la même classe.

Dosmos et PVM évoluent désormais dans leur propre micro-contexte monothread et peuvent donc cohabiter pacifiquement avec Marcel, et nous en arrivons comme prévu à la dernière étape, l'ajout de la couche PM^2 au dessus de Marcel et Dosmos.

3.3 Fusion de PM^2 et de Dosmos

En ce qui concerne Marcel et PM^2, aucun problème en perspective car ils fonctionnent déjà ensemble, et nous pouvons remarquer que nous n'avons pas touché au fonctionnement de Marcel au cours de la phase précédente. Il n'en va pas de même de la cohabitation de Dosmos et PM^2. Assez curieusement, la librairie PVM est une nouvelle fois impliquée dans les difficultés rencontrées.

3.3.1 Des messages ambigus

Nous avons vu au chapitre 2 que PM^2 et Dosmos utilisent la librairie PVM pour prendre en charge leurs communications interprocessus. Ces communications prennent la forme de *messages* possédant un identificateur et un destinataire. Le programmeur est libre de choisir la valeur de l'identificateur de chaque message, et le sens à attribuer à cette valeur. Dans le cas de Dosmos, par exemple, certains messages seront destinés à des requêtes, d'autres serviront d'accusé de réception, etc ... Pour PM^2, ils auront comme principale fonction la demande d'exécution de services distants, et la réception des résultats. Or, si le message de numéro 1 a une certaine définition pour PM^2, il n'y aucune raison pour que ce même message 1 ait une signification identique pour Dosmos. Nous avons déjà remarqué lors de la fusion entre Marcel et Dosmos que le destinataire d'un message n'était spécifié que par le numéro du processus, sans aucune indication concernant la thread cible. De plus, il faut savoir que PM^2 possède sa propre thread de contrôle. Donc, supposons qu'une thread recoive le message 1 précédemment cité ; si la thread est une thread normale, alors c'est Dosmos qui reçoit le message 1 et qui l'interprète à sa manière. Si par contre, au moment où le message est reçu, la thread de contrôle de PM^2 est active, c'est elle qui reçoit et interprète le message 1, différement de Dosmos. Il est évidemment fort gênant que le sens du message change selon la date à laquelle il arrive ! Il est donc indispensable d'attribuer des plages de numérotation de messages différentes pour Dosmos et pour PM^2. Le plus simple est de décaler les valeurs des identificateurs de messages de PM^2 car les numéros de messages de Dosmos sont réservés jusqu'à 1400, alors que PM^2 n'utilise qu'un petit nombre de messages distincts.

Il n'y a ainsi plus d'ambiguïté sur le sens de chaque message. Toutefois, cela ne les empêche pas pour autant de ne pas arriver à la bonne adresse.

3.3.2 Aiguillage des messages

En effet, si nous reprenons notre message de numéro 1. Il a désormais une signification unique, et on sait de plus que c'est un message Dosmos car il est dans la plage réservée de celui-ci (entre 0 et 1400). Mais il peut très bien être reçu quand la thread de contrôle de PM^2 a la main. Dans ce cas, que se passe-t-il? PM^2 ne reconnaît pas le message car les siens ont un numéro supérieur à 1400. Il génère donc une exception qui provoque l'arrêt du programme. Dans le cas où le message est pour PM^2 et que Dosmos le reçoit, l'exécution est terminée de manière encore plus brutale car Dosmos n'utilise pas de mécanisme d'exception. La solution serait de pouvoir filtrer les messages juste avant la réception pour que Dosmos ne reçoive que des messages Dosmos, et de même pour PM^2. Dans ce but, la librairie PVM propose justement la fonction `pvm_recvf` qui permet d'installer une routine de tri des messages en amont de la réception. La routine en question reçoit en paramètre l'identificateur du message arrivant. Elle doit retourner une valeur indiquant si le message doit être effectivement lu, ou au contraire remis dans la file des messages pour être traité ultérieurement. En ce qui nous concerne, la logique de cette routine est on ne peut plus simple. Trois cas sont possibles :

- la thread de contrôle PM^2 n'a pas la main et le message a un identificateur inférieur à 1400 : le message est pour Dosmos et peut donc être lu.
- la thread de contrôle PM^2 a la main et le message a un numéro supérieur ou égal à 1400 : le message est pour PM^2 et peut-donc être lu.
- la configuration ne correspond pas aux deux cas de figure précédents : le message ne doit pas être lu.

Comme indiqué ci-dessus, la routine de tri reçoit automatiquement le numéro du message en paramètre. Pour assurer sa tâche, elle doit en outre disposer d'un drapeau signalant si la thread active est la thread de contrôle PM^2 ou non. Ce drapeau — une variable globale de type entier — est initialisé à la valeur 0 (Faux). Quand la thread de contrôle PM^2 reçoit la main, elle place l'indicateur à 1 (Vrai), puis elle le remet à 0 quand elle rend la main. De cette façon, la routine d'aiguillage des messages sait à tout instant qui de Dosmos ou de PM^2 demande le message et les deux environnements peuvent ainsi recevoir convenablement les messages qui leur sont destinés, et seulement ceux-ci.

Plus rien ne s'oppose donc dorénavant à ce que PM^2 et Dosmos coopèrent, dans le calme et la discipline, au sein d'un seul et unique environnement. Et nous n'attendrons d'ailleurs pas un instant de plus pour l'essayer ...

Chapitre 4

Mise en œuvre

Nous nous proposons donc maintenant d'essayer sur un exemple notre environnement flambant neuf. Parmi les innombrables algorithmes candidats à être implémentés sur celui-ci, l'un d'eux semble, toujours dans un esprit de cohérence, s'imposer ici : le *tri fusion* !

4.1 Le tri fusion et l'environnement fusionné

Cette dernière implémentation du tri fusion s'inspire en grande partie de celle que nous avons réalisée pour PM^2 seul, c'est-à-dire qu'elle utilise elle aussi un *service "tri"*. En effet, le nouvel environnement étant constitué pour moitié de PM^2, et PM^2 étant lui même spécialisé sur la partie contrôle du programme, il va de soit d'utiliser son modèle de programmation (basé, rappelons-le, sur le principe de services exécutés par des appels légers distants) comme structure du code exécutable. La différence principale entre la version purement PM^2 et la version actuelle est apportée par Dosmos qui, par ses capacités de gestion d'objets partagés, simplifie grandement les choses. Nous nous souvenons en effet que dans notre première implémentation, il était nécessaire d'empaqueter et dépaqueter les zones du tableau à trier pour les passer en paramètres au service de tri. Plus besoin de cela désormais : le tableau à trier est maintenant un objet Dosmos accessible par tous les processus comme un tableau classique. Le service de tri ne nécessite donc plus que les bornes inférieure et supérieure de la zone à trier en paramètres, comme dans l'implémentation séquentielle de l'algorithme. Nous voyons là le tour de force réussi par notre nouvel environnement : non seulement les données partagées sont accessibles aussi simplement que des données classiques, mais surtout, l'algorithme implémenté en parallèle s'exprime comme s'il était programmé séquentiellement !

Quant-aux performances atteintes, s'il faut se garder d'espérer qu'elles soient très élevées, elles n'en réservent pas moins une surprise non négligeable ...

4.2 Des performances surprenantes

4.2.1 Comportement sur différentes architectures

Voici pour commencer (voir tableau 4.1) les temps d'exécution obtenus sur plusieurs architectures aux caractéristiques différentes, pour des tableaux de 1024 éléments avec un découpage en objets systèmes de 16 éléments. Il apparaît clairement que non seulement une seule machine puissante et dotée d'une grande quantité de mémoire vive (Serveur UltraSparc) offre des performances

Structure utilisée	Temps moyen (s)	Erreur moyenne (s)
5 stations Sun Sparc 4/5	114.2	3.0
2 stations Sun Sparc 4	87.6	2.2
2 serveurs Sun UltraSparc	27.3	1.9
1 serveur Sun UltraSparc	27.5	1.2

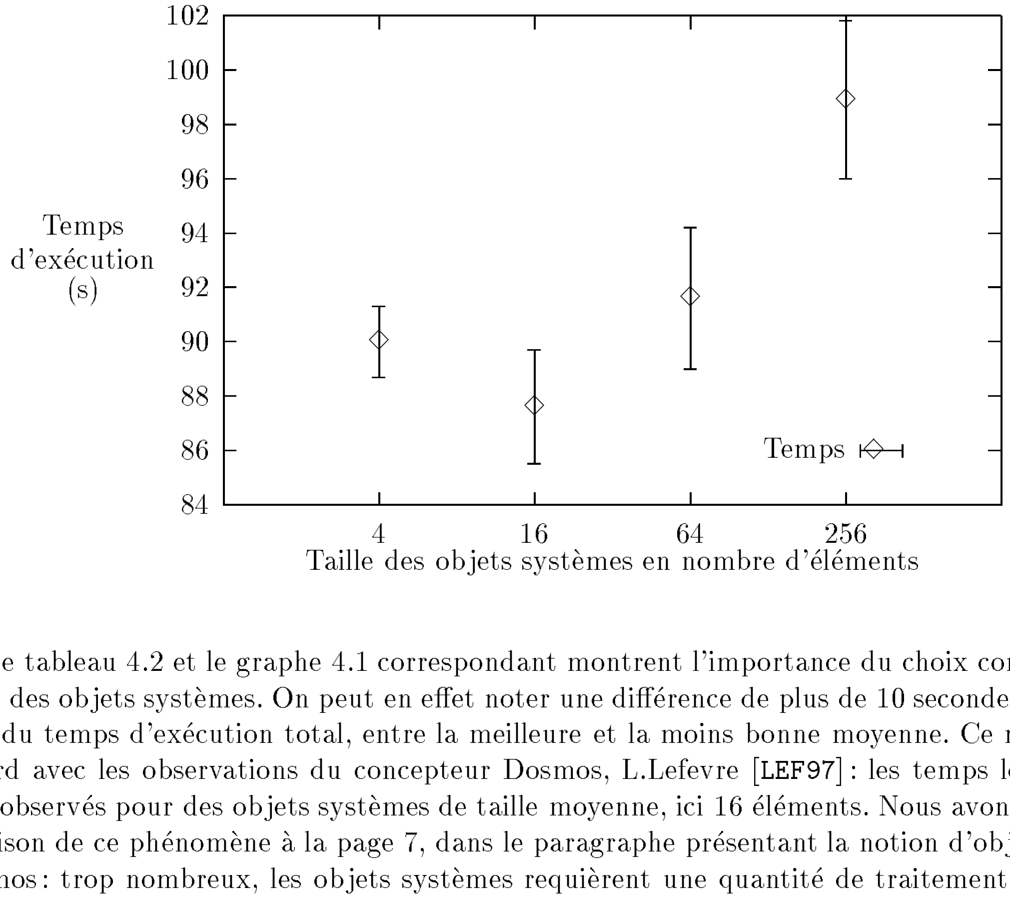
TAB. 4.1 - *Temps moyens d'exécution sur différentes architectures.*

bien meilleures que plusieurs machines de moyenne puissance et possédant moins de mémoire (Stations à base de processeurs Sparc 4 et Sparc 5), mais qu'en outre, un nombre élevé de machines constitue plutôt un handicap. Ces observations mettent en évidence un problème d'efficacité au niveau des communications.

Remarquons au passage qu'un temps moyen de 27 secondes pour trier un tableau de 1024 éléments avec *deux serveurs UltraSparc*, ca ne donne pas un rendement formidable ! A titre de comparaison, une implémentation de l'algorithme du tri rapide (Quick Sort) sur la librairie Marcel trie le même tableau en moins de 40 millisecondes sur mon propre PC (Pentium 133), c'est-à-dire avec un rapport de *700 à 1* !

4.2.2 Incidence de la taille des objets systèmes

FIG. 4.1 - *Objets systèmes et performances.*



Le tableau 4.2 et le graphe 4.1 correspondant montrent l'importance du choix convenable de la taille des objets systèmes. On peut en effet noter une différence de plus de 10 secondes, soit environ 10% du temps d'exécution total, entre la meilleure et la moins bonne moyenne. Ce résultat est en accord avec les observations du concepteur Dosmos, L.Lefevre [LEF97] : les temps les plus courts sont observés pour des objets systèmes de taille moyenne, ici 16 éléments. Nous avons d'ailleurs vu la raison de ce phénomène à la page 7, dans le paragraphe présentant la notion d'objet système de Dosmos : trop nombreux, les objets systèmes requièrent une quantité de traitement qui peut-être

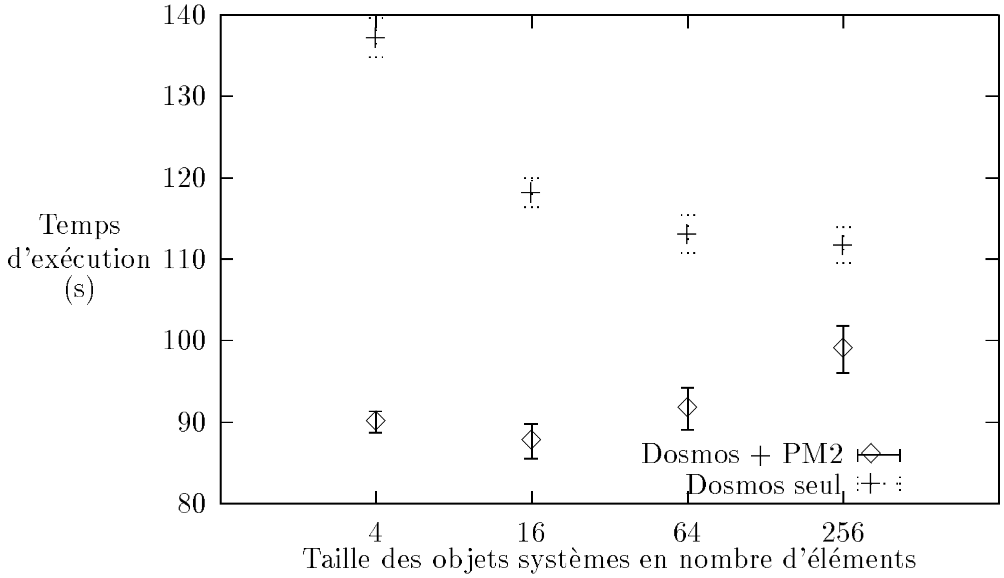
Taille des objets systèmes	Temps moyen (s)	Erreur moyenne (s)
4 éléments	90.0	1.3
16 éléments	87.6	2.1
64 éléments	91.6	2.6
256 éléments	98.9	2.9

TAB. 4.2 - *Temps moyen d'exécution en fonction de la taille des objets systèmes.*

préjudiciable aux performances, et trop grands, ils provoquent des communications plus grandes tout en limitant la portée de leur action, ce qui entraîne là aussi des temps d'exécution plus longs.

4.2.3 Comparaison avec Dosmos

FIG. 4.2 - *Temps d'exécution comparés Dosmos seul/Dosmos+Pm2.*



Taille des objets systèmes	Temps moyen (s)	Erreur moyenne (s)
4 éléments	137.2	2.4
16 éléments	118.2	1.8
64 éléments	113.1	2.3
256 éléments	111.7	2.2

TAB. 4.3 - *Temps moyens d'exécution du tri fusion sous Dosmos.*

Pour terminer, comparons les temps du tableaux 4.2 avec ceux obtenus à partir de notre implémentation purement Dosmos du tri fusion et présentés dans le tableau 4.3. Le graphe 4.2 illustre la comparaison entre les deux séries de mesures. Une remarque s'impose d'elle-même : les temps d'exécutions obtenus avec Dosmos sont supérieurs d'environ 30 secondes en moyenne à ceux obtenus avec notre nouvel environnement, soit *30% plus longs* ! Ce résultat peut sembler d'autant plus paradoxal que nous n'avons apporté aucune amélioration ou optimisation au code Dosmos, bien au contraire (nous avons par exemple substitué des boucles d'attente active aux primitives de réceptions bloquantes, ce qui est, par définition, moins efficace !).

Ce comportement est cependant logique (c'est plutôt rassurant !). Il est la conséquence de deux apports de PM^2 (plus particulièrement de Marcel) à Dosmos : le multithreading d'une part, et la préemption liée à celui-ci d'autre part. Le multithreading améliore les performances de Dosmos car il permet l'exécution simultanée, par deux threads différentes, d'une commande d'émission et d'une commande de réception, ainsi que le recouvrement des phases d'attente de messages par du calcul. En ce qui concerne la préemption, l'explication de l'amélioration des performances qu'elle induit est moins évidente. Elle est donnée par C.Perez, du LIP, qui a lui-même observé le phénomène au cours de ses propres recherches et qui rapporte que le simple fait d'inclure la librairie PM^2 dans un programme — sans faire usage des LRPC — améliore ses performances (!) : lorsqu'il est présent dans un programme PM^2, prend à sa charge la gestion des signaux, et ceci de manière plus efficace que le gestionnaire par défaut, ce qui se traduit par de meilleurs temps d'exécution.

Donc, malgré la médiocrité, somme toute attendue, des performances réalisées par notre environnement, nous terminons tout-de-même sur la satisfaction d'avoir assez sensiblement amélioré celles de Dosmos.

Chapitre 5

Conclusion

Au cours de ce stage, nous avons dans un premier temps analysé deux environnements de programmation parallèle, Dosmos et PM^2, et étudié leur modèle de programmation. Cette étude a mis en évidence leur complémentarité de-même que la nécessité d'une mise en commun de leurs caractéristiques propres dans le but de l'obtention d'un modèle de programmation complet, c'est-à-dire axé à la fois sur les *données* et sur le *contrôle*. Cette constatation nous a conduit à la réalisation d'un nouvel environnement de programmation parallèle, fruit de la fusion entre Dosmos et PM^2, et réunissant de ce fait les points forts propres à chacun d'eux : les données pour Dosmos, et le contrôle pour PM^2. Enfin, la mise en œuvre de cet environnement hybride a démontré sa viabilité et son avancée non négligeable dans le sens de la simplification de la programmation parallèle, grâce à un modèle de programmation plus proche de la programmation séquentielle, mais aussi, hélas, mis en relief un manque de performances flagrant, limitant considérablement son intérêt pratique.

Toutefois, ne perdons pas de vue qu'il s'agit là d'un simple prototype, avec toutes les imperfections que l'on peut en attendre. D'autant plus qu'au cours de la réalisation de celui-ci, nous avons eu à plusieurs reprises des choix déterminants à effectuer, et ceux-ci se sont généralement portés sur la solution la plus simple plutôt que la plus optimale. Et il ne faut pas oublier de noter, de surcroît, que nous avons dès le départ agis en vue d'adapter Dosmos à PM^2. Or, la démarche inverse, c'est-à-dire l'adaptation de PM^2 à Dosmos, eut été tout aussi envisageable, et peut-être plus intéressante. Enfin, nous aurions pu partir d'un seul environnement (voire d'une simple feuille blanche) et développer les fonctionnalités manquantes. Cette dernière voie, que nous ne pouvions suivre faute de temps, est cependant susceptible d'être celle empruntée dans le cadre de futurs projets plus orientés vers la réalisation d'environnements destinés au public, car elle permettrait une meilleure, et donc plus efficace, intégration des différents services.

Quoi qu'il en soit, la tâche est donc loin d'être terminée. Il nous reste maintenant à transformer l'essai, à passer du prototype à l'objet (ou plutôt l'environnement) de consommation courante. Pour cela, nous devrons de toute évidence concentrer l'essentiel de nos efforts sur l'amélioration des performances, qui constituent vraisemblablement le principal obstacle potentiel à la large diffusion de ce nouveau type d'environnements, sans pour autant cesser notre progression dans le sens d'un modèle de programmation simple et accessible au non-spécialiste. Les évolutions récentes de PM^2 et Dosmos devraient considérablement nous aider dans cette entreprise audacieuse, dans le monde de l'informatique du parallélisme, de réunification, au sein d'un seul et même paradigme de programmation parallèle, des deux piliers de l'informatique classique : le *contrôle* et les *données*.

Annexes

Annexe A

Tri fusion sous PM^2

A.1 Déclaration du service de tri

```
/* rpc_defs.h : tri fusion */

#ifndef __RPC_DEFS_H__
#define __RPC_DEFS_H__

#include <pm2.h>

BEGIN_LRPC_LIST
        LRPC_TRI_FUSION
END_LRPC_LIST

/* tab = tableau à trier
   n   = taille du tableau */
LRPC_DECL_REQ(LRPC_TRI_FUSION, double *tab; int n;) ;

LRPC_DECL_RES(LRPC_TRI_FUSION, double *tab; int n;) ;
#endif
```

A.2 Implémentation du service de tri

```
/* s_tri_fusion.c */

#include "rpc_defs.h"

BEGIN_SERVICE (LRPC_TRI_FUSION)
{
  if (req.n > 1) /* Le tableau a plus d'un élément -> on le partage */
    {
      /* structures de données pour l'appel récursif */
      LRPC_REQ(LRPC_TRI_FUSION) req_1, req_2 ;
      LRPC_RES(LRPC_TRI_FUSION) res_1, res_2 ;
```

```
pm2_rpc_wait att[2] ;

int milieu ; /* milieu du tableau */

milieu = req.n / 2 ;

req_1.n = milieu ;
req_1.tab = req.tab ;

req_2.n = req.n - milieu ;
req_2.tab = req.tab + milieu ; /* adresse de l'élément du début du
second sous tableau */

/* Tri Fusion sur la première moitié du tableau */
LRP_CALL(pm2_self(),
          LRPC_TRI_FUSION,
          STD_PRIO,
          DEFAULT_STACK,
          &req_1,
          &res_1,
          &att[0]
         ) ;

/* Tri Fusion sur la seconde moitié du tableau */
LRP_CALL(pm2_self(),
          LRPC_TRI_FUSION,
          STD_PRIO,
          DEFAULT_STACK,
          &req_2,
          &res_2,
          &att[1]
         ) ;

LRP_WAIT(&att[0]) ; /* attente de la fin du premier tri */
LRP_WAIT(&att[1]) ; /* attente de la fin du second tri */

/* Fusion */
{
  int i, i1, i2 ;

  i = i1 = i2 = 0 ;
  while (i < req.n)
    {
      if (i1 < res_1.n)
        {
          if ((i2 < res_2.n) && (res_2.tab[i2] < res_1.tab[i1]))
            {
              req.tab[i] = res_2.tab[i2] ;
```

```
                i ++ ;
                i2 ++ ;
              }
            else
              {
                req.tab[i] = res_1.tab[i1] ;
                i ++ ;
                i1 ++ ;
              }
          }
        else
          {
            req.tab[i] = res_2.tab[i2] ;
            i ++ ;
            i2 ++ ;
          }
      }
    }

    /* Libération des ressources allouées lors de la réception
       des données */
    tfree(res_1.tab) ;
    tfree(res_2.tab) ;

    res.tab = req.tab ;
    res.n = res_1.n + res_2.n ; /* uniquement à titre de vérification */
  }
  else
  {
    /* Sinon, le tableau a un seul élément -> on le retourne tel quel */

    /* Transfert des données dans la structure de retour */
    res.n = req.n ;
    res.tab = req.tab ;
  }
}
END_SERVICE (LRPC_TRI_FUSION)

main()
{
  pm2_init_rpc();

  DECLARE_LRPC_WITH_NAME(LRPC_TRI_FUSION, "Tri_Fusion", DO_NOT_OPTIMIZE);

  pm2_init() ;
  pm2_exit() ;
}
```

A.3 Empaquetage/dépaquetage

```
/* rpc_defs.c : tri fusion */

#include "rpc_defs.h"

/* arg->tab doit contenir un pointeur sur la première case de la
   partie du tableau à trier, et n doit contenir le nombre de cases
   du tableau à trier. */
PACK_REQ_STUB(LRPC_TRI_FUSION)
     pm2_pkint(&arg->n, 1, 1);
     pm2_pkdouble(arg->tab, arg->n, 1);
END_STUB

/* décompactage à l'arrivée. tmalloc alloue la mémoire nécessaire
   au tableau, dynamiquement. La libération de la mémoire se fait lors
   du retour du tableau trié, dans PACK_RES_STUB.
 */
UNPACK_REQ_STUB(LRPC_TRI_FUSION)
     pm2_upkint(&arg->n, 1, 1);
     arg->tab = (int *)tmalloc(arg->n * sizeof (double));
     if (arg->tab == NULL)
     {
        pvm_exit() ;
        exit (1) ;
     }
     pm2_upkdouble(arg->tab, arg->n, 1) ;
END_STUB

PACK_RES_STUB(LRPC_TRI_FUSION)
     pm2_pkint(&arg->n, 1, 1);
     pm2_pkdouble(arg->tab, arg->n, 1);
     tfree(arg->tab) ;
END_STUB

/* ** Attention ** - Ne pas oublier de liberer la memoire allouee ici */
UNPACK_RES_STUB(LRPC_TRI_FUSION)
     pm2_upkint(&arg->n, 1, 1);
     arg->tab = (int *)tmalloc(arg->n * sizeof (double));
     if (arg->tab == NULL)
     {
        pvm_exit() ;
         exit (1) ;
     }
     pm2_upkdouble(arg->tab, arg->n, 1) ;
END_STUB

/* m_tri_fusion.c */
```

```
#include <sys/time.h>
#include "ultra.h"
#include "rpc_defs.h"

void tri(double *tab, int n)
{
  int slave_id ;
  LRPC_REQ(LRPC_TRI_FUSION) req ;
  LRPC_RES(LRPC_TRI_FUSION) res ;

  /* Lancement du module */
  {
    int resultat ;

    resultat = pm2_start_modules("s_tri_fusion",
                                 NULL,
                                 PvmTaskDefault,
                                 "",
                                 1,
                                 &slave_id
                                 ) ;

    /* verification du lancement correct du module */
    if (resultat != 1)
      {
        tfprintf(stderr, "Spawn impossible\n") ;
        pvm_exit();
        exit(1) ;
      }
  }

  req.n = n ;
  req.tab = tab ;

  LRPC (slave_id,
          LRPC_TRI_FUSION,
          STD_PRIO,
          DEFAULT_STACK,
          &req,
          &res
         ) ;

  /* Affichage du resultat */
  {
    int i ;

    tfprintf (stderr, "n = %d, res.tab = %p\n", res.n, res.tab) ;
    for (i = 0 ; i < n ; i ++)
```

```
      {
        tfprintf(stderr, "%lf\n", res.tab[i]) ;
      }
  }

  /* Liberation du tableau resultat */
  tfree(res.tab) ;

  {
    int self ;

    self = pm2_self() ;
    pm2_kill_modules (&slave_id, 1) ;
    pm2_kill_modules (&self, 1) ;
  }
}

main()
{

  double *tableau ;
  int n ;

  printf ("n = ") ;
  scanf ("%d", &n) ;

  rinit (time(0), time(0));

  {
    int i ;

    tableau = (double *)malloc (n * sizeof (double)) ;
    if (tableau == NULL)
      {
        exit (1) ;
      }

    for (i = 0 ; i < n ; i++)
      {
        tableau [i] = duni () ; /* commande FS-Ultra  */
      }
  }

  pm2_init_rpc();
  pm2_init();


  tri (tableau, n) ;
```

```
  pm2_exit() ;
}
```

Annexe B

Tri fusion sous Dosmos

B.1 Implémentation

```
/* Tri fusion en Dosmos */

#include <Dosmos.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "ultra.h"

#define TAILLE 1024
#define TAILLE_SYSTEME 256

#define MIN(a, b) ((a)<(b))?(a):(b)

int taille_tableau = TAILLE ;
int taille_objet_systeme = TAILLE_SYSTEME ;
shared double tableau[TAILLE](TAILLE_SYSTEME) ;
shared double tableau_2[TAILLE](TAILLE_SYSTEME) ;/* nécessaire pour
                                                     les fusions */
static struct timeval _tv;
static struct timezone _tz;

void init_time()
{
  gettimeofday(&_tv, &_tz);
}

unsigned long cpu_time ()
{
  struct timeval t;

  gettimeofday(&t, &_tz);
  return (1000000L * (t.tv_sec - _tv.tv_sec)
```

```
          + t.tv_usec - _tv.tv_usec) / 1000;
}

void fusion (int debut_1, int fin_1,
             int debut_2, int fin_2,
             int tab_dest)
{
  int i1 ;
  int i2 ;
  int i ;

  i = i1 = debut_1 ;
  i2 = debut_2 ;

  if (tab_dest == 0)
    {
      while (i < fin_2)
        {
          if (i1 < fin_1)
            {
              if ((i2 < fin_2) && (tableau_2[i2] < tableau_2[i1]))
                {
                  tableau[i] = tableau_2[i2] ;
                  i++ ;
                  i2++ ;
                }
              else
                {
                  tableau[i] = tableau_2[i1] ;
                  i++ ;
                  i1++ ;
                }
            }
          else
            {
              tableau[i] = tableau_2[i2] ;
              i++ ;
              i2++;
            }
        }
    }
  else
    {
      while (i < fin_2)
        {
          if (i1 < fin_1)
            {
              if ((i2 < fin_2) && (tableau[i2] < tableau[i1]))
```

```
              {
                tableau_2[i] = tableau[i2] ;
                i++ ;
                i2++ ;
              }
            else
              {
                tableau_2[i] = tableau[i1] ;
                i++ ;
                i1++ ;
              }
          }
        else
          {
            tableau_2[i] = tableau[i2] ;
            i++ ;
            i2++;
          }
      }
    }

}

void tri(int debut, int fin, int tab_dest, int principal_tab_dest)
{
  if ((fin - debut) <= 1)
    {
      if (tab_dest == principal_tab_dest)
        {
          if (tab_dest == 0)
            {
              tableau[debut] = tableau_2[debut] ;
            }
          else
            {
              tableau_2[debut] = tableau[debut] ;
            }
        }
    }
  else
    {
      int milieu ;

      milieu = (fin - debut) / 2 + debut ;

      tri (debut, milieu, 1 - tab_dest, principal_tab_dest);
      tri (milieu, fin, 1 - tab_dest, principal_tab_dest);
      fusion (debut, milieu,
```

```
              milieu, fin,
              tab_dest);
    }
}

#include <Dosmos_main.h>
{
  int i;
  int saut ;
  /*  unsigned long temps_0 ;
  unsigned long temps ;*/
  unsigned long temps_2 ;

  /* Initialisation de FSUltra */
  rinit (time(0), time(0));

  init_time();

  dm_use_dosmos();

  printf ("==================================================\n");
  printf ("Taille tableau : %d\n", taille_tableau) ;
  printf ("Taille objet systeme : %d\n", taille_objet_systeme) ;
  printf ("Nombre de processus : %d\n", dm_DN_max) ;
  printf ("==================================================\n");

  saut = dm_DN_max * taille_objet_systeme ;

  {
    int debut ;

    debut = (dm_DN - 1) * taille_objet_systeme ;
    while (debut < taille_tableau)
      {
        /* permet d'assigner un objet systeme au processus correspondant */
        dm_acquire(tableau[debut]);
        dm_acquire(tableau_2[debut]);
        dm_release(tableau_2[debut]);
        dm_release(tableau[debut]);

        debut += saut ;
      }
  }

  dm_sync(tableau);

  if (dm_DN == 1)
    {
```

```
    for (i = 0 ; i < taille_tableau ; i++)
      {
        tableau[i] = duni() ; /* commande FSUltra */
      }
  }


/* Barrière de synchronisation pre_tri */
dm_sync(tableau);

/* Tri des sous_tableaux 'systèmes' */
{
  int debut ;

  debut = (dm_DN - 1) * taille_objet_systeme ;
  while (debut < taille_tableau)
    {
      /* Tri fusion sur les sous-tableaux */
      int fin ;

      fin = MIN((debut + taille_objet_systeme), (taille_tableau)) ;
      tri (debut,
           fin,
           1,
           1
           );
      debut += saut ;
    }
}

/* Barrière de synchronisation post_tri */
dm_sync(tableau);
/* Fusion terminale */
{
  int tab_dest = 0 ;
  int saut_temp ;

  saut_temp = taille_objet_systeme ;

  while (saut_temp < taille_tableau)
    {
      int debut_1 ;
      int fin_1 ;
      int debut_2 ;
      int fin_2 ;

      /*      temps_0 = cpu_time();*/
```

```
      debut_1 = (dm_DN - 1) * saut_temp * 2 ;

      while (debut_1 < taille_tableau)
      {
        debut_2 = fin_1 = debut_1 + saut_temp ;

        if (debut_2 >= taille_tableau)
          {
            debut_2 = taille_tableau ; /* force la recopie du tableau */
          }

        fin_2 = MIN (debut_2 + saut_temp, taille_tableau) ;

        fusion (debut_1, fin_1,
                debut_2, fin_2,
                tab_dest) ;

        debut_1 = debut_1 + (saut_temp * 2 * dm_DN_max) ;
      }

      tab_dest = 1 - tab_dest ;
      saut_temp *= 2 ;

      dm_sync(tableau);
    }


   if (tab_dest == 0)
    {
      int debut ;

      debut = (dm_DN - 1) * taille_objet_systeme ;
      while (debut < taille_tableau)
        {
          int j ;
          int fin ;

          fin =  MIN((debut + taille_objet_systeme), taille_tableau) ;
          for (j = debut ; j < fin ; j++)
            {
              tableau[j] = tableau_2[j] ;
            }

          debut += saut ;
        }
    }
}
```

```
  /* synchronisation avant l'affichage */
  dm_sync(tableau);
  temps_2 = cpu_time();
  printf ("%2d - Fin : temps : %3d.%03ds\n",
          dm_DN, temps_2/1000,
          temps_2%1000) ;

  dm_end_dosmos(1) ;

  return 0 ;
}
```

Annexe C

Tri fusion sous Dosmos + PM^2

C.1 Implémentation

```
/* tri_fusion.dc */
#include <Dosmos.h>
#include <sys/time.h>
#include "ultra.h"
#include "pthread.h"
#include "pthread_sem.h"
#include "pthread_mutex.h"
#include "pm2.h"

#define TAILLE_TABLEAU 1024

static int taille_tableau = TAILLE_TABLEAU ;
shared double tableau_1[TAILLE_TABLEAU](4);
shared double tableau_2[TAILLE_TABLEAU](4);
static pthread_mutexattr_t mutex_attr ;
static pthread_mutex_t mutex ;
static volatile int processus = 0;
shared long tids[16];
static struct timeval _tv;
static struct timezone _tz;
static unsigned long top_1 ;
static unsigned long top_2 ;

void init_time()
{
  gettimeofday(&_tv, &_tz);
}


unsigned long cpu_time ()
{
  struct timeval t;
```

```
  gettimeofday(&t, &_tz);
  return (1000000L * (t.tv_sec - _tv.tv_sec)
          + t.tv_usec - _tv.tv_usec) / 1000;
}


/* liste des services LRPC */
BEGIN_LRPC_LIST
        LRPC_TRI_FUSION
END_LRPC_LIST

/* Declaration des services */
LRPC_DECL_REQ(LRPC_TRI_FUSION, int deb; int fin; int dest;) /* ne pas mettre de
                                                               point virgule */

LRPC_DECL_RES(LRPC_TRI_FUSION, int dummy;)

/* implementation des souches */
PACK_REQ_STUB(LRPC_TRI_FUSION)
     lock_task();
     pm2_pkint(&arg->deb, 1, 1);
     pm2_pkint(&arg->fin, 1, 1);
     pm2_pkint(&arg->dest, 1, 1);
     unlock_task();
END_STUB

UNPACK_REQ_STUB(LRPC_TRI_FUSION)
  pm2_upkint(&arg->deb, 1, 1);
  pm2_upkint(&arg->fin, 1, 1);
  pm2_upkint(&arg->dest, 1, 1);
END_STUB

PACK_RES_STUB(LRPC_TRI_FUSION)
END_STUB

UNPACK_RES_STUB(LRPC_TRI_FUSION)
END_STUB


void fusion (int debut_1, int fin_1,
             int debut_2, int fin_2,
             int tab_dest)
{
  int i1 ;
  int i2 ;
  int i ;
```

```
i = i1 = debut_1 ;
i2 = debut_2 ;

if (tab_dest == 0)
  {
    /*      dm_acquire (tableau_1);*/
    while (i < fin_2)
      {
        if (i1 < fin_1)
          {
            if ((i2 < fin_2) && (tableau_2[i2] < tableau_2[i1]))
              {
                tableau_1[i] = tableau_2[i2];
                i++ ;
                i2++ ;
              }
            else
              {
                tableau_1[i] = tableau_2[i1];
                i++ ;
                i1++ ;
              }
          }
        else
          {
            tableau_1[i] = tableau_2[i2];
            i++ ;
            i2++;
          }
      }
    /*      dm_release (tableau_1);*/
  }
else
  {
    /*      dm_acquire (tableau_2);*/
    while (i < fin_2)
      {
        if (i1 < fin_1)
          {
            if ((i2 < fin_2) && (tableau_1[i2] < tableau_1[i1]))
              {
                tableau_2[i] = tableau_1[i2];
                i++ ;
                i2++ ;
              }
            else
              {
                tableau_2[i] = tableau_1[i1];
```

```
                i++ ;
                i1++ ;
              }
          }
        else
          {
            tableau_2[i] = tableau_1[i2];
            i++ ;
            i2++;
          }
      }
    /*      dm_release (tableau_2);*/
  }
}

/* implementation des services */
BEGIN_SERVICE (LRPC_TRI_FUSION)
{
  if ((req.fin - req.deb) > 1) /* Le tableau a plus d'un élément
                                    -> on le partage */
    {
      /* structures de données pour l'appel récursif */
      LRPC_REQ(LRPC_TRI_FUSION) req_1, req_2 ;
      LRPC_RES(LRPC_TRI_FUSION) res_1, res_2 ;

      pm2_rpc_wait att[2] ;

      int milieu ; /* milieu du tableau */

      milieu = req.deb + (req.fin - req.deb) / 2 ;

      req_1.deb = req.deb ;
      req_1.fin = milieu ;

      req_1.dest = req_2.dest = 1 - req.dest ;

      req_2.deb = milieu ;
      req_2.fin = req.fin ;


      {
        unsigned long lrpc_tid;
        int proc ;

        lock_task();

        proc = processus = (processus + 1) % dm_DN_max ;
        unlock_task();
```

```
        lrpc_tid = tids[proc];

        /* Tri Fusion sur la premiere moitie du tableau */
        LRP_CALL(lrpc_tid,
                 LRPC_TRI_FUSION,
                 STD_PRIO,
                 DEFAULT_STACK,
                 &req_1,
                 &res_1,
                 &att[0]
                 ) ;

        /* Tri Fusion sur la seconde moitie du tableau */
        LRP_CALL(pm2_self(),
                 LRPC_TRI_FUSION,
                 STD_PRIO,
                 DEFAULT_STACK,
                 &req_2,
                 &res_2,
                 &att[1]
                 ) ;
      }

      LRP_WAIT(&att[0]) ; /* attente de la fin du premier tri */
      LRP_WAIT(&att[1]) ; /* attente de la fin du second tri */

      fusion (req.deb, milieu, milieu, req.fin, req.dest) ;

      /* Liberation des ressources allouees lors de la réception
         des données */
    }
  else
    {
      /* Sinon, le tableau a un seul élément -> on le retourne tel quel */

      if (req.dest == 1)
        {
          /*      dm_acquire (tableau_2);*/
          tableau_2[req.deb] = tableau_1[req.deb];
          /*      dm_release (tableau_2);*/
        }
    }
}
END_SERVICE (LRPC_TRI_FUSION)

/* programme maitre */
void tri(void)
{
```

```
  LRPC_REQ(LRPC_TRI_FUSION) req ;
  LRPC_RES(LRPC_TRI_FUSION) res ;

  req.deb = 0 ;
  req.fin = taille_tableau ;
  req.dest = 1 ;

  LRPC (pm2_self(),
        LRPC_TRI_FUSION,
        STD_PRIO,
        DEFAULT_STACK,
        &req,
        &res
        ) ;

  {
    int i ;

    for (i = 0 ; i < taille_tableau ; i ++)
      {
        tfprintf(stderr, "%lf\n", tableau_2[i]) ;
      }
  }


}

#include <Dosmos_main.h>
{
  int proc ;

  pm2_init_rpc();
  DECLARE_LRPC_WITH_NAME(LRPC_TRI_FUSION, "Tri_Fusion", OPTIMIZE_IF_LOCAL);
  pm2_init();
  dm_use_dosmos() ;

  init_time();
  rinit (time(0), time(0));
  pthread_mutexattr_init (&mutex_attr);
  pthread_mutex_init (&mutex, &mutex_attr);

  proc = dm_DN ;

  dm_sync (tids);
  tids[dm_DN - 1] = dm_DN2tid(dm_DN);
  dm_sync (tids);

  if (dm_DN == 1)
```

```
  {
    int i ;


    for (i = 0 ; i < taille_tableau ; i++)
      {
        tableau_1[i] = duni () ; /* commande FS-Ultra  */
      }
  }

  processus = (dm_DN + 1) % dm_DN_max ;
  dm_sync(tableau_1);
  top_1 = cpu_time () ;


  if (dm_DN == 1)
    {
      tri () ;
    }


  dm_sync(tableau_1);

  top_2 = cpu_time();
  tfprintf (stderr, "Duree : %3d.%03ds\n",
          (top_2 - top_1) / 1000,
          (top_2 - top_1) % 1000);

  dm_end_dosmos (0);
  {
    int self ;

    self = pm2_self() ;
    pm2_kill_modules (&self, 1) ;
  }
  pm2_exit() ;
}
```

Bibliographie

[LEF97] L. Lefevre. Thèse : *Conception et Mise en Œuvre d'un Environnement de Programmation Parallèle fondé sur un Système de Mémoire Distribuée Virtuellement Partagée*. Ecole Normale Supérieure de Lyon, 1997.

[NAM97] R. Namyst. Thèse : *PM^2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. Université des Sciences et Technologies de Lille, 1997.