

Habilitation à diriger les recherches

présentée par

Jean-François Méhaut

Maître de conférences de l'Université de Lille 1, détaché à l'INRIA Rhône-Alpes

Supports d'exécution pour le calcul parallèle et distribué

le **28 janvier 2000**

Après avis de : IAN FOSTER
 SACHA KRAKOWIAK
 CATHERINE ROUCAIROL

Devant la commission d'examen formée de :

LUC BOUGE
IAN FOSTER
JEAN-MARC GEIB
SACHA KRAKOWIAK
THIERRY PRIOL
CATHERINE ROUCAIROL
BERNARD TOURANCHEAU

**pour des travaux effectués à l'Université de Lille 1
et au Laboratoire de l'Informatique du Parallélisme
de l'École Normale Supérieure de Lyon.**

Table des matières

1	Introduction	5
1.1	Organisation du document	6
1.2	Remerciements	7
2	Pourquoi des supports exécutifs parallèles ?	9
2.1	Nouvelles architectures pour le calcul parallèle	10
2.1.1	Grilles de calcul	10
2.1.2	Grappes de stations	13
2.1.3	Quelles nouvelles problématiques logicielles ?	15
2.2	Systèmes d'exploitation et calcul parallèle	17
2.2.1	Multithreading	19
2.2.2	Communications dans les systèmes	22
2.3	Qu'est-ce qu'un support d'exécution ?	24
2.4	Modèles de programmation parallèle	26
2.5	Quelques problèmes difficiles	28
2.5.1	Déverminage de programmes parallèles	29
2.5.2	Régulation de charge	30
2.5.3	Recouvrement des communications	32
2.6	Positionnement de notre contribution	33
3	Supports d'exécution multithreads	35
3.1	Deux projets de recherche	36
3.1.1	PVC-BOX : 1990-1995	36
3.1.2	ESPACE : 1995-1998	37
3.2	Modèles d'exécution	38
3.2.1	Composants Actifs de Communication	39
3.2.2	Appel de procédure à distance léger	40
3.2.3	Migration de processus légers	41
3.3	Interfaces de programmation	42
3.3.1	PVC	42
3.3.2	PM ²	44

3.4	Réalisations des supports exécutifs	46
3.4.1	PVC	47
3.4.2	PM ²	47
3.5	Utilisation des supports PVC et PM ²	52
3.5.1	PM ² comme environnement applicatif	52
3.5.2	PVC et PM ² comme cibles de compilation	55
3.5.3	PM ² comme support de régulateurs de charge	58
3.6	Bilan des projets	61
4	Conclusions et perspectives	63
A	Exemple de programme PVC	65
A.1	L'application somme	65
A.2	Le module <i>Module_Dia</i>	65
A.3	Le module <i>Module_Somme</i>	67
B	Exemple de programme PM²	69
B.1	Principe d'implémentation de l'application somme avec PM ²	69
B.2	L'application somme avec PM ²	69

Chapitre 1

Introduction

Le contenu de ce mémoire décrit les activités de recherche que j'ai menées après l'obtention de mon doctorat en 1989. Ces activités se sont principalement déroulées dans le cadre du Laboratoire d'Informatique Fondamentale de Lille où j'ai occupé un poste de Maître de Conférences. Pendant ces années, j'ai participé à la direction et à l'encadrement de deux projets de recherche : le projet PVC-BOX, de 1990 à 1995 et le projet ESPACE de 1995 à 1998¹. Nous décrirons dans ce document les motivations qui nous ont conduit à lancer ces projets, nous en dresserons un bilan complet et critique qui permettra d'établir des perspectives à nos recherches.

J'ai préparé une thèse au sein du projet Omphale qui avait pour objectif la conception et la réalisation d'un système d'exploitation réparti et modulaire. Cette structuration modulaire, basée sur un modèle à objets, était une des spécificités d'Omphale et constituait un thème novateur à cette époque. Un mécanisme original et sophistiqué de nommage et de protection basé sur des capacités avait en particulier été proposé. J'avais plus précisément contribué à définir l'architecture et à développer le micro-noyau de ce système sur des architectures multiprocesseurs de type SM90 [1]. Je ne décrirai pas dans ce document l'architecture du système Omphale², mais j'insisterai plutôt sur les motivations qui nous ont incités à nous investir au niveau des supports d'exécution. C'est aussi à cette époque que des systèmes comme Chorus [117] ou bien encore Mach [109] s'imposaient au niveau industriel et il était difficile, avec les moyens d'une petite équipe universitaire, de poursuivre à plus long terme dans cette voie. La fin des années 80 a vu débiter les premières activités de recherche autour des supports exécutifs [123, 61] et c'est dans ce contexte que notre équipe a réorienté ses travaux de recherche sur des modèles et langages à objets. Cette réorientation a donc coïncidé en 1990 avec la constitution d'une nouvelle équipe de recherche (GOAL : Groupe Objets et Acteurs de Lille) comprenant un professeur (Éric Delattre) et trois maîtres de conférences (Bernard Carré, Jean-Marc Geib et Jean-François Méhaut). Les travaux de l'équipe se sont donc principalement orientés autour du nouveau projet PVC-BOX dont l'ambition était de promouvoir les technologies à objets pour l'exploitation des architectures parallèles et distribuées. Ces travaux se situent dans la même mouvance que des projets autour des acteurs [2, 127] ou bien de langages et systèmes orientés objets [82, 79]. Au sein du projet PVC-BOX, j'ai été principalement responsable des travaux autour du support exécutif PVC³ ainsi que de différents outils d'aide au développement (déverminage, ramasse-miettes, etc.).

¹Suite à mon détachement INRIA, c'est Nathalie Revol qui dirige actuellement les activités du projet ESPACE et participe à l'encadrement de la thèse de Benoit Planquelle.

²Cela a fait l'objet de plusieurs thèses [65, 90, 102] et mémoires d'habilitation [40, 66].

³Processeurs Virtuels de Classe.

En 1995, plusieurs thèses du projet PVC-BOX se sont terminées et ce fut l'occasion pour Jean-Marc Geib et moi de dresser un bilan *a posteriori* du projet PVC-BOX. Le positionnement des activités de notre équipe au niveau des supports exécutifs était pleinement justifié par les résultats et la tendance observée dans la communauté. De plus, nous étions convaincus que ce type d'environnement et les technologies avaient atteint un niveau suffisant de maturité et de performance qui les rendaient utilisables pour le développement d'applications réelles. Notre participation à l'opération Inter-PRC Stratagème [116] nous a donné l'opportunité de travailler avec des équipes de recherche plus spécialisées dans les applications et notre contribution a consisté à concevoir un support exécutif adapté aux besoins des applications irrégulières. C'est donc autour de ces conclusions sur le projet PVC et de différentes collaborations que nous avons entrepris de démarrer en 1995 le projet ESPACE⁴. Nous proposons de nous investir à la fois au niveau des supports d'exécution à base de processus légers (plate-forme PM² : *Parallel Multithreaded Machine*), mais également au niveau des aspects régulation de charge et migration (LBMP⁵) avec un souci de validation et d'expérimentation autour d'applications développées à l'extérieur de notre groupe.

1.1 Organisation du document

Le corps de ce mémoire comprend trois chapitres principaux.

Dans le premier chapitre, je décrirai les motivations et les besoins pour concevoir et réaliser des supports d'exécution. Ce chapitre débutera par une présentation des évolutions actuelles des infrastructures matérielles de calcul à haute performance. La justification majeure des environnements d'exécution provient du fait que les systèmes d'exploitation sont à la fois trop complexes pour être utilisés directement par des concepteurs d'applications parallèles, mais surtout trop rudimentaires au niveau des paradigmes de programmation qu'ils offrent. L'objectif des supports et environnements d'exécution est donc de fournir des paradigmes et des modèles plus riches et plus évolués que ceux disponibles au niveau des interfaces systèmes. Nous décrirons dans ce chapitre quelques-unes des principales contributions dans ce domaine et positionnerons précisément nos travaux.

Le second chapitre décrit principalement nos contributions au niveau des supports d'exécution à base de processus légers : PVC et PM². Les processus légers sont très vite apparus comme une entité suffisamment flexible pour supporter les activités des programmes parallèles. En outre, ils apportent une solution élégante au problème du recouvrement des communications par des calculs. Nous avons donc décidé d'utiliser cette technologie dans les environnements de programmation parallèle PVC et PM². L'objectif de PVC était de fournir la structure de base nécessaire pour le développement de compilateurs de langages à objets et c'est ainsi qu'il a été principalement utilisé par le compilateur du langage BOX. En ce qui concerne PM², nous souhaitons qu'il puisse être utilisé par une gamme plus large d'applications. Nous comparerons nos approches avec celles développées par d'autres projets (Nexus, Athapascan-0) et justifierons nos choix de conception et de réalisation. La dernière partie de ce chapitre résume les principales expériences et utilisations qui ont été réalisées à partir de nos travaux. Comme nous l'avons mentionné précédemment, la validation des environnements en grandeur réelle permet de confirmer si les choix de conception et d'implémentation sont judicieux. Nous décrirons donc les expérimentations les plus significatives, en particulier pour le développement de compilateurs (langages data-parallèles, langages à objets), d'applications irrégulières (optimisation combinatoire) et d'une application d'analyse d'images. Nous terminerons ce chapitre en dressant des perspectives à ces travaux.

⁴Execution Support for Parallel Applications in high-performance Computing Environments.

⁵Load Balancing with Migrations directed by Priorities.

La première partie des annexes comprend un curriculum-vitae, une description des diverses activités d'enseignement et responsabilités administratives que j'ai été amené à assurer, principalement à l'université de Lille 1 et plus récemment à l'École normale supérieure de Lyon. Une série de publications complètes décrivant certains aspects de nos recherches figure dans les annexes de ce document. Certaines détaillent de manière plus approfondie l'ensemble de notre contribution, tandis que d'autres illustrent quelques unes de nos collaborations les plus significatives.

1.2 Remerciements

Si ces travaux ont pu être menés et que des étudiants ont été formés à la recherche et par la recherche, c'est parce qu'un certain nombre d'institutions et de laboratoires les ont activement soutenus. Je commencerai donc par remercier chaleureusement l'université de Lille 1 et le LIFL pour les jeunes et heureuses années que j'y ai passées. J'y associerai également le LIP à l'ENS Lyon, le DSL à Argonne National Laboratory et le LMC de Grenoble qui m'ont accueilli pendant mon année de congé recherche. Ces travaux sont le fruit d'un travail d'équipe où j'ai été particulièrement fier de travailler avec plusieurs étudiants-chercheurs à qui j'espère avoir réussi à communiquer ma passion pour la recherche. Encore merci à eux d'avoir supporté ma présence dans le fameux bureau 304 du M3 et ma façon si particulière d'y ranger mes affaires ! Je citerai donc Luc Courtrai (maître es Transputer), Jean-François Roos (dévermineur de programmes) et Cédric Dumoulin (ramasseur de miettes) qui ont été les ouvriers zélés du projet PVC. J'ai pris aussi beaucoup de plaisir à animer et coordonner les activités autour de PM² avec Raymond Namyst (assembleur à haute performance), Yves Denneulin (ancien migreur) et Benoît Planquelle (swappeur).

Je voudrais aussi rendre hommage aux collègues qui m'ont encouragé à poursuivre une activité de recherche. Christian Carrez dont les cours de système en DEA m'ont passionnés et qui m'a incité à travailler dans ce domaine. J'ai aussi une pensée émue pour Éric Delattre, trop tôt disparu, qui m'avait montré ce que devait être un bon encadrant. J'ai aussi beaucoup apprécié travailler dans la célèbre équipe GOAL du LIFL animée par Jean-Marc Geib et je le remercie vivement pour l'autonomie et la confiance qu'il m'a laissées dans l'orientation et le déroulement de mes activités. Mes sincères remerciements également à Luc Bougé et Yves Robert pour ces deux années que je passe au LIP et qui nous (Raymond et moi) ont permis de continuer et intensifier les travaux autour de PM², mais aussi de découvrir un riche environnement scientifique. Merci aussi à Nathalie Revol, relectrice des thèses et habilitations de notre équipe !

Je voudrai aussi remercier la communauté du parallélisme en France avec qui les échanges scientifiques ont été très riches et fructueux pendant les différentes écoles (Capa, Icare), conférences RenPar et groupes de travail (PRS, Grappes, iHPerf, ResCapA). J'en oublierai certainement, qu'ils ne m'en veuillent pas ! Voici une liste presque complète : Jean Roman à Bordeaux, Catherine Roucairol, Van-Dat Cung, Bertrand Le Cun et Thierry Mautor à Versailles, Jacques Briat, Jean-Louis Roch et Denis Trystram à Grenoble, Françoise Baude, Nathalie Furmento et Michel Syska à Nice, Dominique Lazure à Amiens, Philippe Lalevée, Daniel Millot et Farba Sy à l'INT Évry, Pierre Manneback et Jean-Noël Colin à Mons (Belgique), Thierry Priol et Jean-Louis Pazat à Rennes, Gilles Goncalves, Olivier Caron, Tienté Hsu à Béthune, Tahar Kechadi à Dublin, Mostafa Daoudi à Oujda, Bertrand Ducourthial et Nicolas Sicard à Orsay, Alain Greiner à Paris VI, Hafid Bourzoufi et Arnaud Fréville à Valenciennes...

Je n'oublierai pas les membres de mon jury. Catherine Roucairol, professeur à Versailles, qui nous a beaucoup aidé en sollicitant notre participation à l'opération Stratagème et au groupe de travail Capa. Sacha Krakowiak, professeur à Grenoble, que j'ai d'abord connu au travers de ses livres

sur les systèmes d'exploitation et puis maintenant au sein de l'INRIA Rhône-Alpes. J'ai beaucoup apprécié les discussions avec lui sur les évolutions des recherches en système. J'ai eu aussi la chance de travailler dans l'équipe de Ian Foster et de découvrir un environnement de travail assez riche et le projet Globus, tel que seuls peut-être les Américains sont capables de mener. J'ai vraiment beaucoup appris pendant ce séjour.

Chapitre 2

Pourquoi des supports exécutifs parallèles ?

Au cours de ces quinze dernières années, les recherches et les évolutions autour des technologies de réseaux et de microprocesseurs ont fortement contribué à l'émergence de l'informatique distribuée. Les performances de calcul des microprocesseurs ne cessent de croître, alors que les coûts des recherches et développements sont rapidement amortis avec les grands volumes de vente. De nouvelles technologies de réseau (satellite, GSM, câble, etc.) ouvrent de nouvelles voies pour les communications et l'informatique. À l'échelle mondiale, le réseau Internet illustre parfaitement cette tendance, au niveau d'une entreprise ou d'une université, les environnements informatiques sont aujourd'hui principalement distribués autour de réseaux locaux (Intranet). L'informatique centralisée autour de *mainframes* propriétaires qui était prédominante jusque dans les années 1980 se trouve aujourd'hui largement dépassée par les nouvelles technologies et architectures qui sont nées autour des réseaux de communication.

Dans ce contexte d'informatique distribuée, les services de communication (*mail*, *news*) ont été les premiers à être opérationnels et deviennent aujourd'hui des concurrents sérieux pour les outils de communication plus traditionnels (téléphone, courrier postal). D'autres services électroniques se sont développés, comme par exemple les services de partage de ressources ou bien encore les services d'information qui eux aussi s'avèrent de sérieux concurrents pour les diffuseurs d'informations plus traditionnels (presse écrite, TV, etc.). Pour faciliter l'implémentation de ces nouveaux services, les systèmes d'exploitation se sont vus doter de mécanismes de communication permettant aux différentes machines de s'échanger des données sous différents formats.

Si le potentiel de communication de ces infrastructures est largement utilisé par ces nouveaux services, il n'en va pas de même pour le potentiel de calcul qui reste relativement mal exploité aujourd'hui. En effet, les processeurs et microprocesseurs des machines ne servent qu'à effectuer des "petits" traitements locaux. C'est particulièrement regrettable lorsque l'on sait que pour un banal réseau local de 30 stations de travail, la puissance de calcul cumulée dépasse souvent largement celle de machines parallèles. C'est une des raisons qui a motivé la communauté du parallélisme à s'intéresser aux architectures distribuées afin d'offrir des solutions pour le calcul parallèle à un plus grand nombre d'applications et donc d'utilisateurs.

Plus récemment, l'accent a été mis aux États-Unis sur l'utilisation des réseaux à grande échelle pour constituer des grilles de calcul regroupant des supercalculateurs situés dans les principaux laboratoires de recherche [55, 59]. C'est ce qu'on appelle plus communément le calcul distribué à

haute performance ou *métacomputing*. Dans ce contexte, le principe est de globaliser les ressources des supercalculateurs et d'offrir aux utilisateurs un support flexible s'adaptant aux besoins des applications et aux différentes contraintes de l'environnement. Les environnements Legion [86] et Globus [58] fournissent aux utilisateurs un cadre de conception et d'exécution permettant à leurs applications de s'exécuter efficacement sur des réseaux à grande échelle.

Nous décrivons dans la première section de ce chapitre deux approches architecturales pour effectuer du calcul haute performance. Nous constaterons que les logiciels de base (systèmes d'exploitation et les outils qui leur sont associés) ne sont pas adaptés aux exigences et spécificités du calcul parallèle et qu'il est nécessaire de concevoir et réaliser des supports exécutifs *ad hoc* (middleware du parallélisme!). À partir des besoins et exigences pour le développement d'applications parallèles, nous précisons ce qu'est un support d'exécution et positionnerons les principales contributions dans ce domaine vaste et très actif.

2.1 Nouvelles architectures pour le calcul parallèle

Avant d'aborder les supports exécutifs, il m'a paru qu'il était utile de présenter les évolutions actuelles des architectures pour le calcul parallèle. Le marché des supercalculateurs est aujourd'hui en régression constante et les constructeurs spécialisés ont presque tous disparu; seuls quelques grands constructeurs généralistes (SGI, Sun, IBM, NEC, etc.) conservent une offre significative de machines parallèles. La tendance est de constituer des grilles de calcul entre ces superordinateurs et c'est ce que nous présenterons dans la section 2.1.1. D'un autre côté, le marché de l'ordinateur personnel est en pleine explosion et les technologies d'interconnexion sont devenues très performantes et bon marché. La tentation est donc forte de construire à la main des "machines parallèles" avec des grappes d'ordinateurs personnels interconnectés par des réseaux à très haut débit. C'est ce qu'on appelle le calcul en grappes [23, 24] et nous présenterons cette voie dans la section 2.1.2.

2.1.1 Grilles de calcul

La motivation principale du métacomputing (calcul distribué à haute performance) est de globaliser et fédérer les ressources (calcul, stockage, instruments spécialisés) des supercalculateurs en les interconnectant par des technologies de réseau à haut débit. C'est ce qu'on appelle les grilles de calcul [59]. Les nouveaux problèmes de recherche qui se posent sont maintenant de définir des approches pour le développement et l'exécution d'applications sur des réseaux de supercalculateurs.

Les expérimentations les plus significatives de métacomputing aujourd'hui sont actuellement menées aux États-Unis par l'équipe de Ian Foster à Argonne et Carl Kesselmann à l'ISI. Leurs travaux se sont dans un premier temps appuyés sur la grille I-WAY [55] mise en place à l'occasion d'une démonstration pour la conférence SuperComputing de 1995. Ces expérimentations se sont ensuite poursuivies de manière plus permanente autour de GUSTO (Globus Ubiquitous Supercomputing Testbed Organization). Le principe d'I-WAY et de GUSTO est d'interconnecter à une grande échelle, c'est-à-dire au niveau d'un pays comme les États-Unis, un ensemble très hétérogène d'équipements, tels que des superordinateurs (SGI, IBM-SP, Convex), des équipements de réalité virtuelle (CAVE [36]), des équipements de stockage massif de données et des instruments spécialisés tels que par exemple des microscopes, des accélérateurs de particules ou des télescopes [80]. La technologie d'interconnexion utilisée est principalement basée sur ATM [16] à 155 Mb/s où TCP/IP a été choisi comme protocole de communication. Pour des raisons de portabilité et de difficulté de mise en œuvre, les interfaces de plus bas niveau (basées sur les couches d'adaptation AAL) n'ont pas été

retenues. Il y a quelques mois, c'était plus de 17 sites qui interconnectaient 330 supercalculateurs, avec approximativement 3600 processeurs fournissant une puissance globale cumulée de plus de 2 TeraFlop/seconde! La fédération de ces équipements permet donc un meilleur partage de ressources coûteuses pour supporter des applications exigeant à la fois puissance de calcul, mais également des ressources plus spécialisées au niveau stockage ou visualisation. L'enjeu consiste aujourd'hui à justifier de telles structures avec le développement de nouvelles applications impliquant une large communauté d'utilisateurs et de scientifiques et exploitant efficacement l'immense potentiel de ces nouvelles infrastructures. D'une manière un peu analogue à ce qui s'est passé avec Internet, il reste à affronter les réticences que peuvent avoir les utilisateurs à prêter leurs ressources de calcul pour les applications d'autres utilisateurs ou institutions.

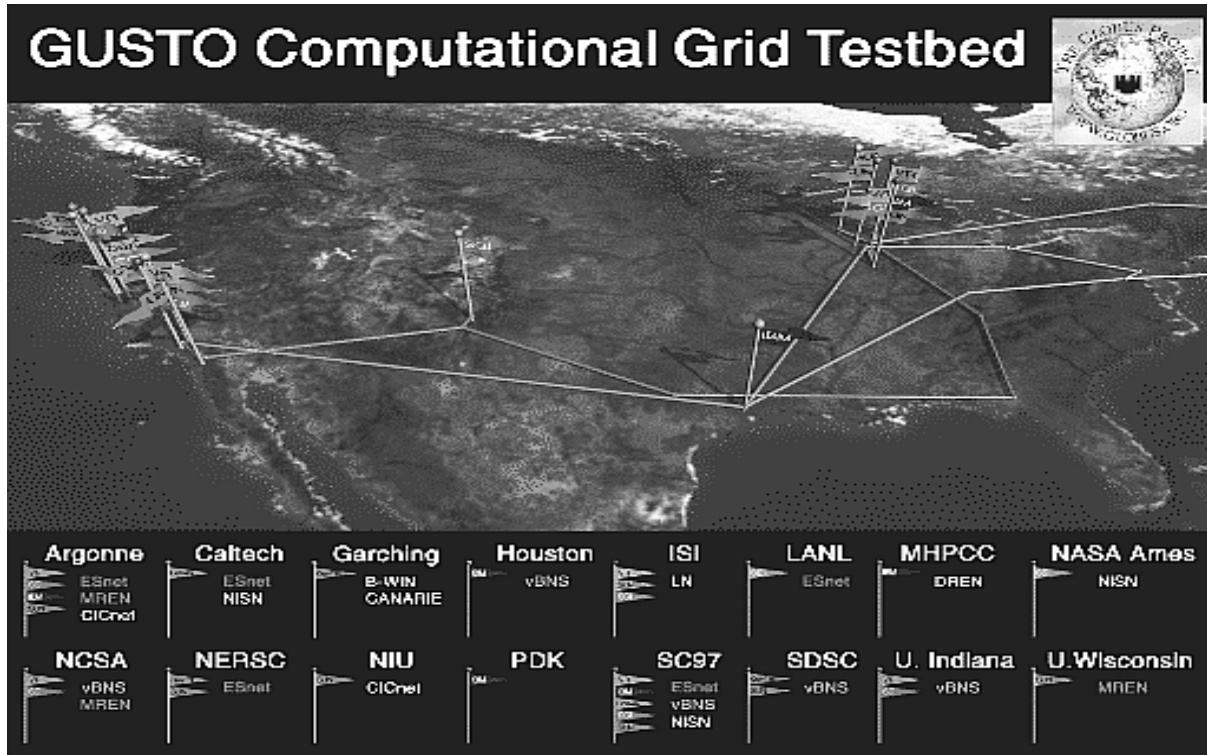


FIG. 2.1: Gusto : un exemple de grille de calcul.

De telles grilles de calcul, si elles sont relativement faciles à mettre en œuvre au niveau d'un pays comme les États-Unis, ne sont pas encore à l'ordre du jour des pays européens. Plusieurs raisons peuvent expliquer le retard du vieux continent dans ce domaine.

- L'Europe est relativement démunie en équipements de type supercalculateurs. Les rares supercalculateurs installés par exemple en France le sont surtout à des fins d'exploitation dans quelques organismes comme Météo France ou bien encore le Commissariat à l'énergie atomique (CEA).
- Les télécommunications ont été pendant plusieurs années en Europe des monopoles d'état, ce qui a fortement freiné le développement des nouvelles technologies. Aujourd'hui le retard des pays européens est important et il est difficile et coûteux de mettre en place un réseau performant sur une large étendue. Cependant, il reste à espérer que le marché des télécommunications s'ouvrira plus largement pour permettre la mise en place de tels réseaux et contribuer

au développement des recherches sur le métacomputing et de ses applications..

Après avoir décrit les grilles de supercalculateurs d'un point de vue architectural, nous allons maintenant aborder les logiciels disponibles sur les nœuds (machines parallèles) d'une grille. Nous identifierons ensuite quelques uns des problèmes posés par l'utilisation de ces logiciels.

Logiciels de base des supercalculateurs

Les logiciels de base fournis avec les machines parallèles s'appuient sur un système d'exploitation et sur un certain nombre d'outils spécifiques comme des compilateurs, des bibliothèques de communication et des outils d'exploitation. Les constructeurs cherchent au travers de ces différents outils le moyen d'offrir aux utilisateurs et applications un niveau de performance proche du potentiel théorique annoncé dans les spécifications du matériel. Nous décrirons dans cette section quelques-uns de ces outils.

Afin de tirer pleinement profit des capacités des processeurs, les constructeurs proposent des compilateurs optimisés (principalement pour les langages Fortran et C) exploitant efficacement le micro-parallélisme des programmes et les spécificités architecturales des processeurs (cache, pipeline, etc.). De nouvelles directives et options de compilation sont généralement ajoutées aux options classiques des compilateurs. L'utilisation de ces compilateurs est alors vivement recommandée pour les applications et des évaluations de performances ont montré des gains significatifs¹ par rapport à des compilateurs du domaine public comme par exemple ceux du GNU. L'inconvénient majeur de l'utilisation des compilateurs et de leur spécificité est bien évidemment un manque de portabilité et d'adaptabilité dû au fait que, d'un constructeur à l'autre, ces directives et options sont complètement différentes.

En ce qui concerne les bibliothèques de communication, les constructeurs se doivent de fournir aux utilisateurs et aux applications les moyens d'accès aux meilleures performances des réseaux d'interconnexion des machines parallèles [75]. Pour atteindre cet objectif, les premières bibliothèques de communication fournies avec les machines parallèles étaient spécifiques pour chaque constructeur [35, 101]. Là encore, l'inconvénient d'une telle démarche est évidemment un manque de portabilité des applications. Depuis la définition du standard de communication MPI, la tendance est aujourd'hui de proposer des implémentations efficaces [62] de MPI sur les machines parallèles. Toutes les machines parallèles ou presque disposent aujourd'hui de telles implémentations. Malheureusement, il faut regretter le manque d'interopérabilité [119] qui interdisent à deux programmes MPI s'exécutant sur deux machines issues de deux constructeurs différents de communiquer entre elles. Chaque implémentation MPI peut avoir sa propre pile de protocole propriétaire et ces différentes piles sont alors totalement incapables d'interopérer. Pour faire face à ce problème d'interopérabilité, deux voies sont à envisager : une première consiste à utiliser la technologie CORBA pour faire coopérer des applications parallèles, c'est une technologie qui est utilisée par l'équipe de Thierry Priol à Rennes [110]. Concernant la technologie CORBA, nous pensons qu'elle constitue une solution intéressante dans le cadre du couplage de codes, mais qu'en aucun cas elle ne peut être considérée comme une alternative au standard de communication MPI. La seconde approche envisageable pour faire interopérer des applications MPI serait évidemment d'étendre la spécification MPI en définissant les directives pour les futures implémentations MPI. C'est ce qui est en train d'être réalisé par un comité d'industriels qui ont rédigé une première spécification appelée IMPI². Des implémentations interopérables de MPI ont par ailleurs été développées dans le projet Globus (MPICH-G [57]) ou bien PACX-MPI [11] de l'université de Stuttgart.

¹De l'ordre de 35% pour des applications d'algèbre linéaire.

²Interoperable MPI : <http://impi.nist.gov/IMPI>.

Les machines parallèles sont comme la plupart des ordinateurs accessibles et utilisables potentiellement par plusieurs applications. Dans le cas des machines et applications séquentielles, c'est le système d'exploitation qui assure un partage équitable des ressources. Dans le contexte du parallélisme et des applications à haute performance, le temps partagé et l'utilisation des ressources de calcul par plusieurs utilisateurs interactifs peut poser de sérieux problèmes de performances. Lorsqu'une application parallèle s'exécute sur un ensemble de processeurs, la totalité des ressources doit être réservée et allouée à cette application pour garantir une efficacité maximale. Ce problème est tout à fait similaire à celui des premières générations d'ordinateurs et la solution avait été alors de soumettre les travaux à un moniteur d'enchaînement de programmes. Des outils pour l'ordonnement et l'exécution de programmes parallèles ont été très rapidement fournis avec les machines. L'utilisateur soumet l'exécution de son programme avec des outils comme LoadLeveler [76] ou LSF³ en spécifiant les besoins pour exécuter correctement son programme, comme le nombre de processeurs, l'espace mémoire, la bande passante réseau. En fonction des besoins exprimés pour une application, le système d'ordonnement tentera d'allouer les ressources nécessaires à l'application pour s'exécuter. À ce niveau aussi, chaque constructeur fournit ses propres outils d'ordonnement et il est regrettable de constater le manque d'interopérabilité des solutions disponibles.

Après avoir étudié les grilles de calcul et les supercalculateurs, nous allons maintenant étudier les grappes (*clusters*) de stations.

2.1.2 Grappes de stations

Comme nous venons de le voir dans la section précédente, les machines parallèles ne sont aujourd'hui accessibles financièrement qu'à des grands organismes et laboratoires de recherche. L'étroitesse de ce marché a entraîné un grand écrémage parmi les constructeurs spécialisés⁴ au profit de quelques constructeurs généralistes (IBM, SGI, Sun, Fujitsu) continuant à occuper ce créneau. L'architecture des machines parallèles repose aujourd'hui sur des processeurs généraux (PowerPC, Mips, Intel, Sparc) communiquant par des réseaux d'interconnexion spécialisés, performants [75] mais coûteux.

Le coût de ces machines parallèles étant prohibitif, une alternative nouvelle s'ouvre avec les réseaux de communication à très haut débit permettant d'interconnecter très efficacement et à un coût moindre des équipements de type station de travail ou PC. C'est l'approche dite des grappes de stations (*Cluster Of Workstations*). Les performances de ces réseaux à haut débit atteignent le Gigabit/seconde et constituent de sérieux concurrents en terme de performance pour les réseaux des machines parallèles.

L'exemple de réseau à haut débit le plus significatif est le réseau Myrinet [15] de la société Myricom qui est aujourd'hui largement utilisé pour construire des grappes de PC ou de stations Sun. Il se construit autour de liens en full-duplex à 1,28Gb/s. La longueur des liens est limitée à 30 mètres. La technologie Myrinet employée pour les commutateurs est dérivée de celle employée dans les superordinateurs et basée sur un routage *wormhole* autour d'un *crossbar* 8×8 . La carte d'interface réseau est équipée d'un processeur programmable (LANai) disposant d'un Méga-octets de mémoire S-RAM accessible par le bus d'entrées-sorties du processeur. Les performances de ce type de réseau sont mesurées à environ 6 μ s en latence et à plus de 100 Méga-octets en bande passante avec les interfaces de communications disponibles.

Des grappes à plusieurs centaines de stations ont été mises en place dans plusieurs universités,

³Load Sharing Facilities : <http://www.platform.com>

⁴Sauf peut-être encore TERA (<http://www.tera.com>) !

comme par exemple en Hollande dans l'équipe d'Henri Bal⁵, à l'université d'Illinois par l'équipe d'Andrew Chien⁶ avec une configuration à 128 PC bi-processeurs ou bien encore au laboratoire Argonne⁷ avec une configuration à plus de 256 unités de calcul. À Lyon, des configurations de taille plus modeste à 12 PC (PoPC : pile of PC) interconnectés par Myrinet ont été mises en place à des fins expérimentales en collaboration avec la société Matra Systèmes et Information.



FIG. 2.2: PoPC : une grappe de 12 PC interconnectés par Myrinet.

L'architecture d'une grappe peut donc se définir comme un ensemble de nœuds interconnectés par un réseau de communication. À partir de cette description synthétique, on pourrait croire qu'une grappe ne constitue en fait qu'une architecture répartie à petite échelle, en fait un mini-Internet. Il existe un certain nombre de spécificités qu'il est important d'avoir à l'esprit.

Homogénéité. Une grappe est un système fortement homogène construit autour d'un ensemble de processeurs de même type, d'un espace mémoire physiquement distribué, d'un système d'exploitation et d'un même ensemble d'utilisateurs. La différence la plus notable avec un système distribué à grande échelle comme peut l'être Internet est très certainement l'homogénéité de l'architecture, des logiciels et des utilisateurs. Cela donc a pour conséquence une intégration uniforme des ressources et évidemment des mécanismes de communication allégés. Des mécanismes comme l'encodage (hétérogénéité) et l'encryptage (sécurité) des données n'ont pas lieu d'être dans les modules de communication à l'intérieur des grappes.

Forte localité. Tous les nœuds d'une grappe sont connectés de manière très rapprochée via un réseau à très haut débit. Les caractéristiques de ce réseau sont une très faible latence et une bande passante importante. Cette bande passante est très économique puisqu'elle n'est pas fournie par une compagnie de télécommunications. Comme nous l'avons vu dans les paragraphes précédents, les communications dans les grappes sont efficaces parce qu'elles s'appuient sur des interfaces et protocoles spécialement conçus pour ce type de communications. C'est tout à fait le contraire des systèmes distribués, comme le Web où les communications sont relativement lentes, peu fiables et onéreuses car étant bien souvent facturées par un opérateur. Le haut débit dans les communications bouleverse quelque peu les rapports d'échelle et font par exemple que les communications à l'intérieur d'une grappe sont beaucoup plus performantes que les temps d'accès au disque. C'est ainsi que des recherches sont par exemple menées en système

⁵<http://www/cs/vu.nl/~bal/das.html>

⁶<http://access.ncsa.uiuc.edu/CoberStories/SuperCluster/super.html>

⁷<http://www-unix.mcs.anl.gov/~evard/chibacity>

pour mettre en place des mécanismes de va-et-vient (*swapping*) et de mémoire virtuelle et sauvegarder les processus ou les pages sur les mémoires distantes [54, 25].

Confiance. Les nœuds d'une grappe se partagent une même base d'utilisateurs et il n'est donc pas utile de dresser des barrières entre eux. La charge globale du système peut être partagée entre les différents nœuds et un nœud peut en remplacer un autre en cas de défaillance. Ceci est bien évidemment à opposer aux réseaux à grande échelle comme Internet où des mécanismes sophistiqués de protection et de sécurité doivent être mis en place.

Ces trois propriétés fortes caractérisant les grappes vont donc avoir certaines conséquences sur l'utilisation de ces grappes, sur la configuration des systèmes et aussi sur la nature des supports exécutifs et couches de communication.

Logiciels de base des grappes Contrairement aux machines parallèles qui sont vendues avec le système d'exploitation du constructeur, la particularité des grappes est qu'elles sont construites autour d'équipements de type PC ou stations. Le choix du système d'exploitation n'est pas imposé et c'est donc l'utilisateur qui décide en fonction de ses besoins. Aujourd'hui, c'est principalement UNIX et ses versions du domaine public qui sont retenues (Linux, FreeBSD) pour les facilités de développement qu'ils offrent en laissant disponibles les sources complètes du système. Cela permet aux équipes de recherche de développer plus facilement de nouveaux pilotes de périphériques (*drivers*). D'autres équipes choisissent le système Windows NT pour les débouchés industriels potentiels qu'il ouvre et l'immensité de la base installée de ce système.

Les interfaces de programmation fournies avec les cartes d'interface réseau sont aujourd'hui propriétaires et relativement difficiles à utiliser. C'est le cas par exemple des interfaces de communication GM ([95]) pour Myrinet ou bien encore le driver Dolphin pour SCI[47]. La tendance plus en vogue est de se tourner vers des interfaces développées dans des laboratoires de recherche (Active Message, Fast Message [99], SBP [118], BIP [108]) dont les performances⁸ sont souvent largement supérieures à celles obtenues avec les interfaces des constructeurs. Le principe de base de ces interfaces est de réaliser l'essentiel des communications en contexte utilisateur évitant ainsi les interactions coûteuses avec le noyau du système et aussi les recopies de données superflues. Un effort de standardisation de ces interfaces est aujourd'hui mené par plusieurs industriels (Compaq, Intel, Microsoft) pour unifier ces interfaces. Cela a abouti à la proposition d'une nouvelle interface : *Virtual Interface Architecture* [31, 51]. À l'instar des autres interfaces de communication, VIA reprend le principe des communications en contexte utilisateur mais comprend également des suggestions faites aux constructeurs de cartes d'interface réseau pour intégrer des mécanismes matériels à leur carte qui devraient faciliter le développement d'implémentations efficaces de VIA.

2.1.3 Quelles nouvelles problématiques logicielles ?

2.1.3.1 Grilles de calcul et métacomputing

Internet et le Web n'auraient certainement pas rencontré un tel succès si des outils simples (lecteurs de mail, *Mosaic*, *Netscape*, etc.) n'avaient pas été rapidement mis à la disposition des utilisateurs. Il est donc clair que le succès du métacomputing passe également par une mise à disposition d'outils et d'environnements de développement permettant de concevoir des applications exploitant efficacement le potentiel des infrastructures distribuées.

⁸C'est surprenant !

Ces dernières années, les travaux de recherche autour des environnements d'exécution parallèle se sont principalement concentrés autour d'architectures fortement homogènes (processeurs, mémoire, réseaux). Avec l'avènement du métacomputing et du calcul distribué à haute performance, une des caractéristiques principales des infrastructures est qu'elles seront fortement hétérogènes. C'est donc en partie au niveau de ces environnements que l'hétérogénéité devra être prise en charge. Si l'on sait aujourd'hui parfaitement communiquer des données entre processeurs hétérogènes, par exemple en utilisant des formats standard de représentation des données (XDR), il n'en est pas de même au niveau des protocoles de communication. Jusqu'à aujourd'hui, l'approche retenue était de s'appuyer sur un unique protocole de communication (TCP/IP) pour interconnecter les ordinateurs de la planète. Cependant, les recherches montrent les limites de ce protocole pour le calcul à haute performance. Dans le contexte du métacomputing, pour éviter de s'appuyer sur ce protocole unique, un des défis sera de réussir à faire cohabiter dans les environnements d'exécution (MPI, CORBA) les protocoles spécifiques et performants de communication (*e.g.* BIP⁹, VIA¹⁰) qu'on trouve sur des architectures parallèles ou des grappes, avec les protocoles de communication à grande échelle (TCP/IP/UDP).

Le déploiement de calculs lourds sur un ensemble de processeurs distribués dans différents pays et institutions exige que des règles de partage soient clairement définies, acceptées et respectées par les différents utilisateurs. Dans le contexte d'un centre de calcul et d'exploitation, les processeurs d'une grappe ou d'une machine parallèle peuvent être réservés aux applications locales du centre et par conséquent non utilisables pour des calculs provenant de l'extérieur. Il est donc impératif d'établir des règles précises d'exploitation et de partage des ressources comme cela pouvait se faire jadis dans les grands centres de calcul. Ces règles d'exploitation devront être négociées entre les différentes institutions et préciser à tout moment les ressources (processeurs, mémoires et disques) disponibles pour les applications du métacomputing. Ces règles devront être codifiées et contrôlées par les supports exécutifs. Le support exécutif devra donc permettre le déploiement des calculs sur les processeurs en garantissant le respect des règles d'exploitation et en fournissant éventuellement des mécanismes de migration des calculs vers d'autres sites dans le cas où un processeur et une machine auraient à être utilisés localement.

L'exploitation de ces infrastructures pose de nouveaux problèmes au niveau de l'administration des systèmes. Le travail d'administration est aujourd'hui principalement centralisé¹¹ et consiste essentiellement à gérer les comptes des utilisateurs et leurs environnements de travail. Dans le contexte du métacomputing, une des difficultés est de réussir à éviter l'alourdissement de cette charge d'administration. Il n'est par conséquent pas raisonnable d'envisager une administration où l'ensemble des utilisateurs posséderait un compte sur l'ensemble des machines du réseau¹². Il n'est pas non plus raisonnable d'imaginer une solution où un seul utilisateur (méta-utilisateur) serait le seul utilisateur connu et enregistré sur l'ensemble des machines. Cette approche présente l'inconvénient de n'offrir aucune protection entre les données et programmes des utilisateurs¹³. Une des difficultés est de trouver une solution intermédiaire qui n'accroisse pas trop le travail d'administration tout en fournissant un niveau de protection satisfaisant.

Dans l'optique d'une utilisation des superordinateurs comme nœuds d'une plate-forme de métacomputing, nous avons essayé de montrer que les logiciels disponibles auprès des constructeurs ne sont pas conçus pour une telle approche et qu'il est nécessaire de concevoir de nouveaux en-

⁹<http://lhpc.univ-lyon1.fr>

¹⁰<http://www.viarch.org>

¹¹Dans les centres de ressources informatiques.

¹²Cela limiterait fortement le nombre d'utilisateurs et de machines!

¹³Ce serait le retour à un système mono-utilisateur!

vironnements permettant des communications efficaces, aidant à la conception d'applications et à l'exploitation des ressources distribuées. Les aspects hétérogénéité (protocoles, données) et sécurité constituent des contraintes à prendre en compte par ces nouveaux environnements supportant le métacomputing. Les environnements Globus et Legion offrent des outils pour la programmation et l'exploitation des grilles de supercalculateurs et servent de support à des applications scientifiques.

2.1.3.2 Grappes de stations

Pour que le calcul à haute performance sur les grappes soit vraiment attractif, il paraît hautement stratégique de fournir aux utilisateurs des outils de développement qui puissent être aussi évolués de ceux qu'on retrouve sur les machines parallèles. Les constructeurs de machines parallèles s'étaient attachés à fournir le logiciel de base minimal avec des bibliothèques de communication, des compilateurs, des ordonnanceurs, etc. En l'absence de véritables constructeurs de grappes, il existe au niveau des laboratoires et centres de recherche en parallélisme un créneau extrêmement prometteur à occuper au niveau des logiciels et environnements pour les grappes de PC ou stations. Contrairement à ce qui s'est passé au niveau des architectures parallèles, il s'agira de proposer des environnements dont la portabilité est vraiment garantie sur un large éventail de grappes (stations et réseaux). Un des défis est de réussir à convaincre les utilisateurs des machines parallèles qu'on peut trouver sur les grappes de stations des services du même niveau que ceux fournis par les supercalculateurs.

Dans le cadre du calcul à haute performance, il est maintenant largement admis qu'il est crucial d'exécuter un maximum d'opérations en contexte utilisateur et de minimiser les interactions avec le système d'exploitation qui sont coûteuses. Nous avons décrit dans la section précédente le principe des bibliothèques de communication, mais c'est également le cas pour les bibliothèques de processus légers qui, lorsqu'elles sont développées en contexte utilisateur, offrent les meilleures performances. Pour offrir ce haut niveau de performances avec des grappes de station, il nous paraît judicieux de développer des outils évolués et efficaces qui soient exécutés en dehors du système d'exploitation. Comme nous le verrons dans le chapitre suivant, nous nous sommes efforcés dans nos travaux de développer un ensemble de composants en mode utilisateur pour le calcul et les communications. Une des tendances aujourd'hui au niveau du parallélisme et des supports exécutifs est de réussir à intégrer en mode utilisateur des services qui étaient assurés par les systèmes pour minimiser les changements de contexte entre le mode utilisateur et le mode noyau. C'est ce que nous allons étudier plus précisément dans la section suivante.

2.2 Systèmes d'exploitation et calcul parallèle

Après cette étude des nouvelles architectures pour le calcul parallèle, nous allons maintenant nous intéresser aux systèmes d'exploitation et à certaines de leurs fonctionnalités qui nous paraissent essentielles pour le calcul à haute performance. Il s'agira principalement des paradigmes fournis par les systèmes pour la gestion des activités et la gestion des communications. En ce qui concerne les activités, le concept central est le processus qu'on retrouve aujourd'hui sous différentes formes dans la plupart des systèmes d'exploitation. Les processus ont très rapidement constitué les unités d'exécution des premiers environnements de programmation parallèle (PVM ou MPI). Dès le début de nos recherches en 1990, notre attention s'est portée sur les concepts et technologies autour des processus légers qu'on identifie derrière le terme anglophone de "multithreading". C'est tout d'abord autour d'une machine parallèle à base de Transputers et du système d'exploitation

Helios que nos premières expérimentations ont été menées. L'originalité de ce processeur Transputer était de disposer d'un support matériel (des registres et instructions) optimisant la gestion de la multiprogrammation. L'interface de programmation du système Helios définissait aussi un ensemble d'opérations permettant de créer des tâches, terme générique qu'on peut aujourd'hui assimiler à celui de threads. Si ces processus légers nous ont vivement intéressés, c'est principalement pour les nouvelles possibilités qu'ils offraient pour le support d'applications à grain plus fin et la compilation de langages parallèles. De plus, ils constituaient une solution élégante au problème, maintes fois étudié en parallélisme, du recouvrement des communications par des calculs. C'est donc autour de ces nouveaux concepts que sont nées nos premières réflexions autour du projet PVC-BOX dont une des finalités était d'étudier les nouvelles technologies à objets (langages, méthodologies) dans le contexte du parallélisme et des architectures distribuées. Les premières évaluations du Transputer et des threads du système Helios ont été menées par Dominique Lazure pendant son stage de DEA [84] en 1990 et se sont ensuite poursuivies avec les premiers développements du projet PVC pendant la thèse de Luc Courtrai. Le projet PVC et son environnement exécutif seront détaillés plus longuement dans le chapitre suivant. Nous rappellerons brièvement dans ce chapitre les fondements du multithreading du point de vue des systèmes d'exploitation.

La seconde facette des systèmes qui retiendra notre attention concerne les communications. Nous montrerons de manière synthétique l'évolution des fonctionnalités de communication du point de vue des systèmes d'exploitation. Nous nous intéresserons plus particulièrement à deux des volets les plus importants dans les communications : 1) les interfaces de programmation qui sont déterminantes puisque les utilisateurs (des programmeurs plus ou moins expérimentés) ont à les mettre en œuvre dans leurs applications ; 2) les protocoles de communication qui définissent la manière dont les données sont véhiculées entre les différentes machines participant à une communication. Nous essaierons de montrer qu'à ce niveau, ni les interfaces de programmation des communications dans les systèmes ni les protocoles de communication ne sont vraiment adaptés aux besoins et aux exigences des applications parallèles. C'est donc pour répondre aux besoins des utilisateurs que de nouvelles interfaces de communications (PVM, MPI) ont été définies et implémentées sur un panel d'architectures assez large comprenant les machines parallèles et les grappes de stations.

Dans le contexte du parallélisme et des applications, le mode d'interaction entre les programmes et les systèmes d'exploitation peut engendrer des attentes ou blocages qui peuvent réduire fortement les performances. La plupart des appels au système ont été définis pour fonctionner dans un mode synchrone, c'est-à-dire que le programme utilisateur reste bloqué tant que la fonction système n'est pas terminée. C'est particulièrement pénalisant pour des opérations du système qui peuvent être assez longues comme des communications ou des entrées-sorties. Le recouvrement des temps d'attente par du calcul constitue un des éléments clés pour obtenir de hautes performances. Alors qu'au niveau architectural de puissants dispositifs d'entrées-sorties ou de communications sont parfois disponibles, l'interface de programmation synchrone des systèmes permet difficilement aux utilisateurs d'en tirer vraiment profit. C'est une des raisons qui explique pourquoi les interfaces des systèmes proposent maintenant de plus en plus de possibilités pour gérer l'asynchronisme, soit par exemple en utilisant des processus légers, soit en définissant des interfaces de programmation asynchrones, comme par exemple pour les entrées-sorties (POSIX *asynchronous input/output*).

Comme nous le verrons dans les paragraphes suivants avec les processus légers ou les communications, le coût des appels au système d'exploitation peut s'avérer important et donc nuisible aux hautes performances. Une des tendances qu'on observe aujourd'hui est de réaliser au niveau utilisateur un certain nombre de fonctionnalités de bas niveau qu'on pourrait simplement croire dévolues aux systèmes. C'est par exemple le cas de processus légers qu'on peut implanter complètement dans

une simple bibliothèque de fonctions, ou bien encore des communications efficaces qui sont prises en charge sans aucune intervention du système. Nous constaterons quand même que les approches en contexte utilisateur présentent un certain nombre d'inconvénients et nous pensons qu'à l'avenir les composants seront mixtes, c'est-à-dire réalisés à la fois en espace utilisateur et en espace noyau. C'est en particulier ce que nous verrons au niveau des communications avec le standard VIA qui identifie clairement les interactions entre le système et la partie des communications qui se fait en contexte utilisateur.

2.2.1 Multithreading

Ces dernières années, les threads¹⁴ sont devenus des éléments incontournables des systèmes d'exploitation. Les concepteurs et implémenteurs de systèmes d'exploitation ont rapidement été attirés par les processus légers, en particulier pour tirer pleinement bénéfice du potentiel des architectures multiprocesseurs à mémoire partagée (SMP¹⁵). Ils constituent de plus une solution élégante et efficace au fameux problème de recouvrement des entrées-sorties qui constituait un des obstacles notables pour les serveurs. La plupart des systèmes d'exploitation du marché proposent aujourd'hui des implémentations et des interfaces plus ou moins sophistiquées des processus légers. Dans le domaine du calcul à haute performance et du parallélisme, les processus légers sont aussi apparus comme étant une entité d'abstraction et d'exécution adaptée pour les supports exécutifs et les langages parallèles.

Le modèle traditionnel d'activités dans les systèmes, tel qu'il a été initialement défini dans Unix par B. Kerningham et D. Ritchie, reposait sur un unique concept de processus. Celui-ci peut se définir comme entité active associée à un programme ou une application en cours d'exécution. D'un point de vue structurel, il se caractérise par un flot séquentiel d'exécution qui évolue dans un espace d'adressage et auquel des ressources ont été allouées par le noyau du système. La création de processus est généralement assez coûteuse car elle est principalement réalisée au niveau du noyau du système parce que des mécanismes sophistiqués de partage de ressources doivent être mis en place entre les processus père et fils. Il en est de même pour les opérations de changement de contexte entre processus qui sont assez coûteuses car elles provoquent la sauvegarde ou la restauration complète de l'ensemble des registres et une ré-initialisation de l'espace d'adressage (MMU¹⁶). Les mécanismes de communication entre processus sont également coûteux puisque les processus ne coopèrent pas directement via de la mémoire commune, mais indirectement via des ressources allouées par le système. C'est le cas par exemple de fichiers ou bien des tubes (*pipes*) qui constituent une des abstractions les plus courantes dans les systèmes d'exploitation. Ce mode de communication, s'il convient bien à des applications à couplage faible, se révèle inadapté et inefficace lorsque le critère de haute performance est recherché.

Le modèle des processus légers (threads) peut très intuitivement se décrire comme une extension du modèle de processus pour permettre à plusieurs flots d'exécution concurrents d'évoluer dans un même processus lourd pour y partager l'espace d'adressage et l'ensemble des ressources du système qui sont allouées. La principale contribution des threads est donc de définir un modèle efficace de partage de ressources entre activités concurrentes et d'apporter une réponse aux limitations du modèle traditionnel de processus. Nous présenterons plus en détails dans les paragraphes 2.2.1.1 et 2.2.1.2 les principaux concepts et les techniques d'implémentation des threads. Cette technologie s'est très rapidement imposée chez les constructeurs et concepteurs de systèmes et presque tous les

¹⁴Processus légers.

¹⁵Symmetric Multi-Processors.

¹⁶Memory Management Unit.

systèmes aujourd'hui fournissent une bibliothèque de processus légers à leurs utilisateurs. Comme cela se passe souvent en informatique, chaque constructeur a essayé dans un premier temps de définir sa propre interface de programmation. Un standard a rapidement été proposé et adopté par l'IEEE avec l'interface de programmation des threads (POSIX-Threads 1003.4a) qu'on trouve aujourd'hui sur la plupart des architectures et systèmes du marché. Au niveau des systèmes d'exploitation, les implémentations les plus connues et les plus significatives ont été celle du micro-noyau MACH (Université de Carnegie Mellon, CMU) et celle du système Solaris de la société Sun Microsystems. Le micro-noyau Mach implémente donc des processus légers et a défini une des premières interfaces de programmation pour les processus légers C-Threads permettant à des programmes utilisateurs d'accéder aux threads du micro-noyau. Les auteurs ont en particulier été les premiers à étudier les principaux problèmes posés par les threads, en particulier les problèmes de réentrance des fonctions de bibliothèques. La société Sun Microsystems, avec son système Solaris, a été une des premières compagnies à fournir un produit largement utilisé sur les stations et serveurs de sa gamme. Le modèle des threads Solaris est assez novateur et permet d'assurer l'ordonnancement de processus légers de niveau utilisateur au-dessus d'un ensemble de processus légers de niveau noyau.

Les threads peuvent être implantés de deux manières : soit au niveau utilisateur, toutes les opérations sur les threads étant simplement supportées par des fonctions dans une bibliothèque, soit au niveau système et les opérations sur les threads sont alors exécutées au sein du noyau via un appel système. C'est ce que nous allons maintenant étudier dans les deux paragraphes suivants.

2.2.1.1 Threads de niveau utilisateur

Lorsque les threads sont implémentés au niveau utilisateur, l'ensemble des services et fonctions est fourni par une simple bibliothèque de fonctions liée à l'application. La caractéristique fondamentale de ce type d'implémentation est que l'ordonnanceur principal du système n'est pas conscient que les processus lourds qu'il gère contiennent plusieurs threads. À l'intérieur d'un processus, un ordonnanceur de thread est fourni par la bibliothèque et peut laisser la possibilité à l'utilisateur expérimenté de modifier les caractéristiques de l'ordonnancement. On parle alors d'ordonnancement à deux niveaux, l'ordonnanceur du système s'occupant des processus lourds et l'ordonnanceur utilisateur s'occupant des threads à l'intérieur d'un processus lourd. Il y a des avantages et des inconvénients pour cette approche d'implémentation des processus légers. D'abord en ce qui concerne les avantages, les opérations élémentaires de manipulation de threads (création, changement de contexte) sont efficaces puisqu'elles sont exécutées en espace utilisateur, sans appel système ni changement de contexte. Le second avantage de cette approche est de laisser inchangée la structure et l'interface des systèmes d'exploitation. Cela intéresse évidemment les concepteurs de systèmes, mais aussi les utilisateurs qui sont généralement assez perturbés par les profonds changements dans les systèmes. Les contributions les plus significatives pour de telles bibliothèques de threads l'ont été encore par la société Sun Microsystems et sa bibliothèque *lwp* [122] pour son ancien système d'exploitation SunOS¹⁷. Une des contributions universitaires les plus remarquables a été la librairie développée par Franck Mueller [92, 94] pour une implémentation portable de l'interface POSIX. Une des utilisations les plus significatives des processus légers de F. Mueller a été très certainement leur intégration dans le compilateur GNU-ADA [94] où ils ont permis l'implémentation des tâches du langage. Dans le contexte de nos travaux et du parallélisme, nous avons développé la bibliothèque Marcel que nous décrirons plus précisément dans le chapitre suivant.

Si les avantages des threads utilisateurs sont nombreux, il est aussi important de mettre en

¹⁷Ancêtre de Solaris.

évidence certaines de leurs limitations. Cela concerne principalement leur incapacité à exploiter le parallélisme interne d'une machine. Le principe même de ce type de thread étant d'appartenir à un processus qui constitue l'unité élémentaire d'ordonnancement, il n'est pas possible d'utiliser deux processeurs d'une machine avec deux threads utilisateurs appartenant à un même processus. Cette limitation est devenue d'autant plus pénalisante que les architectures SMP sont devenues aujourd'hui d'un excellent rapport coût/performance. Pour résoudre ce problème, la solution consiste à réussir à faire un mariage structuré et cohérent des threads de niveau utilisateur avec des threads de niveau noyau. C'est ce que fait aujourd'hui le système d'exploitation Solaris avec deux niveaux de threads clairement identifiés : les *lightweight processes* du noyau et les threads de niveau utilisateur.

2.2.1.2 Threads de niveau système

Par opposition aux threads utilisateurs, les threads de niveau système deviennent les unités élémentaires d'ordonnancement du noyau. La notion de processus lourd, telle qu'elle existait dans les systèmes, se construit alors autour d'un thread système auquel sont affectées un ensemble de ressources mémoire et système. L'ensemble des primitives et opérations sur les threads s'exécutent au travers d'appels au système et nécessitent donc un changement de contexte du processeur pour passer en mode noyau. Ces opérations sont par conséquent plus coûteuses comparées à celle des threads de niveau utilisateur. De tels threads exigent donc des modifications significatives au niveau des systèmes d'exploitation. Deux approches sont envisageables pour les intégrer dans les noyaux.

1. Re-concevoir complètement l'architecture du noyau et du système, c'est l'approche qu'a retenue la société Sun pour son système Solaris à partir d'un noyau System V.
2. Prendre un noyau existant, inclure la gestion des threads dans ce noyau en protégeant certaines sections critiques du noyau. C'est l'approche actuellement retenue pour la version 2.2 du système Linux à partir des threads noyau¹⁸ développés par Xavier Leroy de l'INRIA.

Comme nous le mentionnions dans le paragraphe précédent, l'avantage des threads de niveau noyau est de pouvoir exploiter efficacement le parallélisme des architectures SMP. Avec un seul processus où évoluent plusieurs threads de niveau système, il est possible d'avoir un parallélisme réel. De plus, lorsqu'un tel thread effectue un appel système bloquant, par exemple une communication ou une entrée-sortie, les autres threads noyaux peuvent continuer à s'exécuter normalement. On peut donc constater que les threads de niveau noyau constituent une solution technique intéressante pour les problèmes posés par les opérations bloquantes du système. L'inconvénient majeur des threads de niveau système est que leur gestion est assez coûteuse et que cela restreint leur intérêt dans des applications du parallélisme où le degré de parallélisme pourrait être plus important. Comme nous l'avons vu dans le paragraphe précédent où Franck Mueller utilisait sa bibliothèque comme support du langage ADA, il ne nous paraît pas judicieux d'utiliser des threads de niveau noyau pour des applications construites autour d'activités parallèles à grain assez fin.

2.2.1.3 Discussion

L'approche la plus intéressante en terme d'implémentation de processus légers est de réussir à faire cohabiter harmonieusement des threads utilisateurs avec des threads de niveau noyau. C'est la solution qui a été retenue par le système Solaris. Malheureusement, si cette intégration paraît conceptuellement intéressante, elle manque singulièrement de souplesse car les utilisateurs ou les

¹⁸<http://paulliac.inria.fr/~xleroy/linuxthreads/index.html>

concepteurs d'environnements exécutifs n'ont pas la possibilité d'agir finement sur l'ordonnement des threads utilisateurs. L'évolution actuelle de notre bibliothèque Marcel est de fournir ces deux niveaux de threads en laissant la possibilité aux programmeurs de placer et contrôler l'exécution des threads utilisateurs sur les threads de niveau noyau.

Ce travail a été mené par Vincent Danjean [37] pendant son stage à l'Université du New Hampshire. Il a en particulier intégré dans le noyau du système Linux le mécanisme des activations qui avait été proposé par T. Anderson [5]. La notion d'activation est assez proche de celle d'un thread de niveau noyau. Les différences principales sont les suivantes : c'est le programmeur qui décide explicitement de créer un thread de niveau noyau à partir du code d'une fonction de l'application. Dans le cas des activations, c'est le noyau du système qui décide de créer une nouvelle activation pour exécuter une fonction de l'application. Par ailleurs, une autre spécificité des activations est qu'à chaque fois qu'une activation se bloque ou se réveille dans le noyau, l'application en est informée. Pour informer l'application d'un tel événement, c'est un mécanisme de type *upcall*¹⁹ qui est mis en œuvre. Dans le cas d'une bibliothèque de threads de niveau utilisateur développés au dessus de threads de niveau noyau, cela permet une collaboration entre le noyau du système et l'ordonneur des processus légers de niveau utilisateur. Il est ainsi possible pour des processus légers de niveau utilisateur d'effectuer des appels bloquants sans bloquer les autres processus légers.

2.2.2 Communications dans les systèmes

Avec l'explosion des nouvelles technologies de réseau, les composants de communication sont aujourd'hui devenus essentiels dans les systèmes d'exploitation. Lorsque les technologies des réseaux n'étaient pas ce qu'elles sont aujourd'hui (performances, qualité de service, grappes), les communications pouvaient se résumer à savoir comment on allait permettre à deux processus d'une même machine de communiquer, sans avoir recours ni à des variables partagées ni à des outils de synchronisation de trop bas niveau comme peuvent l'être des sémaphores. Les communications peuvent alors se définir simplement autour de deux primitives simples : le **send** (*message*) pour émettre un message vers un processus récepteur et le **receive** (*message*) pour recevoir un message provenant d'un processus émetteur. Ces deux opérations sont à la base de tout système de communication et permettent donc à des processus d'échanger des données en définissant une sémantique précise de communication : synchrone, asynchrone, fiable, etc.

Le cas du système d'UNIX est intéressant car il illustre clairement les évolutions des technologies et des systèmes vis-à-vis des communications. Cela montre aussi qu'aucun concept unificateur n'a vraiment émergé, que les modes de communication foisonnent et qu'on a généralement procédé en définissant de simples extensions pour répondre à de nouveaux besoins. C'est ainsi qu'aujourd'hui la maîtrise de l'ensemble des techniques de communication disponibles dans les systèmes exige des compétences rares que n'ont pas forcément les développeurs d'applications de calcul intensif.

Dans les premières versions d'Unix définies par B. Kerningham et D. Ritchie, les tubes de communications constituent les seules possibilités de communication entre processus. Les envois et réceptions de messages s'effectuent au travers de lectures et d'écritures dans des descripteurs de fichiers partagés entre les processus communicants. Si ce mode de communication est adapté pour une exécution de deux processus sur une même machine, il ne l'est évidemment pas lorsque ces processus s'exécutent sur deux machines différentes. La principale raison est que l'implémentation du mécanisme de communication par tube repose sur une zone de mémoire partagée entre processus et un héritage de descripteurs. C'est pourquoi de nouveaux mécanismes ont été conçus pour permettre

¹⁹ Appel vers le haut en français!

la communication entre des processus s'exécutant sur des machines distantes. D'autres extensions ont également été apportées pour permettre à des processus de partager physiquement les mêmes données, sans être contraints de les communiquer dans des tubes. L'ensemble des extensions issues du System V reposent ainsi sur la notion de segment de mémoire partagée.

L'ajout d'extensions au système, si elle permet de conserver les fonctionnalités existantes, présente le sérieux inconvénient d'offrir une palette d'outils qui peut s'avérer trop complexe à maîtriser pour les utilisateurs.

2.2.2.1 Communications entre processus distants

Les extensions des communications entre processus distants ont été rendues nécessaires pour exploiter les premières technologies de communication. Nous définissons deux processus comme étant distants s'ils sont ordonnancés par deux noyaux de systèmes distincts. Les premières extensions qui sont apparues sont celles qui ont été à la base du développement d'Internet, à la fois du côté des protocoles (TCP/IP), mais également au niveau de l'interface de programmation (les *sockets*). Ces différents outils ont ensuite été utilisés pour implémenter les premiers outils Internet comme le transfert de fichiers ou dans la circulation des messages électroniques.

L'objectif des concepteurs des sockets était de définir une abstraction qui allait permettre d'implémenter des communications dans un schéma de type client-serveur. Une socket peut se définir comme point de communication autour duquel sont associés des processus qui vont pouvoir envoyer ou recevoir des données. Même si aujourd'hui elles sont principalement utilisées avec le protocole TCP/IP, elles peuvent être associées avec d'autres protocoles comme par exemple XNS (Xerox Internet Protocol). Cette interface de programmation supporte des modes de communication en mode totalement connecté ou déconnecté suivant les protocoles. Dans le contexte du parallélisme et des applications, le modèle de communication de type client-serveur se révèle assez limitatif et trop hiérarchique. L'autre raison qui fait que les sockets sont relativement peu utilisées directement dans les applications est que cette interface est de trop bas niveau et que les utilisateurs peuvent avoir des difficultés pour en maîtriser toutes les subtilités. Ces raisons poussent la communauté du parallélisme à s'intéresser à des interfaces de communication plus accessibles pour les utilisateurs et c'est ainsi qu'une interface comme MPI est devenue aujourd'hui presque incontournable.

Le second point à étudier dans un composant de communication est le protocole de communication qui définit comment les données sont échangées entre les ordinateurs. Aujourd'hui, le protocole le plus connu qu'on trouve dans les systèmes est TCP/IP qui constitue la brique de base du réseau Internet. Ce protocole a été conçu pour fonctionner à grande échelle, avec des réseaux à performances relativement modestes dont la fiabilité est relativement limitée. Ce protocole utilise en particulier des techniques pour partager équitablement la bande passante entre l'ensemble des connexions. Dans le contexte du parallélisme, les caractéristiques sont tout autres, l'environnement est fortement couplé et les pertes de messages sont suffisamment rares. C'est ainsi qu'au niveau des machines parallèles, les constructeurs ont défini des protocoles de communication propriétaires et efficaces pour les réseaux de leurs machines. En ce qui concerne les grappes de stations et leurs réseaux à haut débit, toutes les études montrent que les performances de TCP/IP sont décevantes dans ce cadre et ne permettent pas d'exploiter le potentiel de ces réseaux. Le problème est dû à la fois à un problème d'interface de communication en mode système, mais également au niveau du protocole et par exemple de la non adéquation du découpage des messages en paquets de tailles fixes comme peut le faire TCP/IP. C'est pourquoi les interfaces de communication pour les grappes constituent un sujet de recherche brûlant à la fois au niveau des interfaces et des protocoles.

2.2.2.2 Interface pour les réseaux à haut débit

Les recherches sur les interfaces de communication de bas niveau ont largement montré que les approches en contexte utilisateur (Fast Message [98], Active Message [126], U-Net [125]) étaient devenues incontournables pour obtenir de hautes performances (latence minimale et bande passante maximale), en particulier sur les réseaux à très haut débit. L'objectif des sociétés qui ont défini VIA était de définir une interface unique et donc standard pour les communications à haute performance. VIA s'appuie sur le principe des communications en contexte utilisateur en définissant précisément les interactions entre le système d'exploitation et le composant de communication qui s'exécute en contexte utilisateur. La spécification de VIA s'articule autour de deux modules principaux : 1) le *module noyau* qui initialise l'environnement de communication et interagit avec la gestion mémoire du système ; 2) le *module utilisateur* qui implémente les opérations en contexte utilisateur comme les *send* et *receive*. Il est important de noter que l'articulation entre ces deux modules est tout à fait indépendante de tout système d'exploitation. Pour s'en convaincre, il suffit de constater que les implémentations de VIA qu'on trouve à ce jour sont disponibles pour les systèmes Windows-NT, Linux et Solaris.

L'*interface virtuelle* (VI) est le concept clé de cette architecture (voir la figure 2.3). C'est la structure de données qui va permettre aux processus utilisateurs d'effectuer des communications efficaces. Une VI comprend deux files de messages : une file pour les envois de messages et une file pour la réception. Le programme utilisateur soumet des requêtes d'émission ou de réception qu'il ajoute à ces deux files. Une requête comprend principalement un descripteur indiquant le travail à effectuer par la carte d'interface réseau. Un programme utilisateur peut posséder plusieurs VI, une pour chaque connexion bidirectionnelle.

Cette interface de programmation dans sa première version comprend environ 40 primitives qui n'ont pas vocation à être utilisées directement par les programmes et les applications. La spécification de VIA ne donne aucune information sur les protocoles à définir pour les implémentations de VIA. Le niveau d'interface de communication de VIA constitue un exemple significatif et caractéristique que les environnements et supports exécutifs ont à intégrer. Actuellement, il y a de multiples travaux et développements autour de VIA : certains concernent le portage [46] de l'interface de communication MPI sur VIA. De notre côté, notre contribution a été de porter PM² et sa couche de communication sur VIA [18]. Microsoft et Intel collaborent au développement des couches de communication système (TCP/IP) au dessus de VIA.

2.3 Qu'est-ce qu'un support d'exécution ?

Comme nous venons de le décrire dans la section précédente, les architectures et systèmes peuvent s'avérer fort complexes à manipuler pour des utilisateurs, pas forcément très férus d'informatique et système. Les environnements et supports exécutifs tentent de masquer cette complexité aux utilisateurs et ils permettent également de masquer les évolutions technologiques des architectures et des systèmes. Ces évolutions concernent à la fois les unités de calcul (microprocesseurs) mais aussi les dispositifs de communication comme par exemple les réseaux locaux à haut débit (Myrinet [15], MPC [106], SCI [77], DEC Memory Channel [68]). Des recherches très actives ont également été menées pour concevoir des environnements de programmation parallèle complets comprenant un ensemble très riche d'outils allant du dévermineur de programmes parallèles jusqu'aux supports d'exécution de bas niveau, en passant par les régulateurs de charge. Nous nous intéresserons dans ce document principalement aux supports d'exécution en dégageant les principaux paradigmes qu'ils

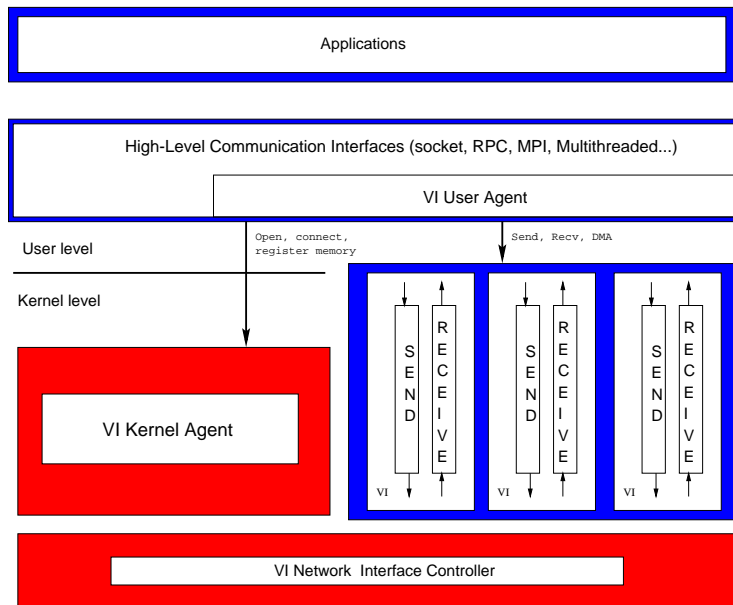


FIG. 2.3: Les différents composants de VIA.

proposent.

Le premier constat à faire autour des architectures parallèles et distribuées est que les constructeurs ont retenu comme système d'exploitation de leur plate-forme UNIX ou un de ses dérivés. Les qualités de ce système sont indiscutables en termes d'outils de développement, d'interactivité ou de disponibilité de logiciels. Cependant, le modèle de programmation UNIX et son interface de programmation ne correspondent pas exactement aux besoins des développeurs d'applications parallèles. Des fonctionnalités élémentaires comme la création de processus (**fork**) ou la communication en utilisant des **sockets** exigent des connaissances en système que possèdent rarement les développeurs de code scientifique. Le besoin s'est donc fait sentir de développer des couches logicielles intermédiaires offrant des paradigmes et des opérateurs plus évolués et plus simples à mettre en œuvre.

Un support d'exécution est généralement une couche logicielle développée au-dessus de l'interface des systèmes d'exploitation et qui fournit aux programmeurs des paradigmes de programmation parallèle plus évolués que ceux disponibles au niveau des systèmes. Les fonctionnalités du support sont accessibles, soit directement à partir de fonctions appelables d'un langage séquentiel (C, Fortran, C++), soit à partir d'extension de langages (C-Linda [27]), soit encore au travers de langages parallèles généraux développés avec des supports d'exécution. Les supports d'exécution parallèle constituent ce qu'on pourrait appeler le *middleware* du parallélisme comme peut l'être aussi CORBA dans le domaine de l'interopérabilité des applications dans un contexte réparti.

Le second constat concerne les systèmes d'exploitation des architectures parallèles et distribuées. Les différentes implantations et versions d'UNIX peuvent différer d'un constructeur à l'autre. Le programmeur doit alors effectuer un travail de portage et de paramétrage qui peut s'avérer fastidieux. D'autres problèmes, comme celui de l'hétérogénéité des architectures, peuvent aussi rendre difficile et ingrat le travail du développeur. Un des rôles des supports d'exécution sera de contribuer à améliorer la portabilité des applications, voire même de faire interopérer des applications s'exécutant dans des contextes hétérogènes.

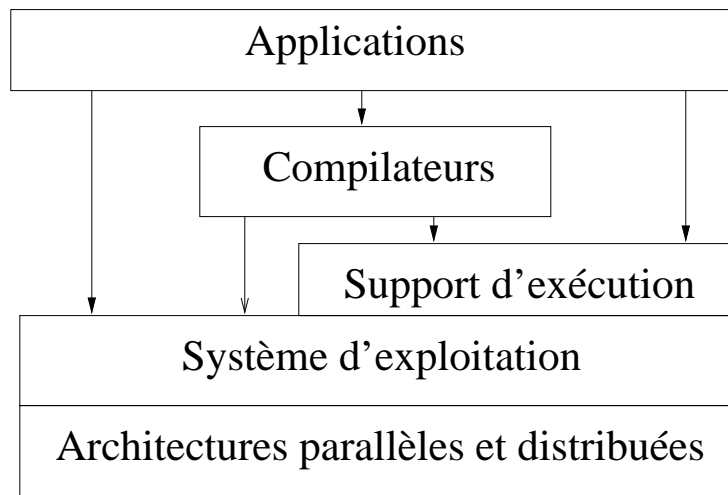


FIG. 2.4: Positionnement des supports d'exécution parallèle.

Les décideurs et utilisateurs d'applications exigent du parallélisme et des programmeurs un niveau d'efficacité proportionnel aux investissements matériels et logiciels. Le contexte de développement (algorithmique, langage, support) doit permettre aux programmeurs de tirer profit des performances intrinsèques des architectures parallèles. Le **support d'exécution** ne doit pas être un frein à l'obtention de bonnes performances. Le surcoût généré par ces environnements devra rester raisonnable en fournissant les paradigmes de programmation les plus évolués avec un niveau d'efficacité suffisant.

2.4 Modèles de programmation parallèle

Nous allons maintenant présenter les modèles de programmation parallèle les plus significatifs. Un des rôles des supports exécutifs est de fournir aux utilisateurs un modèle de programmation qui corresponde à leurs besoins. Bon nombre d'applications parallèles²⁰ sont malheureusement encore développées sans méthodologie d'analyse et de conception. Alors qu'une démarche d'analyse complète et structurée des applications parallèles serait nécessaire, le développement consiste souvent à prendre comme base initiale une version séquentielle de l'application et à identifier plus ou moins grossièrement ce qui peut être exécuté en parallèle. La conséquence d'une telle démarche est que la phase de développement s'avère extrêmement délicate, que les performances obtenues sont généralement décevantes par rapport à ce qui pouvait être espéré et que la maintenance de ces applications (portabilité, évolutivité) est difficile et coûteuse.

L'étude des modèles de programmation parallèle nous permettra cependant de dégager des paradigmes structurants à mettre en œuvre au niveau des supports d'exécution parallèle. Le fait de fournir dans les environnements des concepts de haut niveau aura l'avantage de contraindre le programmeur à avoir une démarche plus structurée dans ses développements. Parmi les différents modèles de programmation parallèle, nous en citerons quatre qui nous paraissent significatifs.

Le modèle data-parallèle a pris racine dans les machines synchrones SIMD. Le principe de base du parallélisme de données [38] est de définir des structures de données dites parallèles et

²⁰Mais également des applications séquentielles...

d'y associer des opérateurs. C'est l'exécution de ces opérateurs sur les structures qui est la source génératrice de parallélisme dans les applications. Les structures régulières, comme des tableaux ou vecteurs, sont particulièrement bien adaptées à ce modèle. L'inconvénient d'un tel modèle est qu'il est relativement peu adapté aux applications dont les structures de données et traitements sont irréguliers (matrices creuses, recherches arborescentes). Les travaux et recherches en data-parallélisme se poursuivent principalement avec le langage HPF [73] et les problèmes nouveaux de compilation posés par ce langage sur des architectures asynchrones à mémoire distribuée.

Le modèle à partage de mémoire est motivé par le souci de fournir un modèle de programmation simple. Le principe de ce modèle est de s'appuyer sur un espace partagé pour le stockage des données et objets globaux des programmes parallèles. La coopération et la communication entre les activités des programmes parallèles s'effectue au moyen d'opérations de lecture et écriture des objets partagés. Nous ne l'avons pas encore mentionné, mais ce modèle a aussi été fortement encouragé par le succès de quelques architectures matérielles à mémoire partagée (ou virtuellement partagée) (KSR1, Sun Enterprise, SGI Power Challenge) qui ont de tout temps occupé une part non négligeable du marché. Une des difficultés majeures dans un modèle à partage de mémoire est réussir à établir un bon compromis entre le degré de cohérence²¹ choisi (forte, faible, relâchée) et les performances, en particulier sur des architectures distribuées. Les contributions majeures dans ce domaine sont le modèle de programmation Linda défini à l'université de Yale [27, 28] et l'environnement ThreadMarks développé à l'université de Rice [4].

Le modèle à objets est considéré comme une approche incontournable dans le domaine du génie logiciel. Aujourd'hui, c'est principalement autour des langages de programmation (Java, C++, Eiffel) que la communauté se focalise, au détriment certainement des aspects analyse et méthodologie de conception. Tout le monde reconnaît en effet qu'il est difficile pour un programmeur, souvent formé à une méthodologie procédurale, de structurer son application en un ensemble d'objets et de classes. Les langages de programmation à objets ont l'avantage de placer le développeur dans un cadre conceptuel l'obligeant plus ou moins fortement à avoir une telle démarche. Beaucoup de travaux du domaine se sont donc naturellement concentrés sur les aspects langages de programmation, soit en définissant de nouveaux langages de programmation [3, 82], soit en étendant la syntaxe et la sémantique de langages à objets [81, 26], soit finalement en proposant des classes spéciales [79] pour l'implémentation de mécanismes parallèles. Ces recherches ont abouti à différentes propositions pour l'expression de la concurrence, de la synchronisation et du compromis entre le travail du programmeur, du compilateur et du système d'exploitation. Au niveau des compilateurs des langages à objets, il a été mis en évidence que des supports spécialisés et portables facilitaient fortement leur développement. C'est ce que nous montrerons avec les supports PVC et PM² qui ont été utilisés comme supports exécutifs de tels compilateurs.

Les modèles à processus communicants sont aujourd'hui les plus couramment utilisés et sont dérivés du modèle CSP²² défini par Hoare [74]. Un calcul parallèle est constitué d'un ensemble de tâches communiquant et se synchronisant par échange de messages. Les idées et concepts à la base des modèles à processus communicants sont issus à la fois des réflexions sur la programmation modulaire (encapsulation du code et des données), mais également²³ des modèles

²¹Comment sont perçues les modifications des objets ?

²²Communicating Sequential Processes.

²³Même si ce n'est jamais dit explicitement !

d'exécution et caractéristiques technologiques des machines parallèles. Le principe de ces modèles est de structurer chaque application comme un ensemble de tâches (processus) qui sont, un peu comme les procédures ou modules d'un programme séquentiel, l'unité de conception et de décomposition des calculs. Chaque tâche exécute séquentiellement un ensemble d'instructions comprenant des opérations de calcul, des accès en mémoire locale et des communications. Les différentes tâches du programme peuvent s'exécuter en parallèle sur les différents processeurs d'une machine parallèle ou sites d'un réseau de stations. Ces modèles ne font en général aucune supposition sur la topologie ou sur le placement des tâches du programme sur les processeurs ou sites. L'ensemble des tâches peut être *statique*²⁴ ou *dynamique*²⁵.

Cette rapide présentation des modèles de programmation a permis de faire apparaître quelques concepts majeurs pour le calcul parallèle et distribué. Le premier de ces concepts est celui de tâches qui constituent les unités de conception et de décomposition du calcul et qui sont la principale source génératrice du parallélisme. Suite à la phase d'analyse et de conception, le programmeur devra affronter un environnement d'exécution pour implémenter ces tâches avec les unités d'exécution (activités) disponibles dans l'environnement. Ce travail d'intégration dans l'environnement sera d'autant plus facilité que ces activités seront suffisamment souples pour supporter différents types de classes. Il est en particulier souhaitable que les coûts de gestion des activités soient raisonnables et compatibles avec le mode d'exécution du programme.

Les communications constituent certainement le second aspect fondamental au niveau des modèles de programmation parallèle. Comme nous l'avons remarqué, la tendance est de fournir au concepteur d'applications des abstractions plus évoluées que de simples opérations de type envoi/réception. L'appel de procédure à distance présente à notre avis beaucoup plus d'avantages en terme d'expressivité mais aussi d'efficacité. Il est également intéressant de noter qu'un tel mécanisme peut facilement être intégré au niveau d'un langage de programmation parallèle, d'un environnement de programmation, mais également au niveau de bibliothèques de communication de bas niveau.

Un des rôles les plus importants que doivent assumer les environnements et supports d'exécution est de fournir au développeur des paradigmes et abstractions suffisamment évolués lui permettant de "coder" facilement ses tâches et le schéma de communication associé. L'interface de programmation du support devra permettre d'exprimer les tâches et les communications. D'autre part, pendant la phase de développement, le programmeur devra disposer d'outils lui permettant d'améliorer les performances de son programme ou de mettre au point ce programme. C'est ce que nous étudierons dans la prochaine section.

2.5 Quelques problèmes difficiles

Si dans la section précédente nous nous sommes attachés à dégager quelques abstractions et paradigmes incontournables pour la programmation parallèle, nous allons maintenant étudier quelques dispositifs facilitant le travail des programmeurs. Un des aspects les plus délicats est très certainement la phase de détection et de correction des erreurs dans les programmes. C'est ce qu'on appelle plus communément la phase de mise au point ou encore le déverminage. C'est une étape presque incontournable quand on sait que les technologies et méthodologies du parallélisme sont encore mal maîtrisées par les développeurs. Nous essaierons dans cette section d'identifier les nouveaux problèmes posés par la mise au point de programmes parallèles et les solutions mises en œuvre pour

²⁴Les tâches sont toutes lancées au début du programme.

²⁵Les tâches peuvent être créées ou détruites à n'importe quel moment de l'exécution.

leur faire face.

D'autre part, lorsque les applications sont fortement irrégulières, la répartition des calculs sur les différents processeurs est délicate. Toutes les techniques élaborées pour les applications régulières, souvent basées sur la répartition des données, ne peuvent être appliquées aux applications irrégulières. L'irrégularité provient principalement du contenu des données manipulées (données initiales d'une application d'optimisation) ou de structures de données fortement irrégulières (matrices creuses non structurées) ou dynamiques. Le risque est alors d'observer pendant les exécutions des programmes des déséquilibres importants de charge des processeurs qui peuvent se révéler être fortement nuisibles pour les performances. Il se peut par exemple, que la totalité des calculs soit concentrée sur quelques processeurs et que la majorité des autres processeurs soient inactifs. Nous essaierons également de décrire quelques mécanismes que peuvent fournir les environnements pour aider le programmeur à gérer ces situations.

Si le problème de la régulation de charge constitue souvent un obstacle pour de bonnes performances, le recouvrement des communications par du calcul est également délicat à appréhender et peut aussi nuire à la qualité des performances obtenues. Il s'agit donc d'éviter la situation où le processeur serait en attente de la fin d'une communication et n'aurait aucun traitement à effectuer pendant cette attente. Ce problème est particulièrement critique dans les situations où le réseau de communication est lent en comparaison avec les performances du processeur. D'un point de vue algorithmique de l'application, le programmeur doit être capable de déterminer les phases de calcul pouvant recouvrir les communications. Du point de vue du support d'exécution, des opérations permettant de contrôler la terminaison des phases de communication doivent être fournies aux utilisateurs. Nous présenterons dans cette section le principe des opérations asynchrones définies dans MPI. Un des objectifs des environnements à base de processus légers est d'assurer ce recouvrement par un découpage des calculs en processus légers, le recouvrement s'effectuant grâce à l'ordonnement des processus légers. C'est ce que nous verrons dans le chapitre présentant nos contributions sur les supports d'exécution.

2.5.1 Déverminage de programmes parallèles

Dans un contexte parallèle, à cause du non-déterminisme, deux exécutions d'un même programme pour une même donnée peuvent produire deux résultats différents. Soit le programme parallèle est correct, alors le résultat des exécutions sera toujours bon. Soit le programme parallèle est erroné et certains résultats peuvent être bons, d'autres incorrects. À la seule vue d'un résultat d'une exécution de programme parallèle, on ne peut pas décider si le programme donnera toujours le bon résultat.

Le problème du non-déterminisme dans l'exécution de programmes parallèles provient du fait que l'ordre des événements peut différer d'une exécution à l'autre. Les programmes erronés peuvent produire des résultats différents si deux événements se produisent dans un ordre différent. Les causes du non-déterminisme sont liées à plusieurs problèmes : le non-déterminisme dans les langages de programmation parallèle, la charge du réseau de communication, la charge de calcul des processeurs.

Le déverminage de programmes parallèles peut se faire, avant pendant ou après l'exécution.

Avant l'exécution. Par analyse statique du source des programmes, il est possible de détecter certains problèmes qui pourraient se produire pendant l'exécution, par exemple certains inter-blocages. On peut aussi citer les recherches plus théoriques qui se font dans le domaine de l'analyse [9, 89] de preuve de programmes parallèles.

Pendant l'exécution. Le déverminage risque alors de perturber le comportement du programme

parallèle. Il semble naturel d'imposer que les exécutions d'un programme avec et sans déverminage produisent le même résultat, avec la même séquence d'événements. Cette exigence rend difficile ce type de déverminage.

D'autre part, les fonctionnalités des dévermineurs de programmes séquentiels sont difficilement réalisables avec des programmes parallèles s'exécutant sur des machines parallèles.

Visualiser les variables d'un programme parallèle. La visualisation est plus délicate dans un cadre où la mémoire est distribuée. L'accès au contenu des variables exige bien évidemment l'utilisation du réseau de communication pour accéder aux variables stockées dans les mémoires distantes. De plus, on peut avoir, dans le cas des modèles d'exécution de type SPMD, des variables globales dupliquées sur les différents processeurs.

Suspendre l'exécution d'un programme. Beaucoup de questions se posent pour arrêter un programme parallèle : l'arrêt doit-il être complet sur toutes les activités du programme ? ou alors, l'arrêt peut-il être partiel et ne concerner qu'une ou plusieurs activités ? Généralement, l'arrêt des programmes est complet et concerne toutes les activités. Le problème est alors de stopper, simultanément ou de manière cohérente, toutes les activités du programme.

Exécuter un programme en pas à pas. Que pourrait signifier une exécution en pas à pas sur un programme parallèle ? Au niveau d'une activité, au niveau de l'ensemble des activités du programme ?

Beaucoup de questions sont posées. Il est souvent difficile d'y répondre et de proposer des réalisations. Dans le cadre du projet PVC, nous nous sommes résolument orientés vers le déverminage de programmes après une première exécution de référence.

Après l'exécution. Beaucoup d'outils de déverminage sont basés sur le principe de la génération de traces pendant l'exécution de programmes. La perturbation du comportement de l'application pendant la génération de traces doit rester faible, voire nulle. Les traces permettent ensuite, suivant les environnements, soit d'analyser l'exécution et d'essayer d'en détecter les anomalies, soit de réexécuter les programmes.

À ce jour, peu d'environnements de programmation parallèle ont été fournis avec des outils pour la mise au point. Plusieurs constructeurs ont cependant fourni des outils spécifiques pour leurs machines parallèles, mais seulement pour des environnements de type MPI. Cependant, aucune solution ne s'est vraiment imposée dans la communauté. Dans le cadre des travaux de notre équipe, Jean-François Roos [113] s'est intéressé à définir un mécanisme de réexécution de programmes et a proposé un mécanisme de point d'arrêt distribué dans le langage de programmation à objet BOX. Des travaux similaires ont été menés dans le projet Apache de Grenoble pendant les travaux de thèse d'Alain Fagot [53] et dans le projet Guide de Grenoble [78] pour un système et un langage à objets persistants.

2.5.2 Régulation de charge

S'il est un domaine applicatif où la régulation de charge est un enjeu capital pour de bonnes performances, c'est bien celui des applications fortement irrégulières. Les problèmes à forte irrégularité sont ceux dont le graphe de précédence est complètement imprévisible et ne peut donc pas être utilisé pour réaliser un ordonnancement des tâches de l'application parallèle. Les informations qu'on pourrait en extraire seraient très rapidement obsolètes et comme, de plus, il n'est pas possible de les utiliser pour prédire le comportement futur elles sont, de toute façon, inutiles. L'ordonnancement

ne peut, par conséquent, être effectué que de manière purement dynamique durant l'application en agissant, *a priori* et/ou *a posteriori*, sur la localisation et la date de lancement des tâches, par exemple en retardant leur exécution afin de les utiliser pour résoudre un déséquilibre découvert ultérieurement.

Le haut degré d'irrégularité des applications de cette classe provient de leur non-déterminisme. Si on se reporte à la définition de l'irrégularité d'une application, ordonnancer les activités d'une telle application est un problème complexe du fait de la trop grande quantité d'informations nécessaire à la description de l'application et surtout parce que ces informations ne sont disponibles qu'au moment où les tâches doivent être ordonnancées.

Exemple L'exemple probablement le plus connu de cette classe d'applications est celui du parcours d'arbre de recherche par une méthode telle que la méthode Branch&Bound. Cette méthode est utilisée pour résoudre de façon exacte des problèmes d'optimisation combinatoire, contrairement aux méthodes heuristiques comme la méthode tabou ou les algorithmes génétiques qui obtiennent des solutions approchées dont elles ne prouvent pas l'optimalité. Son principe est d'explorer un arbre de recherche décrivant l'ensemble des solutions possibles du problème. Les nœuds de l'arbre contiennent des solutions partielles, les feuilles contiennent des solutions complètes. Par une heuristique, à chaque nœud de l'arbre est associé un poids qui dépend de la solution partielle qu'il contient. Cette heuristique a la particularité d'être strictement monotone²⁶ suivant la profondeur dans l'arbre, c'est-à-dire que tous les nœuds contenus dans un sous-arbre auront une évaluation supérieure à celle du nœud racine du sous-arbre.

En utilisant cette évaluation et cette propriété, il est possible de couper, c'est-à-dire de ne pas explorer, des branches de l'arbre durant l'exploration et donc de réduire la taille totale de l'espace exploré. Un des critères principaux pour juger un algorithme de Branch&Bound est sa capacité à trouver tôt dans l'exécution de bonnes solutions, c'est-à-dire proches de l'optimal, permettant ainsi de couper beaucoup de branches. La quantité de nœuds élagables ainsi que leur localisation dans l'arbre ne peuvent être calculés avant leur exploration, ce qui est la cause principale du haut degré d'irrégularité de ce type d'applications. En effet, il est impossible d'évaluer au début de l'exploration d'un sous-arbre quel pourcentage de ce sous-arbre il faudra réellement explorer et donc la charge de calcul qu'il nécessitera.

Un autre type d'applications représentatif de cette classe, mais non issu de l'optimisation combinatoire, est celui de la simulation d'un ensemble de particules. La simulation du déplacement et des interactions d'un ensemble de particules dans un espace fini est une des applications principales développées par les physiciens et chimistes [14] gros utilisateurs de machines parallèles. Une des applications classiques est celui de la dynamique moléculaire. Elle consiste en la modélisation du mouvement de particules élémentaires, qui peuvent être des atomes, des molécules ou encore des électrons, dans un espace clos en prenant en compte les phénomènes d'attraction/répulsion, de collision, de fusion, etc. L'évolution des données, c'est-à-dire des particules, est complètement imprévisible et peut varier considérablement pour des données initiales pourtant très proches, ce qui rend impossible toute tentative d'analyse *a priori* du comportement de l'application. L'exploitation d'un graphe de calculs serait difficile, ne serait-ce que par l'incertitude portant sur l'existence des particules à la prochaine étape. Dans ce cas la seule solution viable est un ordonnancement à la volée des tâches, représentant une (ou un ensemble de) particule, sur les processeurs afin de garantir une exécution efficace. Pour cela on utilise généralement un algorithme de placement pour répartir initialement les tâches puis on modifie ce placement pendant l'exécution pour prendre en compte les

²⁶Croissante dans le cas d'une minimisation, décroissante dans le cas d'une maximisation.

variations de charge causées par l'irrégularité de l'algorithme, ici par les créations et les destructions de particules.

La régulation de charge est un des problèmes difficiles en parallélisme et a fait l'objet de nombreuses recherches à la fois théoriques et expérimentales. L'objectif de la régulation de charge est de réussir à répartir équitablement la charge générée par une application parallèle sur l'ensemble des ressources dédiées à cette application. C'est un problème complexe car il a été montré que le bon choix d'une stratégie de régulation est lié à la nature de l'application. Pour des applications de nature régulière, une simple répartition des données sur les différents processeurs sera suffisante. Pour des applications plus irrégulières, la répartition des données au début du calcul ne suffit pas, des déséquilibres peuvent apparaître pendant l'exécution.

Un certain nombre de services sont à prévoir au niveau des supports exécutifs pour faciliter la conception de modules de régulation de charge. Tout le monde semble aujourd'hui d'accord pour dire qu'il n'existe pas de régulateur miracle, mais qu'il est important de fournir des régulateurs génériques. Pour construire de tels régulateurs, les supports exécutifs se doivent de fournir des opérateurs de placement et même de déplacement d'activités.

Placer une tâche sur un processeur. Cette opération de placement est disponible au moment de la création sous la forme d'un paramètre qui donne le numéro du processeur où sera créée la tâche.

Obtenir des informations sur l'état de charge d'un processeur. Pour concevoir un mécanisme de régulation de charge, le programmeur aura besoin de connaître des informations sur la charge d'un processeur, l'occupation mémoire, et même des informations sur la charge et les performances du réseau de communication.

Migrer une tâche d'un processeur vers un autre. Le régulateur de charge peut, dans les situations où des déséquilibres apparaissent, décider de déplacer des charges de calcul d'un processeur vers un autre. De nombreux travaux sur la migration ont été menés dans le contexte d'environnements et de systèmes où les tâches sont supportées par des processus lourds²⁷. Les performances des systèmes de migration de processus lourds sont décevantes et n'ont pas vraiment d'intérêt dans le contexte du calcul parallèle. Or, nos travaux se situent plutôt dans ce contexte et nous avons montré que cette opération de migration pouvait être intéressante dans le cas où les tâches sont légères. Dans le contexte des applications parallèles, notre position est d'affirmer que la migration est complémentaire à l'opération de placement initial des tâches pour corriger des situations de déséquilibre.

2.5.3 Recouvrement des communications

Le problème du recouvrement des communications par du calcul a toujours constitué un obstacle majeur aux bonnes performances des applications parallèles. D'un point de vue exécutif, il peut se caractériser par le fait qu'un processeur soit inactif pendant l'exécution d'une opération de communication. Cette situation apparaît naturellement avec des réseaux d'interconnexion peu performants où les communications peuvent alors freiner de manière très significative les calculs. Ce problème apparaît également avec des schémas d'exécution centralisés de type Maître-Travailleurs lorsque des processus Travailleurs attendent des requêtes provenant du processus Maître. Pendant cette période d'attente, les processeurs sont inactifs et les évaluations de performance montrent que ce problème apparaît vite pour des configurations avec un grand nombre de processeurs.

²⁷C'est-à-dire des processus UNIX !

Les communications asynchrones et les processus légers apportent des solutions au problème du recouvrement.

Pour faire face à ce problème, les supports exécutifs fournissent des opérations de communications dites non bloquantes ou asynchrones. C'est le cas des environnements de programmation PVM et MPI. Une opération de communication asynchrone se déroule en deux phases. La première phase consiste à initialiser une opération de communication et à la soumettre au support. Pendant que le support s'occupe de traiter cette demande, l'application peut poursuivre son exécution et dans un second temps, lorsqu'elle aura besoin des résultats de sa communication, demander à consulter le résultat de la communication.

```
/* Amorçage asynchrone de la communication */
MPI_Isend (vecteur, taille, MPI_FLOAT, destinataire,
           0, MPI_COMM_WORLD, &requete) ;

/* Quelques calculs pendant la communication */
calcul () ;

/* Attente de la fin de la communication */
MPI_Wait (&requete, &status) ;
```

FIG. 2.5: Exemple de communication asynchrone sous MPI.

Le programmeur doit donc définir un schéma de communication lui permettant de pipeliner ou paralléliser les communications pour recouvrir au mieux ses communications. Ce problème est assez difficile à appréhender et de nombreuses équipes de recherche s'intéressent à concevoir des algorithmes parallèles qui intègrent des mécanismes sophistiqués de recouvrement. C'est le cas par exemple de l'équipe de Jean Roman [45] à Bordeaux qui travaille sur l'algèbre linéaire parallèle.

2.6 Positionnement de notre contribution

Notre contribution se situe dans le domaine des supports d'exécution basés sur les processus légers. Nos travaux ont débuté en 1990 avec le projet PVC-BOX qui sera décrit dans le chapitre suivant. C'est aussi à cette époque que les premiers projets de conception d'environnements de programmation parallèle débutaient, avec parmi les plus connus, les environnements P4 [19, 22], PVM [123], Linda [27]. PVM a remporté un vif succès auprès de la communauté scientifique et des utilisateurs et il a fait prendre conscience de l'intérêt de tels environnements. Une des idées originales de PVM était de définir la notion de machine virtuelle "logicielle" qui regroupe dans une même entité des supercalculateurs, des serveurs de calcul de taille moyenne ou même de simples stations de travail. Cet environnement a été très rapidement disponible sur Internet pour une large communauté d'utilisateurs. Son interface de programmation était simple à assimiler. PVM s'est d'ailleurs révélé être aussi un excellent outil pour l'enseignement et l'initiation à la programmation parallèle. Plusieurs constructeurs ont fourni des versions optimisées de PVM sur leurs machines parallèles. Des réunions régulières d'utilisateurs se sont tenues pour des échanges sur les différentes expériences et applications développées. Sa principale utilisation concerne évidemment le calcul scientifique. Même si l'expression n'était pas encore tout à fait à la mode à cette époque, PVM pouvait être considéré dans le domaine du parallélisme comme un des premiers outils *middleware*.

Le début des années 90 a peut-être aussi marqué une réorientation des grands projets de recherche dans le domaine des systèmes d'exploitation. Le projet Chorus [117] développé par l'INRIA a donné naissance à une *start-up* qui, quelques années plus tard, a finalement été rachetée par Sun Microsystems. Cela montre combien il est difficile d'aller jusqu'au bout d'une démarche originale dans le domaine des systèmes à côté de grands groupes industriels. On peut cependant espérer que les idées originales issues de ces projets soient intégrées à terme dans les systèmes d'exploitation commerciaux. D'autres projets de recherche en système, comme Amoeba [111], se sont terminés en 1995. Une des difficultés de la recherche en système est qu'elle exige des lourds développements logiciels qui sont fortement dépendants des architectures matérielles (processeurs, interfaces réseaux) et qu'il est difficile de suivre les évolutions technologiques. En France, nous avons aussi constaté ces dernières années qu'il était particulièrement difficile de recruter des étudiants motivés par la recherche en système. D'autres domaines sont beaucoup plus attractifs (bases de données, multimédia, réseaux) en vue d'une intégration dans la vie professionnelle.

Depuis trois ou quatre années, les systèmes d'exploitation *freeware* (Linux, FreeBSD) connaissent un grand succès. Dans le monde des PC sous Unix, ces systèmes sont devenus sans conteste les leaders. Beaucoup de travaux des équipes de recherche en système s'appuient aujourd'hui sur ces systèmes pour les développements de pilotes de communication [108, 118] de nouveaux gestionnaires mémoire [25, 29]. Une des tendances de la communauté de la recherche en système est de se concentrer sur des aspects *middleware* [121] pour applications réparties. Les technologies de référence sont aujourd'hui celles imposées par les grands groupes industriels (CORBA [67], DCOM [52]). Des équipes s'intéressent par exemple au problème de la configuration [12] et au déploiement des applications réparties.

C'est donc dans le domaine du *middleware* pour le parallélisme que nos contributions se situent et plus précisément dans celui des supports d'exécution à base de processus légers. Alors que les environnements comme PVM ou bien MPI s'appuyaient sur des technologies système mûres, notre objectif a été d'élaborer de nouveaux environnements construits au-dessus de technologies nouvelles (threads, réseaux à haut débit) en ayant comme souci de fournir aux utilisateurs un haut niveau de performance. Les différentes réalisations logicielles ont été principalement effectuées sur des architectures distribuées de type grappes. Nous avons suivi deux approches dans la réalisation de ces environnements. Dans le contexte du projet PVC, nous avons utilisé les outils et bibliothèques qui nous étaient fournis sur la machine parallèle. Nous nous sommes ensuite vite rendus compte des limitations et des difficultés à porter efficacement nos supports vers d'autres architectures. Pour le projet ESPACE et la plate-forme PM², notre approche était résolument tournée vers un développement de bibliothèques spécifiques, portables et efficaces. L'intérêt d'une telle approche est aussi de réussir à marier plus harmonieusement communication et multithreading qui n'étaient pas vraiment faits au départ pour fonctionner ensemble. Nous nous sommes également intéressés aux problèmes de régulation de charge, en ayant à ce niveau une démarche dirigée par les applications. Une de nos préoccupations a toujours été de valider et de faire expérimenter nos réalisations logicielles par d'autres équipes de chercheurs françaises et étrangères. Le prochain chapitre présentera plus longuement nos travaux en les positionnant précisément par rapport aux travaux similaires dans cette communauté.

Chapitre 3

Supports d'exécution multithreads

Le développement du parallélisme et des applications s'est pendant plusieurs années principalement appuyé sur des environnements de programmation comme PVM (Parallel Virtual Machine) et plus récemment sur MPI (Message Passing Interface). Ces environnements sont disponibles sur un large éventail d'architectures et utilisés par une large communauté scientifique. Une des caractéristiques majeures de ces environnements est de s'articuler autour d'unités d'exécution calquées sur la notion de processus UNIX comme unité d'exécution. Il est souvent dit que ces environnements sont à **gros grain** par le fait que les processus consomment d'importantes ressources mémoire et système. Une des conséquences de cette granularité est que le degré de parallélisme des applications (nombre d'activités concurrentes) est limité et qu'il est difficile voire impossible de mettre en œuvre efficacement des applications avec plusieurs centaines de tâches. D'autre part, avec ce type d'environnement. Le programmeur dispose de très peu de fonctionnalités pour ordonnancer et réguler son application. Pour toutes ces raisons, nous nous sommes décidés à étudier des environnements basés sur un grain plus fin (processus légers). De tels supports exécutifs offrent des possibilités nouvelles en termes de virtualisation de l'architecture et une meilleure gestion des ressources, par exemple au niveau du recouvrement des communications par du calcul.

Le début des années 1990 a été marqué par l'émergence des méthodologies de programmation à objets, l'apparition de nouveaux concepts dans les systèmes autour des processus légers et de technologies pour les machines parallèles (processeur Transputers). Comme il a été dit dans les paragraphes précédents, c'est aussi à cette époque que les activités autour des environnements de programmation parallèle ont véritablement décollé. C'est donc dans ce contexte que nous nous sommes intéressés à définir des modèles de programmation et d'exécution basés sur la notion de processus légers (*thread*). Cette thématique scientifique a été menée dans le cadre de deux projets où ont été étudiées deux approches pour ce nouveau type d'environnement.

- Une première approche (1990 à 1995), que nous qualifierons de processus légers communicants, avait été proposée dans le cadre du projet PVC-BOX et avait été spécialement choisie pour le support de la compilation du langage à objets actifs BOX. Nous décrivons l'environnement PVC dans ce chapitre, les principaux choix de conception concernant le modèle de programmation, son interface et les choix d'implémentation.
- Une seconde approche, basée sur le concept d'appel de procédure à distance (RPC), a ensuite été étudiée pour la plate-forme PM². L'intérêt du modèle RPC avait été par ailleurs largement argumenté pour la conception d'applications réparties comme une alternative au traditionnel modèle client-serveur. Dans le contexte du parallélisme et de la conception d'applications, nous pensons que cet opérateur répond aussi aux besoins des programmeurs et de leurs applications.

L'implémentation des RPC dans un environnement à base de processus légers est efficace et n'exige pas d'avoir un mécanisme de désignation et de localisation globale des activités de calcul. Cette stratégie d'implémentation ouvre la possibilité de concevoir des mécanismes sophistiqués, comme par exemple la migration des activités qui nous intéressait pour la mise en place de stratégies de régulation de charge. L'environnement PM² sera présenté dans ce chapitre et nous comparerons les choix de conception avec ceux de la plate-forme PVC.

Ces travaux s'inscrivent dans la même lignée que ceux autour de l'environnement Nexus développé au laboratoire Argonne. À l'instar de PVC, la vocation première de Nexus [60] était de servir de support pour des compilateurs ou des bibliothèques, plutôt que d'être utilisé directement dans les codes applicatifs. Aujourd'hui, une des principales utilisations de Nexus est de constituer la couche de communication du projet Globus qui est destiné à fournir des services logiciels pour le développement d'applications dans un contexte de métacomputing. Le modèle d'exécution de Nexus est l'appel de procédure à distance (RSR), mais c'est principalement les capacités de Nexus à gérer plusieurs protocoles [56] que nous retiendrons. Au niveau de la communauté française, les travaux les plus proches des nôtres sont ceux menés par le projet INRIA APACHE à Grenoble qui vise à définir un environnement de programmation et d'exécution pour applications irrégulières. Ce projet s'appuie également sur un noyau exécutif basé sur des processus légers **Athapascan-0** [21]. Les différences notables avec PM² se situent principalement au niveau de l'interface de programmation et des choix méthodologiques de développement. Athapascan-0 est implémenté au dessus des composants standards qu'on trouve sur les systèmes et architectures parallèles (Threads POSIX, couche de communication MPI) alors que l'approche que nous avons retenue pour PM² est radicalement différente puisque nous avons choisi de développer nos propres composants, pour mieux répondre aux exigences et spécificités de ce type d'environnement concernant les communications et les processus légers. Le lecteur trouvera dans les annexes un article [20] rédigé à l'occasion d'une école sur le parallélisme présentant les supports d'exécution multithreads Athapascan et PM².

3.1 Deux projets de recherche

La thématique scientifique présentée dans l'introduction de ce chapitre a donc été étudiée dans deux projets. Le premier, PVC-BOX, nous a fait explorer deux voies : un support d'exécution à base de processus légers d'un côté, et la définition et l'implémentation d'un langage de programmation à objets distribués de l'autre.

Pour le second projet, ESPACE, nous avons poursuivi dans cette même thématique avec deux axes : 1) le support exécutif qu'on voulait cette fois-ci plus portable que PVC et plus ouvert à un plus large éventail d'applications ; 2) les applications comprenant comme pour PVC-BOX des compilateurs (HPF, Java, C++//) mais aussi des applications de calcul scientifique (optimisation combinatoire, algèbre linéaire).

3.1.1 PVC-BOX : 1990-1995

Notre motivation principale était d'étudier les méthodologies et les technologies orientées objets pour le parallélisme et ses applications. Les premières réflexions autour de PVC ont débuté en 1990 par la notion de Processeur Virtuel de Classe qui peut se définir comme un serveur encapsulant une classe d'objets et ses instances. Cette proposition se basait sur le principe de localité du code et des données et d'une fragmentation des objets dirigée par les liens d'héritage. Cette architecture logicielle a servi ensuite de support pour la conception et l'exécution de programmes développés

dans des langages à objets parallèles. Très rapidement, deux types d'activités ont été menés dans le groupe : une première activité concernant la définition et la réalisation du support exécutif (PVC), une seconde activité s'attachant à définir le langage de programmation [69] à objets distribués, à l'architecture de son compilateur et à discuter de l'interface avec le support d'exécution.

J'ai été dans ce projet responsable des activités autour du support exécutif PVC. Dominique Lazure [84] a pendant son mémoire de DEA évalué l'architecture du processeur Transputer, d'une machine parallèle Parsytec à 16 processeurs et son système d'exploitation Helios, un dérivé du système Unix. Les premiers développements ont ensuite été réalisés par Luc Courtrai qui a proposé dans sa thèse [33] un modèle exécutif basé sur les processus légers que nous appelons : *Composants Actifs de Communication* (CAC). Les premières versions de PVC tournaient exclusivement sur cette machine parallèle. L'avenir de la technologie du Transputer étant rapidement devenu incertain, nous nous sommes ensuite intéressés à porter PVC sur des architectures plus classiques comme par exemple des réseaux de stations de travail.

Les trois objectifs pour le support PVC étaient les suivants.

L'indépendance vis-à-vis du système d'exploitation. Un des premiers objectifs du support d'exécution PVC est de cacher à l'utilisateur les spécificités du système d'exploitation (Helios) et de l'architecture (Parsytec). Si cette indépendance concernait principalement l'utilisateur, il s'est ensuite révélé difficile de porter PVC sur d'autres architectures et systèmes.

La virtualisation de l'architecture. Nous souhaitions un environnement où l'architecture, c'est-à-dire les processeurs, la mémoire et le réseau de communication, soit complètement cachée aux programmeurs pour permettre qu'un même programme puisse s'exécuter sans aucune modification sur deux configurations différentes¹ de la même machine.

Répartition statique du code et dynamique des activités. Enfin, après la phase de conception de l'application parallèle, nous avons proposé des outils [72] pour permettre de répartir statiquement le code sur les différents processeurs de la machine. À cette époque, les processeurs disposaient de peu de mémoire et nous avons choisi de ne pas répliquer le code sur l'ensemble des processeurs. Les activités étaient ensuite réparties dynamiquement sur les processeurs capables de les accueillir.

D'autres travaux ont également été menés autour de PVC. Jean-François Roos a défini dans sa thèse une architecture pour la récolte de traces d'exécution et les mécanismes de base pour l'aide à la mise au point des programmes dans PVC [113]. L'article² [115] présente un mécanisme efficace d'enregistrement des événements de programmes PVC. Un mécanisme de réexécution déterministe des programmes parallèles a ensuite été développé pour les programmes orientés objets BOX[113].

Cédric Dumoulin s'est intéressé à définir des techniques de ramassage de miettes [49, 50] pour faciliter et optimiser l'utilisation des ressources mémoire. Les travaux de Fred Hémerly [72] concernaient des mécanismes de régulation de charge et, pour aborder ce problème difficile, des mécanismes de simulation avec un langage synthétique de description d'applications parallèles ont été proposés.

3.1.2 ESPACE : 1995-1998

Le projet ESPACE³ a débuté en 1995 avec la thèse de Raymond Namyst [96] sur le support PM². Alors que pour le projet PVC-BOX notre cible de développement était initialement constituée d'une

¹De processeurs.

²Cet article accompagne ce document.

³Execution Support for Parallel Applications in high-performance Computing Environments.

machine parallèle à base de Transputers, nous nous sommes fixés comme principal objectif que PM² soit un environnement portable sur les architectures et systèmes les plus couramment rencontrés. Alors qu'habituellement la portabilité ne s'obtient qu'au prix de sacrifices importants sur l'efficacité, un des principaux défis pour l'environnement PM² était de montrer qu'il était tout à fait possible de combiner portabilité et efficacité. Contrairement à la réalisation de l'environnement PVC qui s'appuyait complètement sur les composants de communication et gestion des activités du système Helios, l'approche retenue pour PM² était de concevoir nos propres composants de communication et de gestion des processus légers. À notre connaissance, PM² est le seul environnement multithread à s'appuyer complètement sur ses propres composants.

Une des différences majeures entre les environnements PVC et PM² se situe également au niveau du modèle de programmation. Alors que PVC s'articulait autour d'un modèle d'exécution basé sur des processus légers communicants, le modèle PM² peut se décrire comme une extension de l'appel de procédure à distance léger (LRPC). Il s'apparente au mécanisme RSR de l'environnement Nexus et à l'appel de procédure à distance qui existait dans la première version de l'environnement Athapascan-0a[30]. Ce choix a été motivé par le fait que le modèle procédural RPC est très expressif pour la construction d'applications parallèles. Il est d'ailleurs facile de montrer que les deux modèles sont équivalents puisqu'il est possible de construire le RPC au-dessus de l'envoi de message et vice-versa. Les implémentations du modèle procédural montrent également qu'on tire pleinement profit des capacités et des performances des réseaux de communication en optimisant l'utilisation de la mémoire (plus de stockage intermédiaire des messages) et en évitant également un nommage global des activités. Nous souhaitions également fournir aux programmeurs des opérateurs leur permettant de définir des modules de régulation de charge. C'est ainsi que nous avons proposé un opérateur de migration permettant à un processus léger d'être migré pendant son exécution vers un autre processeur. Les opérateurs de migration et d'appel de procédure à distance (RPC) sont très facilement associables alors que la migration et l'envoi de messages le sont beaucoup moins et que des problèmes se posent en particulier au niveau des redirections de messages.

Enfin, notre volonté était aussi de faire utiliser la plate-forme PM² par d'autres groupes et équipes de recherche. Pour cela, l'interface de programmation a été spécialement conçue pour être simple et compréhensible par les utilisateurs. Un effort assez important, pas toujours visible dans un rapport, a été mené aussi pour assister les utilisateurs dans l'installation, les tests et même aussi parfois dans les développements et mises au point de leurs programmes.

La plate-forme PM² a également été utilisée pendant la thèse d'Yves Denneulin [42] qui s'est intéressé à définir des politiques d'ordonnancement et de régulation génériques adaptées aux différentes spécificités des applications parallèles irrégulières. Dans la même lignée que Nexus, Benoît Planquelle évalue les extensions nécessaires à PM² [103, 104] pour être utilisé dans un contexte du métacomputing. Cette thèse devrait se terminer en septembre 2000.

3.2 Modèles d'exécution

Deux modèles d'exécution bien distincts ont été étudiés dans les projets PVC et PM². Les composants actifs de communication (CAC) pour le projet PVC, l'appel de procédure à distance léger (LRPC) et la migration de processus légers pour PM². Nous allons maintenant les décrire.

3.2.1 Composants Actifs de Communication

Le Composant Actif de Communication (CAC) est un objet actif et autonome permettant de construire des applications parallèles. C'est une entité de programmation comprenant une partie traitement (processus), une partie donnée (mémoire locale) et une interface de communication (boîte aux lettres). Cette structure de CAC est proche de celle des acteurs proposée par G. Agha [2].

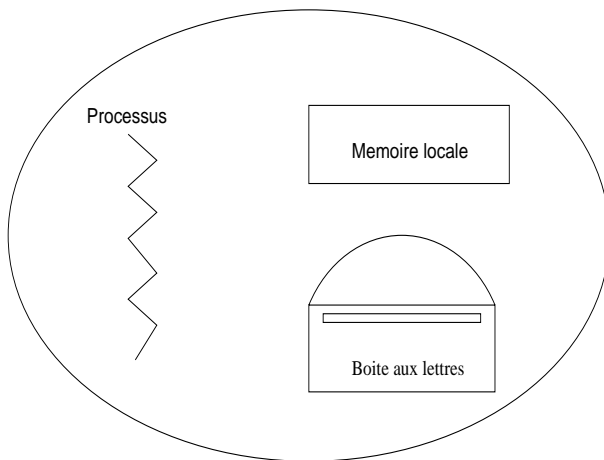


FIG. 3.1: Structure du CAC.

Processus. Chaque composant s'articule autour d'une activité autonome exécutant un ensemble d'instructions que nous appelons le comportement du composant. Ce comportement est défini par le programmeur au moyen d'une fonction contenant le code exécuté par le processus. Cette fonction permet au CAC d'effectuer des calculs et accès en lecture-écriture dans la mémoire locale et communique avec les autres composants.

Mémoire locale. C'est la zone de mémoire gérée dynamiquement par le composant dans lequel il est le seul à pouvoir lire ou écrire. Cette mémoire locale est allouée dans l'espace du processeur où s'exécute le composant. La taille de cette mémoire locale peut être modifiée (étendue ou restreinte) à tout moment de l'exécution du composant.

Boîte aux lettres. À chaque composant est associée une boîte aux lettres dans laquelle sont stockés tous les messages destinés au composant. Les communications inter-composants sont asynchrones et s'effectuent par le dépôt du message dans la boîte aux lettres du composant destinataire. Une boîte aux lettres peut contenir un nombre quelconque de messages, la seule contrainte étant la mémoire disponible sur le processeur. Le processus du composant extrait séquentiellement chacun des messages de sa boîte pour le traiter. Le processus ordonnance à son gré le traitement des messages. La boîte est mise en place dès la création du composant et son nom unique identifie le composant dans le système.

La programmation dans l'environnement CAC s'effectue dans le langage C et la manipulation des composants actifs de communication est réalisée au travers d'une interface de programmation et de sa bibliothèque associée.

Désignation. L'environnement CAC requiert la désignation de deux entités distinctes : les comportements pour la création des CAC et les CAC eux-mêmes pour que les communications entre composants puissent avoir lieu.

Comportements. La création d'un CAC nécessite la désignation d'un comportement. La création sur un nœud quelconque impose une désignation globale des comportements sur l'ensemble du système. La création doit respecter une contrainte, dite de localité : le code du comportement doit résider sur le nœud de création.

Composants. La primitive de création d'un composant renvoie un nom global au système pour permettre la communication entre CAC. Ce nom est celui de la boîte aux lettres du composant créé. La primitive d'envoi de messages laisse le message au système qui est chargé de le déposer dans la boîte aux lettres du CAC destinataire : la connaissance de la localisation du CAC destinataire n'est pas nécessaire.

3.2.2 Appel de procédure à distance léger

L'appel de procédure à distance est un concept qui a été élaboré dans le contexte des systèmes répartis pour rendre transparents les mécanismes de bas niveau impliqués dans une communication de type client-serveur. L'idée est de cacher derrière un simple appel de procédure les détails techniques liés aux communications et de fournir au programmeur un opérateur de haut niveau. En effet, les opérations de type *send* et *receive* sont d'un point de vue sémantique d'un niveau assez bas qu'on peut comparer aux opérations d'entrées-sorties (*read*, *write*) des systèmes. L'appel de procédure à distance présente l'avantage d'être d'un niveau plus élevé et d'être assez proche de l'appel procédural de la programmation séquentielle. Le déroulement d'un appel de procédure à distance se déroule de la manière suivante. L'activité *cliente* effectue un appel local « classique » à une procédure spéciale. Cette procédure encapsule des mécanismes capables de transformer cet appel local en un message qui sera émis vers une activité *serveur* apte à exécuter la fonction. Une fois cette fonction exécutée, le résultat est retourné vers la procédure spéciale qui, après avoir extrait les données reçues, retourne le résultat en utilisant les conventions de passage de paramètres de l'*appelant*.

Dans PM², le mécanisme utilisé est un mécanisme d'appel de procédure à distance **léger** (encore appelé *LRPC* pour *Lightweight Remote Procedure Call*) et diffère assez sensiblement du schéma précédent. La différence notable est qu'on se situe dans un environnement de processus légers et que l'exécution d'un LRPC déclenche la création d'un nouveau processus léger sur le nœud distant pour prendre en charge la fonction (figure 3.2).

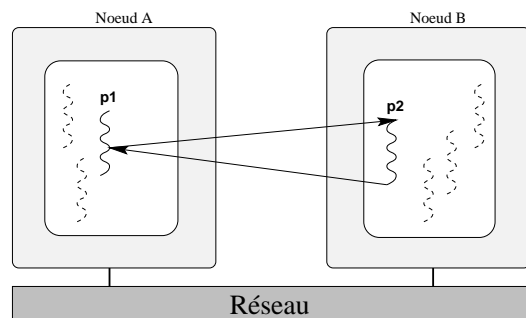


FIG. 3.2: Le LRPC de PM².

Lorsqu'un processus léger (p_1 sur la figure) effectue un appel de procédure à distance léger, il fournit principalement un nom de fonction à exécuter (on parle de *service*), des arguments et un emplacement mémoire pour le stockage des résultats. De manière interne, ces paramètres sont

alors *empaquetés* dans un message, puis celui-ci est envoyé vers le nœud cible. Une fois le message reçu par le nœud, le service à exécuter est identifié et la création d'un nouveau processus léger (p_2 sur la figure) est déclenchée. Ce processus extrait (*désempaquetage*) les paramètres du message et commence l'exécution du service. Lorsque l'exécution est terminée, le résultat est *empaqueté* dans un deuxième message, celui-ci est envoyé vers le nœud source et le processus (p_2) disparaît. Une fois le message reçu par le nœud source, le résultat en est extrait et est rangé à l'emplacement mémoire spécifié lors de l'appel.

3.2.3 Migration de processus légers

Une des spécificités du support d'exécution PM² est de fournir des fonctionnalités pour permettre à une application⁴ d'effectuer la migration d'un processus léger (ou d'un groupe de processus légers) d'un *nœud* à un autre pendant son exécution. Les possibilités qu'offrent un tel opérateur sont multiples et concernent évidemment la mise en place de mécanismes de régulation de charge. Lorsqu'un processeur n'a plus de threads de calcul à exécuter, il lui est possible d'aller en voler à d'autres processeurs qui en auraient trop. Un autre cas où l'utilisation de la migration se justifie serait celui d'une application manipulant de gros volumes de données qu'il pourrait être fort coûteux de répliquer. Il est tout à fait envisageable d'imaginer des stratégies où les processus légers migrent pour accéder aux données qu'ils accèdent. Une des limitations au mécanisme de migration PM² est qu'il ne s'applique qu'au cas où les deux nœuds sont homogènes au niveau du processeur et même du système d'exploitation.

Du point de vue de l'utilisateur, cette opération est très simple à réaliser puisqu'un simple appel à une primitive de la bibliothèque PM² provoque l'arrêt instantané de l'exécution du processus désigné, son transfert sur le nœud destinataire et la reprise de son exécution au point où elle avait été interrompue (figure 3.3).

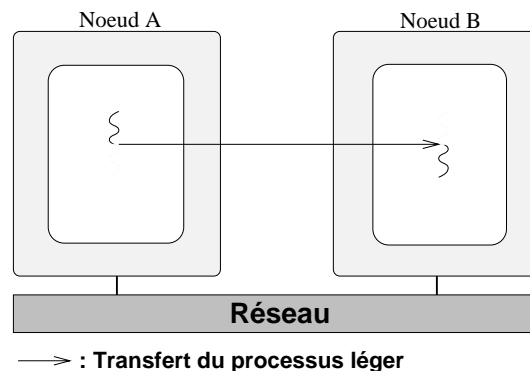


FIG. 3.3: Migration PM² : Changer le nœud d'exécution d'un processus léger.

Le déroulement de l'opération de migration est transparent pour le processus léger déplacé et peut survenir de façon asynchrone. Le processus migré n'est donc pas tenu d'exécuter de quelconques traitements périodiques destinés à sauver son état ou à effectuer son « auto-migration ». Ce dernier point est important car, contrairement à ce qui se passe dans d'autres systèmes [88], la migration d'un processus peut réellement être initiée par un autre processus (par exemple un processus chargé d'équilibrer la charge entre les nœuds) et ne nécessite pas une coopération du processus migré.

⁴Par exemple un régulateur de charge, qui est une application comme une autre du point de vue d'un support exécutif.

De par sa nature, la migration d'un processus léger provoque uniquement le transfert de son contexte local (pile d'exécution, variables locales). Par conséquent, après sa migration, le processus léger se trouve plongé dans un nouvel environnement global (variables globales du processus, variables « système » telles que les fichiers ouverts, etc.). Cette caractéristique est toutefois sans conséquence pour la plupart des applications développées avec PM², car le modèle de programmation ne préconise ni l'accès « direct » aux variables globales d'un processus, ni l'accès (sauf sous contrôle) aux primitives de la bibliothèque C standard. Le cas échéant, une application peut faire usage de primitives permettant de *protéger* temporairement l'exécution d'un processus léger contre son éventuelle migration intempestive.

Dans ses premières versions, les différentes utilisations montraient que les mécanismes de migration étaient difficiles à utiliser, essentiellement à cause des problèmes inhérents aux pointeurs que le programmeur devait explicitement enregistrer. Un des problèmes les plus délicats provenait du fait qu'après la migration, l'adresse de la pile d'exécution du thread pouvait être différente et il était alors nécessaire de réajuster les pointeurs sur des objets de la pile. Une des améliorations les plus récentes dans les mécanismes de migration de PM² a été apportée par Gabriel Antoniu qui pendant son mémoire de DEA [6, 8] a défini un mécanisme de gestion de la mémoire iso-adresse. Notre objectif était de faciliter l'usage et l'implémentation de cette opération de migration. La propriété fondamentale de l'iso-adressage est de garantir qu'un processus léger et sa pile d'exécution retrouveront la même adresse virtuelle après une opération de migration. Cela concerne d'abord la pile du processus léger, mais aussi l'ensemble des données dynamiques privées qui ont été allouées.

3.3 Interfaces de programmation

Dans cette section, nous détaillons les principales fonctionnalités de PVC et PM² au travers de leurs interfaces de programmation. Nous nous intéresserons plus précisément à l'expression du parallélisme, aux opérations de communications et aux fonctions d'accès à la mémoire. Le lecteur trouvera dans les annexes le code d'une même application développée avec PVC (annexe A) et PM² (annexe B).

3.3.1 PVC

Le programmeur décrit une application par un ensemble de comportements modélisant les activités parallèles de son application. La programmation des fonctions comportementales est réalisée en langage C. Toutes les structures de contrôle du langage sont ainsi réutilisables. Les primitives de l'environnement PVC et de manipulation des CAC sont accessibles à travers une bibliothèque de fonctions. La bibliothèque comprend les primitives d'accès aux services du système et les primitives de communication entre CAC. Des primitives de gestion de boîtes aux lettres sont également proposées au programmeur autorisant un CAC à posséder plusieurs boîtes aux lettres locales, ce qui facilite la modélisation et la programmation des applications parallèles. Nous ne donnons qu'un synthétique aperçu des primitives de programmation CAC. Le lecteur intéressé pourra se référer au manuel de programmation [114].

Les composants Dans l'environnement de programmation CAC, les références sur les CAC sont typées (type `Component`).

*Component NewComponent (Component me, Behavior b, int processor, int size, char *arg)*

L'appel à *NewComponent* crée un nouveau CAC de comportement **b**. Le nom de la fonction comportementale (**b**) doit être passé en paramètre à la primitive ainsi que ses éventuels arguments (**arg**). La primitive *NewComponent()* est synchrone et retourne le nom de la boîte aux lettres du composant créé. Le programmeur peut choisir le numéro du processeur où le composant s'exécutera ou éventuellement laisser ce choix à l'environnement. Cette primitive permet la création de CAC à distance. Elle est utilisable de la même manière, que **b** soit un comportement implanté sur le nœud local ou sur un nœud distant. C'est donc cette primitive qui permet l'exécution répartie de nos applications. À la fin de son exécution, le composant exécute la fonction *EndComponent* pour libérer les ressources qui lui sont allouées. La fonction *KillComponent* permet à un composant d'arrêter l'exécution d'un autre composant. Les primitives de gestion de composant sont résumées dans le tableau 3.1.

<i>NewComponent</i>	Création d'un CAC
<i>EndComponent</i>	Terminaison d'un CAC
<i>KillComponent</i>	Destruction d'un CAC

TAB. 3.1: Primitives de gestion de composant.

Communications inter-composants Les communications entre les composants se font par dépôt et retrait de messages dans les boîtes aux lettres des CAC. Les communications entre composants sont asynchrones. Les messages contenus dans la boîte d'un composant sont stockés dans l'ordre de leur arrivée. Seul le composant propriétaire de la boîte aux lettres peut consulter et extraire les messages. Le nom d'un CAC est retourné au CAC créateur par la primitive *NewComponent()* et permet d'identifier sa boîte aux lettres. Une fois créée, la référence d'un CAC peut être passée en argument d'un message. Seules ces deux opérations permettent d'obtenir une référence à un CAC, ce qui autorisera l'envoi de message vers celui-ci. Les messages entre composants actifs de communication n'ont pas de structure particulière. Le mécanisme de communication transmet directement un bloc de données. Le tableau 3.2 donne une liste des primitives de communication de haut niveau. Elles permettent de construire, envoyer et recevoir des messages.

<i>NewMessage</i>	Création d'un message
<i>FreeMessage</i>	Libération d'un message
<i>AddArg</i>	Ajout d'un argument dans le message
<i>GetFirstArg</i>	Récupération d'un pointeur sur le premier argument
<i>GetNextArg</i>	Récupération d'un pointeur sur l'argument suivant
<i>Send</i>	Construction et émission d'un message
<i>GetMessage</i>	Lecture du premier message

TAB. 3.2: Primitives de communication.

La mémoire La mémoire est gérée par le système lors de la création de composants actifs de communication. L'espace mémoire de l'environnement local du composant est créé dans la phase d'initialisation du composant. Le processus du composant peut ensuite modifier explicitement la taille de son environnement local. L'allocation mémoire et la libération s'effectuent par deux primitives. Ces opérations restent locales au nœud. Les primitives de gestion mémoire sont présentées par le tableau 3.3.

<code>void *new(int size)</code>	Allocation mémoire
<code>void dispose(void *p)</code>	Libération mémoire

TAB. 3.3: Primitives de gestion mémoire.

Commentaires

La description complète d’une interface de programmation peut s’avérer fastidieuse et nous avons choisi d’en donner juste un bref aperçu. Pour aider à la compréhension, un exemple synthétique d’application figure dans les annexes A et B

Comme l’objectif majeur de PVC était de servir de support au langage BOX, peu d’efforts ont été faits pour rendre cette interface conviviale et accessible à un utilisateur. Des environnements comme Nexus ou Athapascan-0, qui n’ont pas plus pour vocation d’être utilisés directement par les programmeurs d’applications, offrent également une interface assez complexe. Pour essayer d’atteindre une communauté d’utilisateurs plus large, nous avons proposé pour PM² une interface qui nous paraît plus simple à appréhender.

Concernant les interfaces, on peut s’étonner qu’aucune standardisation des interfaces des supports d’exécution à base de processus légers distribués n’ait été définie, alors que pour les environnements à gros grain MPI est devenu le standard incontestable et qu’OpenMP est en train de devenir le nouveau standard pour la programmation parallèle sur des architectures SMP. Deux explications sont possibles : 1) les technologies logicielles ne sont pas encore assez matures et de nombreux problèmes subsistent ; 2) la demande des utilisateurs n’est pas encore assez forte pour créer un groupe de pression.

3.3.2 PM²

Dans cette section, nous abordons l’interface de programmation PM² en détaillant plus précisément le RPC, l’opération de migration et les fonctions d’accès à la mémoire. Nous verrons en particulier les différentes formes de RPC et le principe de fonctionnement de la migration.

Appel de procédure à distance L’utilisateur du support PM² décrit son application parallèle autour d’un ensemble de procédures qui seront donc activables à distance. Raymond Namyst [96] a proposé trois variantes de l’appel de procédure à distance léger (LRPC) dans l’environnement PM² : synchrone, asynchrone et à attente différée.

<code>LRPC (proc, f, prio, stack_size, in, out)</code>	LRPC synchrone
<code>LRP_CALL (proc, f, prio, stack_size, in, out, wait)</code>	LRPC à attente différée
<code>ASYNC_LRPC (proc, f, prio, stack_size, in, out)</code>	LRPC asynchrone

TAB. 3.4: Interface pour les LRPC.

Appels synchrones. L’appel de procédure à distance léger **synchrone** est le mécanisme dont la sémantique est la plus proche de l’appel de procédure à distance classique. Lorsqu’un tel appel est effectué, le processus appelant est immédiatement *bloqué* par le système (*i.e.* par PM²) jusqu’à la disponibilité du résultat.

Appels à attente différée. L’appel de procédure à distance léger **à attente différée** constitue un opérateur de décomposition parallèle majeur dans l’environnement PM². Son fonctionnement

est comparable à celui d'un appel *synchrone* pour lequel les opérations d'envoi des paramètres et de réception du résultat seraient séparées. Son utilisation s'effectue donc en deux temps.

Dans un premier temps, le processus appelant effectue l'appel en spécifiant les mêmes paramètres que dans le cas d'un appel synchrone (nom de service, arguments et adresse de stockage du résultat) et en y ajoutant une référence sur une variable d'un type particulier qui lui servira de *clé* par la suite. Cette étape donne lieu à la création d'un nouveau processus léger (pour exécuter le service), mais ne « bloque » pas le processus appelant qui peut donc librement poursuivre son exécution.

Dans un second temps, lorsque le processus appelant désire accéder au résultat de l'appel, celui-ci doit effectuer une opération d'attente en spécifiant la clé qu'il a obtenue lors de l'appel correspondant. Cela conduit alors au blocage de ce processus jusqu'à disponibilité du résultat.

Appels asynchrones. La dernière forme d'appel de procédure à distance léger proposée par l'environnement PM² est dite « **asynchrone** », car elle offre le moyen de créer un nouveau flot d'exécution indépendant du processus appelant. Ce mécanisme, qui s'apparente plutôt à un « déclenchement de traitement à distance », diffère des deux précédents par le fait qu'il ne comporte pas de phase de retour de résultat. En ce sens, il est très similaire au mécanisme de RSR (*Remote Service Request*) proposé par l'environnement Nexus [60].

Migration L'opération de migration est principalement conçue pour des modules de régulation de charge ou éventuellement pour des compilateurs qui auraient une connaissance de la localisation des données et des activités. Le déroulement des opérations mises en jeu lors d'une migration de processus légers est beaucoup plus simple que dans le cas d'un appel de procédure à distance léger. Cette simplicité de mise en œuvre est due à la souplesse d'utilisation des fonctionnalités d'*hibernation de processus* fournies par la bibliothèque de processus légers Marcel sur lequel PM² est implémenté. Une opération de migration de processus légers se déroule en trois étapes.

1. Dans un premier temps, l'appel à la primitive `pm2_freeze` *gèle* le processus UNIX, c'est-à-dire stoppe complètement le déroulement des activités (processus légers) autres que le processus appelant la primitive, qui devient alors le seul processus actif sur le nœud.
2. Dans un second temps, l'appel à la primitive `pm2_threads_list` permet d'obtenir la liste de tous les processus légers *migrables* sur le nœud. Comme le système est gelé, le processus appelant peut alors calmement parcourir cette liste pour y sélectionner les processus (éventuellement aucun) qu'il décide de migrer.
3. Une fois ces processus choisis, dans un troisième et dernier temps, l'appel à la fonction `pm2_migrate` permet de déclencher le transfert des processus vers un nœud indiqué. Le processus UNIX courant est alors dégelé et les autres processus reprennent leur activité normale.

Les primitives `pm2_enable_migration` et `pm2_disable_migration` permettent à un processus léger de déclarer qu'il est (resp. n'est pas) candidat à la migration. C'est particulièrement utile lorsque le processus entre dans une section manipulant des ressources système locales comme des fichiers.

Gestion mémoire PM² propose deux opérations qui permettent aux processus légers d'allouer/libérer de la mémoire dans la zone iso-adresse. Les données allouées dans cette zone iso-adresse sont automatiquement transférées en cas de migration du processus léger. Le principe de l'iso-adresse est donc de garantir que ces données se retrouveront à la même adresse virtuelle après migration. Il

<code>pm2_freeze</code>	Gèle l'état du système
<code>pm2_thread_lists</code>	Liste des threads du nœud
<code>pm2_migrate (proc, thread)</code>	Migration simple
<code>pm2_migrate_group (proc, threads)</code>	Migration de groupe
<code>pm2_enable_migration</code>	Migration autorisée
<code>pm2_disable_migration</code>	Migration interdite

TAB. 3.5: Interface pour la migration.

est important de noter que le modèle PM² ne définit pas d'objets et de mémoire partagée. L'étude de ces aspects fait l'objet de la thèse de Gabriel Antoniu en partant de l'expérience issue du projet DOSMOS et de la thèse d'Olivier Reymann [112] qui avait porté l'environnement DOSMOS sur PM².

<code>pm2_isomalloc</code>	Allocation mémoire iso-adresse
<code>pm2_isofree</code>	Libération mémoire iso-adresse

TAB. 3.6: Allocation mémoire iso-adresse.

Commentaires

Un des objectifs que nous nous étions fixés pour cette interface est qu'elle soit simple à utiliser. Les différentes utilisations ont montré que le LRPC était assez facile à mettre en œuvre dans les applications. Par contre, l'utilisation de la migration a posé plusieurs problèmes aux utilisateurs, principalement à cause du fait que PM² n'offre pas de dispositif de partage d'objets. Comme nous l'avons montré au niveau de la section décrivant la gestion mémoire, nos efforts portent actuellement sur l'intégration d'un mécanisme de partage d'objets. Ces travaux débutent et nous avons depuis quelques mois des contacts avec l'équipe de Franck Mueller à Berlin qui est le concepteur de DSM-Threads [93].

Même si cette interface est simple à utiliser, il reste que ce type d'interface et l'environnement qui lui est associé sont des prototypes de recherche. L'absence de normalisation fait que les utilisateurs se tournent plus naturellement vers les standards du marché que sont MPI, OpenMP⁵ et même dans certains cas CORBA. Une stratégie, c'est en partie celle qu'a choisie l'équipe Nexus, est d'effectuer les développements nécessaires pour supporter MPI. MPICH-G [57] est ainsi une des premières implémentations de MPI qui soit effectivement interopérable à partir du support multi-protocole disponible dans Nexus. Cependant, l'inconvénient de cette stratégie est la non utilisation des fonctionnalités du multithreading disponibles dans Nexus. Une approche similaire pourrait aussi être envisagée avec l'interface OpenMP qui pourrait s'appuyer sur des supports exécutifs comme PM², en particulier si des mécanismes de mémoire partagée y étaient intégrés.

3.4 Réalisations des supports exécutifs

La présentation des implémentations de nos deux environnements (PVC et PM²) se fera en deux parties : une première section où je décrirai le composant de gestion du multithreading, une seconde

⁵<http://www.openmp.org>

section détaillant le composant de gestion des communications. Les approches d'implémentation pour PVC et PM² sont totalement opposées. Pour PVC, l'implémentation est directement liée à l'architecture et au système de la machine parallèle qui était disponible dans notre laboratoire. Pour PM², nous nous sommes efforcés de construire une implémentation portable et efficace sur un grand nombre de machines.

3.4.1 PVC

Gestion des activités Une première maquette du support a été implémentée sur une machine parallèle à base de Transputers, le MultiCluster II de Parsytec, équipée de 32 Transputers T800 sous le système d'exploitation distribué Helios. Sur cette architecture, un module est implémenté par une tâche Helios (processus lourd UNIX) et l'activité des CAC par un processus Helios (processus léger). Les processus Helios sont créés dans une tâche et partagent l'espace d'adressage de la tâche. La distribution des modules est réalisée par l'intermédiaire du Component Description Language (CDL), outil du système d'exploitation Helios.

Les premières implémentations de PVC s'appuyaient sur les processus légers disponibles sous le système Helios. Les fonctions de création de processus légers ont été utilisées pour implémenter les opérations de création des CAC. Les opérations de synchronisation (sémaphores) ont quant à elles été utilisées pour implémenter les opérations sur les boîtes aux lettres des CAC.

Cette implémentation des CAC a ensuite été portée sur réseau de stations de travail Sun sous le système d'exploitation SunOS 4.1, utilisant la bibliothèque `lwp` [34] et les communications par sockets. Le module est implémenté par un processus Unix et l'activité d'un CAC par un processus léger de la bibliothèque `lwp`. La distribution des modules est également réalisée par l'intermédiaire d'un outil de lancement d'applications (`cdl`) qui a été porté sous Unix : un démon sur chaque site est chargé de créer les processus Unix et les liens de communication (sockets TCP/IP).

Gestion des communications La réalisation des composants actifs de communication est développée au-dessus d'une gestion de boîtes aux lettres. Il est intéressant de laisser les primitives de manipulation de boîtes aux lettres accessibles au niveau de la programmation. Un composant peut ainsi créer plusieurs boîtes aux lettres locales. Chaque boîte aux lettres permet, par exemple, d'identifier les requêtes, sans gérer un mécanisme de nommage de requêtes et de retour de réponses. Ces structures passives sont locales à l'environnement d'un composant et sont en général temporaires. Le composant qui crée une boîte est le seul à pouvoir la consulter. Le nom de la boîte peut être transmis dans des messages aux autres composants qui pourront alors y déposer des messages par la primitive *SendMessage()*. Les primitives de gestion des boîtes sont présentées dans le tableau 3.7. À partir des primitives *NumberMessageInBox()*, *ReadFromBox()*, *DeleteMessageFromBox()*, *WaitOnBox()*, etc., l'utilisateur peut filtrer les messages reçus.

3.4.2 PM²

L'architecture générale de l'implémentation PM² repose classiquement sur un composant de gestion des threads et un autre composant pour les communications. Toute la difficulté consiste à réussir une intégration flexible et efficace des processus légers et des fonctions de communication. Habituellement, les environnements multithreads (Nexus, Athapascan) s'appuient sur les bibliothèques standard fournies dans les systèmes d'exploitation. La difficulté se situe ensuite au niveau du couplage de ces deux bibliothèques qui n'ont pas forcément été conçues pour coopérer. Notre

DeleteMessageFromBox	Destruction de message
ExtractBox	Extraction de message
FreeBox	Destruction de boîte aux lettres
NewBox	Création d'une nouvelle boîte
NumberMessageInBox	Nombre de messages
PutToBox	Émission d'un message
ReadFromBox	Lecture et copie d'un message
WaitOnBox	Attente d'un nouveau message

TAB. 3.7: Primitives de gestion de boîtes.

approche est radicalement différente et consiste à développer nos propres composants de gestion de threads et de communication. L'avantage d'une telle approche est que l'intégration est naturelle puisque ces deux bibliothèques ont été spécialement fabriquées pour coopérer. Le portage de ces deux bibliothèques est simple sur la plupart des systèmes d'exploitation et couches de communication. Les performances de l'environnement et de nos deux bibliothèques sont de tout premier plan et valident notre approche. Des mesures de performance figurent dans les articles accompagnant ce document en l'annexe D.

Gestion des processus légers : Marcel Comme nous l'avons vu dans la section présentant les fonctionnalités de PM², les exigences de l'environnement sont assez importantes sur les processus légers. Cela concerne par exemple un ordonnancement tout à fait spécifique et que l'on ne retrouve aujourd'hui dans aucune implémentation de processus légers. D'autre part, l'opérateur de migration PM² exige que l'environnement puisse inspecter et accéder aux structures internes de l'ordonnanceur des processus légers. À notre connaissance, aucune autre implémentation ne fournit de tels mécanismes. Tout cela explique nos motivations pour concevoir et développer une nouvelle bibliothèque de processus légers que nous avons appelée Marcel. Notre objectif était de montrer qu'il était possible de construire une bibliothèque de processus légers efficace dont les performances sont de tout premier plan. Nous avons également prouvé que cette bibliothèque était portable puisqu'elle est aujourd'hui disponible sur les architectures les plus courantes (sur des PC), mais aussi sur des architectures de machines parallèles. Cette bibliothèque a été réalisée par Raymond Namyst pendant sa thèse.

La caractéristique principale de Marcel est d'être une bibliothèque de processus légers en contexte utilisateur, c'est-à-dire que les appels au système d'exploitation sont limités et que l'ordonnanceur du système n'a pas connaissance de ces processus légers. Le standard aujourd'hui en matière de programmation des processus légers est POSIX et l'interface de programmation de Marcel respecte la majeure partie de ce standard. Cependant, pour faciliter l'implémentation de PM², un certain nombre d'opérations et d'extensions ont été définies pour permettre l'implémentation d'opérations telles que la migration.

Un effort tout particulier a été mis sur certaines primitives qui nous semblaient être importantes à optimiser. Les opérations de création de processus légers ou bien encore le changement de contexte ont fait l'objet d'une attention toute particulière dans leur réalisation. Nous avons en particulier constaté que l'allocation mémoire d'un processus léger représentait une part très significative⁶ du temps total d'une création de processus léger. Nous avons donc essayé de minimiser ces appels aux allocateurs mémoire, en utilisant par exemple des caches pour les piles, mais également en

⁶De l'ordre de 60%.

compactant la mémoire d'un processus léger à un seul emplacement contigu. L'ordonnanceur de processus légers garantit une durée constante du changement de contexte, quel que soit le nombre de processus légers de l'applications.

L'approche retenue pour l'ordonnement des processus légers dans Marcel est basé sur le principe d'équité qui veut que tous les processus légers puissent s'exécuter concurremment sans risque de famine tout en gardant la possibilité d'en privilégier certains sur d'autres. En programmant une application avec la bibliothèque Marcel, il est possible d'affecter un niveau de priorités aux processus légers. L'intervalle des priorités possibles est fixé entre 1 et 100. Sur une machine mono-processeur, l'ordonnanceur Marcel assure la propriété P suivante "Un thread x (qui a une priorité $pr(x)$) s'exécute $pr(x)/pr(y)$ plus vite qu'un thread y (dont la priorité est $pr(y)$)".

Cette politique d'ordonnement est facilement et efficacement implantée dans la bibliothèque Marcel. Elle est gérée suivant la méthode *round-robin*. Tous les processus légers prêts à être exécutés sont stockés dans une file unique. À chaque tour de tourniquet, le processus léger dispose d'un nombre de quanta de temps processeur proportionnel à sa priorité. Quand ce quota est épuisé, le processeur est affecté au processus suivant dans la liste. Classiquement, lorsque la fin de la liste est atteinte, un nouveau tour de liste commence.

L'ordonnanceur Marcel se caractérise donc par une prise en compte de tous les processus légers, sans risque de famine, et un partage inégal du temps processeur entre eux, qui est conditionné par leur valeur de priorité. Ces deux caractéristiques répondent aux besoins d'application où il est possible de définir une priorité aux activités parallèles. C'est par exemple le cas des applications d'optimisation combinatoire qui ont été étudiées par Yves Denneulin pendant sa thèse.

Les performances brutes obtenues avec Marcel sont de tout premier plan [97, 44] et ont été évidemment comparées avec les implémentations qu'on retrouve sur les différentes architectures. Il est important de souligner que la bibliothèque Marcel n'a pas une vocation aussi générale que les bibliothèques de processus légers qu'on retrouve sur le marché. Le domaine d'utilisation de Marcel concerne principalement le calcul scientifique s'appuyant sur des processus légers. Les performances de Marcel s'expliquent donc par les différentes optimisations qui ont été effectuées, mais aussi par le fait que certaines fonctionnalités un peu lourdes, comme la gestion des signaux, ne sont pas disponibles dans Marcel.

La bibliothèque Marcel est aujourd'hui opérationnelle sur six architectures de processeurs : Sparc, Pentium Intel, Alpha, Power PC et Mips. Le portage sur un nouveau processeur exige d'écrire quelques lignes d'assembleur pour accéder efficacement à certains registres du processeur comme le pointeur de pile ou bien encore le pointeur d'environnement (*frame pointer*). Nos efforts concernent actuellement une évolution majeure de Marcel pour supporter les architectures SMP. Il consiste à faire supporter par plusieurs threads de niveau système des threads utilisateurs. Ce travail a fait l'objet du stage [37] de Vincent Danjean à l'Université du New Hampshire.

Gestion des communications : Madeleine L'efficacité des communications est très certainement un des critères majeurs pour évaluer un support d'exécution parallèle. Si des bibliothèques comme PVM ou MPI ont connu un grand succès, c'est très certainement parce qu'elles étaient simples d'utilisation, mais aussi que leurs niveaux de performance, même s'ils n'étaient pas optimaux, étaient considérés comme largement acceptables par la communauté des utilisateurs. Un des objectifs que nous nous sommes fixés pour l'environnement PM² est d'offrir des mécanismes de communication efficaces. Pour atteindre cet objectif, deux problèmes doivent être étudiés : 1) celui de la gestion des messages, il faut en particulier éviter que les messages soient copiés ; 2) celui de la réactivité de l'environnement et plus précisément la manière dont les fins de communication vont être signalés à

l'environnement (scrutation ou interruption).

Pour supporter efficacement des applications mettant en œuvre des communications portant sur de gros volumes de données, il est important que les mécanismes fournis par l'environnement sous-jacent garantissent un nombre nul de copies intermédiaires des données transmises, du moins lorsque le programmeur le spécifie. Dans une application multithread, où les schémas de communication ne sont pas connus à l'avance, les informations concernant le stockage des données transmises sont donc le plus souvent détenues par l'émetteur du message. Lors de l'arrivée d'un message sur un site, son traitement s'effectue donc en trois étapes : 1) en premier lieu des informations concernant la nature du traitement à effectuer sont extraites du réseau ; 2) ensuite, une série d'allocations mémoire sont effectuées de manière à « préparer » une extraction sans copie ; 3) enfin, les données sont extraites du réseau et rangées directement au bon endroit. En utilisant une interface telle que MPI, qui permet la réception sans copie intermédiaire de types complexes, seuls deux « messages MPI » seront nécessaires. On remarque alors que pour une implantation de MPI au-dessus de TCP/IP, cela rajoute un message par rapport à une implantation directement sur TCP/IP (qui utilise un tamponnage implicite). En fait, le problème n'est pas spécifique à MPI : il concerne toutes les interfaces de communications de haut niveau existant aujourd'hui.

Les performances des applications multithreads sont souvent très sensibles au délai moyen de réaction de l'environnement vis-à-vis des événements liés au réseau (arrivée d'un message, canal à nouveau disponible en émission, etc.). Sur un système non temps-réel, une telle contrainte empêche l'utilisation de mécanismes d'appels bloquants, puisque le système n'offre aucune garantie sur le retard avec lequel un événement sera notifié à l'application. Une des solutions est en fait de recourir à une forme « active » de scrutation des événements (*polling*), de manière à maîtriser – et donc borner – le temps de réaction du système [60]. Un autre type de solution est d'utiliser un mécanisme d'interruption qui se lève lorsqu'une communication se termine. Une fonction *handler* est ensuite responsable du traitement associé à la fin de la communication. En théorie, c'est le mécanisme idéal pour obtenir une bonne réactivité. Malheureusement, les interfaces de communication en disposent rarement. De plus, il exige du noyau un pré-traitement pour sélectionner le processus concerné par l'interruption et génère un surcoût significatif. Ce mécanisme s'avère intéressant lorsque le volume de communication est faible par rapport aux calculs. De nombreuses études [83] ont été menées sur ce sujet et montrent que la solution devrait être basée sur des stratégies adaptatives.

Madeleine [17] est une interface de communication de niveau intermédiaire entre des environnements évolués (PVM, MPI, PM²) et des interfaces de communication comme TCP, BIP[107], VIA[31], SBP[118]). Elle n'a donc pas vocation à être utilisée directement par les programmeurs d'applications. Elle est structurée en deux modules : une couche de portabilité destinée à l'interface avec les protocoles réseaux sous-jacents et une couche fournissant des fonctionnalités d'emballage/déballage de données dans les messages. Le portage de Madeleine sur un protocole réseau donné nécessite uniquement d'adapter sept primitives de base listées dans la table 3.8. Deux d'entre elles sont responsables de l'établissement/extinction des connexions entre nœuds distants (*init* et *exit*). Les cinq autres sont destinées à la gestion des communications proprement dites.

L'envoi d'un message (*send*) est effectué en fournissant un nœud destinataire ainsi qu'une collection de fragments (données contiguës) en paramètres. Le premier de ces fragments a une sémantique particulière : il est supposé contenir des « données préliminaires » devant pouvoir être extraites avant les données proprement dites lors de la réception du message. Un message est donc constitué d'un fragment d'« en-tête » et de plusieurs fragments de données. La réception d'un message est une opération plus complexe et constitue précisément la spécificité de Madeleine. Elle s'effectue en quatre temps et débute toujours par une opération indiquant que l'on se tient prêt à recevoir un fragment

Fonction	Opération
<code>init(configSize, taskIDs)</code>	Installe les connexions
<code>exit()</code>	Termine les connexions
<code>send(dest, iovec, count)</code>	Envoie un msg au nœud 'dest'
<code>recv_header_post(func)</code>	Prêt à recevoir un en-tête
<code>recv_header_poll()</code>	Teste si l'en-tête est reçu
<code>recv_body_post(exped, iovec, count)</code>	Prêt à recevoir des données
<code>recv_body_poll(exped)</code>	Teste si les données sont reçues

TAB. 3.8: La couche de portabilité dans Madeleine.

d'en-tête (`recv_header_post`). Ensuite, la scrutation du réseau doit être explicitement effectuée par un appel à `recv_header_poll`. Dès qu'un en-tête pourra être extrait du réseau, la fonction *callback* précisée lors du `recv_header_post` sera appelée avec en paramètre l'adresse d'une zone mémoire contenant l'en-tête. C'est depuis cette fonction que pourront être effectuées les diverses allocations mémoire en fonction des informations contenues dans l'en-tête. Dans cette même fonction, il sera alors possible d'appeler `recv_body_post` en indiquant la liste des adresses auxquelles il faudra ranger les fragments constituant le corps du message.

Les primitives décrites ci-dessus sont complétées par des fonctionnalités permettant une construction aisée des différents fragments constituant un message. Principalement, il s'agit de primitives d'emballage/déballage des principaux types de données de base du langage C (`pack_int`, `pack_float`, etc.). De manière à permettre aux applications de contrôler la distribution des données dans les divers fragments, ces primitives requièrent toutes en premier argument un mode d'emballage/déballage qui peut prendre la valeur `IN_HEADER`, `IN_PLACE` ou `BY_COPY`. Le mode `IN_HEADER` assure que la donnée correspondante est placée dans le premier fragment, ce qui lui confère le statut de « donnée préliminaire ». Le mode `IN_PLACE` assure qu'aucune copie intermédiaire de la donnée ne sera effectuée par Madeleine. À l'opposé, le mode `BY_COPY` effectue une recopie de la donnée de façon à pouvoir modifier la donnée sans risque *a posteriori*.

```

UNPACK_REQ_STUB(PARAMETRES_OUT *data)
    unpack_int(IN_HEADER, &data->taille, 1);
    data->tampon = malloc(data->taille);
    unpack_byte(IN_PLACE, data->tampon, data->taille);
END_STUB

```

FIG. 3.4: Au cours du déballage des paramètres d'un appel de service, certaines données doivent être extraites immédiatement (*taille*) alors que la réception des autres est différée au moment où l'on connaîtra toutes les adresses de destination des fragments (*tampon*).

La figure 3.4 illustre l'utilisation typique de ces primitives dans une fonction appelée lors du déballage des paramètres d'une invocation de service. Madeleine et PM² sont aujourd'hui disponibles sur différents protocoles de communication : MPI, PVM, TCP, BIP, SBP, SHMEM et plus récemment sur VIA [18]. Un nouveau portage devrait être prochainement entrepris sur le protocole VRP⁷ défini par Alexandre Denis [41] pendant son stage de seconde année du MIM⁸ à l'Université de Sud Californie sous la direction de Carl Kesselman.

Des évolutions de Madeleine sont en cours et menées par Olivier Aumage [10] pendant son

⁷Variable Reliable Protocol.

⁸Magistère d'Informatique et Modélisation.

mémoire de DEA. Ces travaux devraient être intégrés aux prochaines distributions de PM². Des interfaces comme VIA offrent aujourd’hui de multiples possibilités pour la communication de données. VIA permet par exemple, outre le classique envoi de message, d’effectuer des opérations de type DMA⁹ à distance, ce mécanisme pouvant être supporté au niveau du matériel par des unités spécialisées. Ce type de transfert ne se justifie que pour des volumes de données importants et si les zones de mémoire à distances sont déjà punaisées¹⁰ en mémoire. Un des objectifs du travail d’Olivier Aumage était de proposer une évolution de Madeleine qui sache s’adapter aux différents modes de transfert dans les interfaces de bas niveau.

3.5 Utilisation des supports PVC et PM²

Cette section décrit quelques unes des expérimentations réalisées avec nos supports exécutifs. Dans le cadre du projet PVC-BOX, un des objectifs du projet était de concevoir un langage de programmation à objets parallèles (BOX). C. Gransart [69], P. Merle et C. Grenot [70] ont donc été les premiers utilisateurs de notre prototype PVC. Les différentes utilisations de PVC ont été limitées au sein de notre équipe à Lille. Cependant, cette première expérience nous a convaincus des bienfaits des expérimentations pour conforter nos choix sur l’interface de programmation et obtenir des résultats de performances significatifs. Une application de radiosité parallèle avait également été menée avec le support PVC, mais ces travaux n’ont pas été poursuivis car les personnes impliquées ont par la suite réorienté leurs travaux.

À la fin du projet PVC-BOX, une nouvelle orientation des travaux de l’équipe a été décidée avec Jean-Marc Geib. Un des objectifs pour le projet ESPACE était d’étendre les expérimentations vers des applications réelles et un plus grand nombre d’utilisateurs. C’est aussi à cette période que notre groupe a participé à l’action nationale Stratagème [116] où nous avons comme tâche de définir un support exécutif pour des applications fortement irrégulières. Nous nous sommes donc intéressés aux applications d’optimisation combinatoire. N’étant pas nous-mêmes des spécialistes de ce domaine, une collaboration très fructueuse a été menée avec l’équipe de Catherine Roucairol à Versailles. Des résultats intéressants ont été obtenus sur le problème d’affectation quadratique et présentés dans un workshop aux États-Unis [43]. Ce type d’application a par la suite été largement utilisée dans la thèse d’Yves Denneulin [42] pour évaluer les performances de ses régulateurs de charge. L’action Stratagème nous a permis également de collaborer avec le projet INRIA SLOOP de Nice et Françoise Baude qui cherchaient un support pour la bibliothèque orientée objet Schooner [63]. Stratagème fut aussi l’occasion d’établir des discussions très fructueuses et animées avec l’équipe de Grenoble, en particulier pour la définition et la classification de l’irrégularité [64].

D’autres collaborations et expérimentations ont aussi été menées en dehors de tout cadre contractuel ou programme de recherche. On peut citer les travaux avec l’équipe de Jean Roman à Bordeaux autour de l’algèbre linéaire creuse [39], ou bien encore les travaux de Bertrand Ducourthial à Orsay sur des modèles de réseaux associatifs [48].

3.5.1 PM² comme environnement applicatif

Nous décrivons dans cette section quelques-unes des utilisations de PM² où le caractère irrégulier des applications fait que PM² constitue un excellent support. Ces utilisations ont été pour la plupart

⁹Direct Memory Access.

¹⁰L’unité DMA écrit directement dans la mémoire sans passer par le gestionnaire mémoire du système.

faites par des étudiants préparant un DEA ou une thèse. Les différents rapports (thèse, mémoire de DEA) décrivant complètement leur travail sont cités dans les références bibliographiques.

3.5.1.1 Optimisation combinatoire

Comme nous l'indiquons dans l'introduction de ce chapitre, l'optimisation combinatoire a été une des premières applications à utiliser l'environnement PM². L'équipe OPALE (PRiSM, Versailles) a une longue expérience de la résolution des problèmes d'optimisation combinatoire qui, en terme de complexité, sont classés NP-complets. La résolution de ces problèmes consiste à rechercher la meilleure valeur, dans un ensemble fini mais de cardinalité très grande donc excluant toute recherche par une simple énumération exhaustive. Il n'existe à ce jour aucun algorithme polynomial connu pour les résoudre. Les algorithmes qui résolvent ces problèmes manipulent des structures de données irrégulières et le comportement de ces applications est fortement irrégulier. Les méthodes exactes comme les parcours d'espace de recherche en recherche opérationnelle (B&B) ou en intelligence artificielle (A*) génèrent dynamiquement des structures de données (graphes, arbres) de sous-problèmes qui sont souvent irrégulières et dont la taille croît exponentiellement avec celle du problème.

L'affectation quadratique est particulièrement représentative du phénomène de l'explosion combinatoire. Ce problème consiste à implanter n unités sur n sites de façon à minimiser un coût dépendant de la liaison entre les unités à placer (matrice F des flux) et de la proximité entre les sites (matrice D des distances). Notre collaboration avec l'équipe OPALE de Versailles nous a permis de résoudre exactement pour la première fois des problèmes célèbres dont la solution optimale n'était pas encore connue (Nugent 20, 120 millions de nœuds explorés en 3 heures 45 sur l'IBM SP2 de l'équipe APACHE à Grenoble). Ces travaux ont été présentés dans un workshop [43] aux États-Unis dont le thème était l'optimisation combinatoire parallèle. Cet article figure dans la liste des articles qui accompagnent ce document.

Une des contributions de l'équipe OPALE est d'avoir conçu et développé une bibliothèque applicative générique¹¹ pour aider au développement des applications résolvant les problèmes classiques d'optimisation, comme le sac à dos, le voyageur de commerce ou encore l'affectation quadratique. L'environnement PM² constitue une des cibles de la bibliothèque BOB qui a été portée sur PM² pendant la thèse de Bertrand Le Cun [85]. Les expérimentations continuent aujourd'hui avec une évolution majeure de BOB avec une conception orientée objet (BOBO : BOB orientée Objet) pour permettre plus de flexibilité et de modularité aux utilisateurs.

Nous participons également avec les membres de l'équipe OPALE à un programme de coopération entre l'Université de l'Illinois à Urbana Champaign et le CNRS-SPI avec l'équipe de Sanjay Kalé.

3.5.1.2 Algèbre linéaire

L'équipe du LaBRI (Jean Roman, Bordeaux) étudie l'implantation d'algorithmes parallèles de factorisation de grands systèmes linéaires creux. La maîtrise de ces algorithmes constitue souvent la clé de la résolution efficace d'une grande classe de problèmes de calcul scientifique tels que les problèmes non structurés provenant de la résolution d'équations aux dérivées partielles avec discrétisation par éléments finis. Une des contributions de l'équipe de Jean Roman consiste en la conception et la réalisation d'un solveur direct parallèle pour machine MIMD de type "Cholesky-

¹¹BOB, http://www.prism.uvsq.fr/french/parallele/cr/bob_fr.html

Crout” par blocs pour de grandes matrices creuses symétriques définies positives ; le calcul par blocs repose sur l’utilisation systématique de primitives BLAS3.

L’originalité de ce solveur réside dans l’utilisation des mécanismes permettant d’exploiter l’irrégularité du flot de calcul associé aux données : tâches légères (threads) activées dynamiquement et partageant la mémoire du processeur, réception de messages non déterministe, communications asynchrones avec des tampons de taille calculée de manière adaptative pour favoriser le recouvrement calcul/communication. Dans cette méthode de résolution, les éléments non nuls sont, dans une première phase, regroupés en blocs puis un traitement symbolique sur les blocs ainsi créés permet d’obtenir le graphe de précédence de l’algorithme de Cholesky sur la matrice obtenue. Cette deuxième étape ne nécessite que de connaître la taille des blocs créés et non de résoudre complètement, ou même partiellement, le système. À partir de ce graphe il est possible de calculer, statiquement, un ordonnancement optimal pour cet algorithme parallèle irrégulier mais en un temps éventuellement très grand. Des implémentations ont été réalisées sur des machines comme la Paragon et l’IBM-SP2. De nombreux tests pour des grandes matrices creuses (en particulier provenant de la méthode des éléments finis à 260.000 inconnues avec 20 Mo de données) ont montré l’intérêt de la démarche algorithmique pour le prétraitement et les qualités du solveur. Ces premiers tests ont été réalisés avec les implémentations MPI disponibles sur les plates-formes Paragon et SP2. L’objectif de notre collaboration était d’évaluer l’intérêt d’utiliser un environnement comme PM² pour supporter de telles applications. Un étudiant de l’ENSERB et du DEA Informatique de Bordeaux [39] s’est donc intéressé à évaluer l’utilisation de PM² et les possibilités de régulation de charge par migration que propose PM². Malheureusement, les performances obtenues à la fin du stage sont encore décevantes en comparaison avec celles obtenues directement au-dessus de MPI. Après discussion avec Jean Roman, nous pensons que des améliorations peuvent être apportées et ce travail préliminaire devrait être poursuivi cette année avec un nouveau stage d’étudiant.

Dans le contexte du calcul scientifique, notre support a également été utilisé à Lille pour la programmation de méthodes numériques hybrides dans un environnement hétérogène. Pendant sa thèse, Guy Bergère [13] a effectué plusieurs expérimentations avec l’environnement PM² et l’ordonnanceur adaptatif Mars [71].

3.5.1.3 Analyse d’images

C’est à l’occasion de l’école d’hiver Icare à Aussois en décembre 1997 que j’ai eu l’occasion de discuter avec Bertrand Ducourthial en thèse à Orsay qui travaillait sur un modèle exécutif data-parallèle (les réseaux associatifs) et qui cherchait un support exécutif adapté pour satisfaire les exigences de son modèle. L’environnement PM² l’a intéressé pour sa granularité d’exécution et ses mécanismes de migration. D’autre part, le LRI disposait d’un réseau Myrinet avec des PC et PM² était donc un bon candidat pour ses expérimentations. Sa thèse [48] a été soutenue en janvier 1999 et contient un chapitre complet décrivant l’implémentation du modèle et des mesures de performances avec le réseau Myrinet installé à Orsay.

La bibliothèque ANET écrite en C++ permet l’écriture de programmes data-parallèles à données et traitements irréguliers, utilisant un opérateur asynchrone d’activation des unités (directe-association). Un programme écrit avec ANET lance un processus maître et plusieurs esclaves sur l’ensemble des machines de la configuration. Les diverses associations sont parallélisées en respectant l’irrégularité du réseau. Le faible coût de création d’un thread PM² autorise une adaptation rapide à l’irrégularité des données et des traitements. La segmentation d’image constitue l’application qui a été principalement étudiée par Bertrand Ducourthial. Il a développé une série d’expériences

démontrant l'expressivité de son modèle et de bonnes performances avec une implémentation sur PM².

3.5.2 PVC et PM² comme cibles de compilation

Historiquement, l'utilisation de nos supports était principalement destinée à des compilateurs de langages parallèles. Dans le projet PVC-BOX, notre support PVC a servi de support exécutif au compilateur du langage à objets BOX. Les langages à objets parallèles ont presque en permanence constitué des utilisateurs privilégiés de nos environnements. Dans le contexte du projet SLOOP de Nice, Nathalie Furmento a proposé la bibliothèque Schooner [63] comme support d'un compilateur C++//. Plus récemment, Phil Hatcher [87] s'est intéressé à utiliser PM² comme support exécutif de programmes Java s'exécutant sur une grappe de PC.

Les langages à parallélisme de données ont connu un vif succès au début des années 1990 avec des architectures de machines synchrones (MasPar, CM2, etc). Les technologies évoluant, la compilation des langages data-parallèles a posé de nombreux problèmes pour des architectures parallèles asynchrones. Des travaux ont donc été menés pour proposer de nouvelles solutions de compilation et c'est ainsi que les supports à base de processus légers ont constitué une voie alternative intéressante. Nous décrivons dans cette section des expériences d'utilisation de nos supports pour des compilateurs de langages data-parallèles.

3.5.2.1 Langages à objets

Les langages à objets connaissent un succès sans cesse croissant et qui est motivé par des notions de génie logiciel. Les qualités des langages à objets sont la modularité (classes), la réutilisabilité (héritage, généricité) et aussi de forcer les programmeurs à avoir une démarche structurée. La tentation a été assez forte dans les années 1990 pour faire utiliser cette technologie dans le contexte des applications parallèles. C'est ainsi qu'une multitude de langages de programmation plus ou moins exotiques sont apparus¹². Aujourd'hui la tendance est nettement orientée vers l'utilisation de langages standards¹³.

Dans cette section, trois utilisations de PVC et PM² seront présentées. D'abord le langage BOX défini dans notre équipe, ensuite la bibliothèque Schooner développée à Nice pour le support de C++// et enfin une implémentation parallèle et distribuée d'un exécutif Java développée dans le New Hampshire.

BOX

BOX était un langage de programmation parallèle à objets destiné à être utilisé pour la programmation d'applications sur des architectures à mémoire distribuée. Les travaux autour de ce langage constituaient le second axe de recherche dans notre équipe et ses activités étaient principalement dirigées par Jean-Marc Geib. Les développements de ce langage ont fait l'objet de la thèse de Christophe Gransart [69] et de deux étudiants de DEA [70].

L'originalité du langage BOX était d'être un langage orienté objet définissant de nouvelles notions pour prendre en compte le parallélisme et la distribution. Le langage BOX utilise deux classes de composants élémentaires pour structurer et distribuer les applications : les fragments et les objets.

¹²Dont le nôtre.

¹³Java semble en passe de devenir le standard au niveau des langages à objets, n'en déplaise aux puristes!

Les fragments. Il s'agit d'objets actifs, des entités munies d'une activité propre ; ils sont instances d'une classe de fragments définissant des comportements (l'activité du fragment exécutera un de ces comportements) et des attributs (l'environnement d'exécution) ; c'est le grain de décomposition des activités et des données.

Les objets. Ce sont des objets passifs, ils correspondent aux objets classiques des langages à objets ; ils sont instances d'une classe d'objets définissant des attributs et des procédures ; ils sont utilisés pour structurer l'ensemble des fragments en abstractions réutilisables et pour représenter des ressources partagées.

Tout attribut, d'un fragment ou d'un objet, peut référencer un objet ou un fragment. Le langage BOX permet ainsi de définir des « objets actifs complexes » mélangeant des objets et des fragments.

La communication Le langage BOX permet deux types de communication : l'envoi de message et l'appel de procédure.

L'envoi de message asynchrone. Le langage autorise l'utilisation de boîtes aux lettres créées dynamiquement par les objets et les fragments, un fragment étant muni d'une boîte aux lettres implicite ; le fragment est en fait le reflet au niveau du langage de la notion de CAC.

L'appel de procédure. Il correspond à l'appel synchrone procédural d'une méthode sur un objet.

Un objet est un composant passif et ne peut donc être sollicité que par un appel de procédure tandis qu'un fragment, étant un composant actif, ne peut être sollicité que par un envoi de message.

La synchronisation La création de fragments provoque la création de nouveaux flots d'exécution. Le langage BOX doit offrir des possibilités de synchroniser ces différents flots. Il permet deux types de synchronisation : la synchronisation par objets partagés et la synchronisation sur la réception de message.

Synchronisation par objets partagés. Les appels de méthodes sur un objet peuvent être concurrents : les procédures peuvent s'exécuter en parallèle ; si l'accès à l'objet a besoin d'être synchronisé, le langage offre une politique de synchronisation de type lecteurs/rédacteurs [32] pour l'accès aux procédures d'un objet ; les procédures sont déclarées lectrices ou rédactrices de l'objet.

Synchronisation sur la réception de message. Les flots d'exécution peuvent se mettre en attente sur une opération de réception sélective de message sur une boîte aux lettres.

La distribution des entités

Les objets et les fragments peuvent être répartis sur n'importe quel nœud de la machine. La communication entre entités (objets et fragments) soit par appel procédural, soit par envoi de message, est uniforme quelle que soit la localisation des objets.

La création des objets et fragments est dynamique et synchrone. Elle a pour résultat la référence sur la nouvelle entité créée. La localisation de la création est laissée au système. Le compilateur BOX, au travers de son support exécutif, crée les nouveaux objets sur les nœuds où réside le code des méthodes de la classe et la description des attributs. De même, pour un fragment, le support du langage choisit un nœud où réside le code du comportement et le descriptif des attributs de la classe de fragments.

Le programmeur a cependant la possibilité de préciser statiquement dans un fichier de configuration la répartition du code de chacune de ses classes.

Schooner

SLOOP est un projet de l'INRIA Sophia Antipolis dirigé par Jean-Claude Bermond. Un des objectifs de ce projet était d'étudier la simulation à événements discrets avec un modèle de simulation basé sur des objets actifs et des activités. L'implémentation d'un tel environnement de simulation s'est appuyée sur la librairie de communication orientée objet Schooner¹⁴ développée par Nathalie Furmento [63] dans sa thèse. Schooner a pour objectif de fournir aux utilisateurs une hiérarchie de classes pour la programmation d'applications parallèles ou réparties. Cette librairie constitue un support de communication orienté objet dont les buts principaux sont 1) d'être indépendant de toute bibliothèque d'échange de messages (PVM, MPI, etc.), et 2) de donner à l'utilisateur une vision structurée et abstraite de son application.

Le modèle de programmation Schooner permet au programmeur d'ajouter dynamiquement des nœuds de calculs (*clusters*) à son application ; ces clusters peuvent alors interagir entre eux à l'aide de messages actifs. Un objet *cluster* est défini dans une classe et peut être vu comme une entité que manipule le programmeur pour désigner un contexte (CPU, mémoire, etc.) sur une machine.

Les communications dans Schooner reposent sur le modèle bien connu des messages actifs. La réception d'un message par un *cluster* a pour effet de déclencher le traitement associé au type du message reçu. L'intérêt d'un tel mode de communication est qu'il évite du côté récepteur l'appel à une fonction de réception de messages. Un objet d'un programme Schooner est donc créé sur un *cluster* (local ou distant) et cet objet sera par la suite capable d'émettre ou de recevoir des messages.

Java//

L'idée que défend Phil Hatcher est que les programmeurs éprouvent beaucoup de difficultés pour le développement d'applications parallèles parce qu'ils sont souvent contraints d'utiliser d'autres langages de programmation, d'appeler des fonctions bibliothèques de communication ou d'accès à des objets partagés. Pour eux, le rêve serait de concevoir ces applications avec un unique langage de programmation sans aucune modification syntaxique ou ajout de directives. Leur souhait est également que leurs programmes soient portables et que les détails de l'architecture soient complètement cachés. Ces différentes remarques "philosophiques" ont conduit Phil Hatcher à penser que Java pouvait devenir le langage de programmation parallèle de demain. Une des caractéristiques intéressantes de Java est que le processus léger (thread) est une abstraction du langage. Un programme Java peut donc être constitué d'un ensemble de processus légers qui se partagent les données et objets globaux. Le parallélisme peut donc être obtenu par une exécution parallèle des processus légers. Pour que la sémantique Java soit respectée, il est indispensable de disposer d'une mémoire partagée (ou virtuellement partagée) pour permettre le stockage des objets globaux.

Phil Hatcher (Université du New Hampshire) et deux étudiants¹⁵ travaillent depuis deux ans sur une machine virtuelle parallèle Java [87]. Cette machine virtuelle s'appelle Hyperion, son principe de fonctionnement est le suivant : le byte-code Java est converti dans un programme C qui s'exécute de manière parallèle avec des possibilités d'accès à une mémoire partagée d'objets. Dans un second temps, Hyperion fournit les mécanismes pour supporter une exécution parallèle et distribuée du programme C. Les communications sont assurées par la couche de communication SBP de Robert Russell [118], Marcel étant utilisé pour la gestion des processus légers. J'ai pendant un séjour de quinze jours porté la couche de communication Madeleine sur SBP. Une nouvelle version d'Hyperion

¹⁴Sloop Communication Library for High-level, Object-Oriented aNd Efficient Remote exchanges.

¹⁵Mark MacBeth et Keith MacGuigan.

est en cours de développement s'appuyant complètement sur PM^2 et sur un nouveau dispositif de mémoire partagée sur lequel travaille Gabriel Antoniu.

3.5.2.2 Langages data-parallèles

L'exécution à base de processus légers de programmes à parallélisme de données dans un environnement à mémoire distribuée [17] présente plusieurs perspectives intéressantes. Tout d'abord, cette approche favorise le recouvrement des communications par des calculs. En effet, quand un processus léger se bloque en attente d'un message, les autres processus légers peuvent continuer à calculer, le changement de contexte entre processus légers restant d'un coût minime. Ensuite, cette approche permet d'exploiter les grappes de machines multi-processeurs qui sont aujourd'hui devenues des plates-formes de référence pour le calcul parallèle. Il est en effet possible de répartir l'ensemble des processus légers sur les différents processeurs d'un nœud SMP. Enfin la possibilité de migrer des processus légers à faible coût offerte par PM^2 ouvre de nouvelles opportunités pour l'équilibrage dynamique de la charge.

PM^2 a été utilisé comme environnement d'exécution pour des compilateurs data-parallèles. Les compilateurs visés sont le compilateur *Adaptor HPF* développé par Thomas Brandes au GMD à Bonne et le compilateur *UNH C** développé par Phil Hatcher à l'université du New Hampshire. Ce travail a été réalisé pendant la thèse de Christian Perez [100]. Deux prototypes sont opérationnels dans les deux cas et une validation sur des codes applicatifs [7] est en cours. Les développements concernant *Adaptor HPF* ont été faits en liaison avec les autres travaux menés au LIP sur ce compilateur, en particulier avec Frédéric Desprez. Les développements concernant *UNH C** ont été conduits dans le cadre d'un contrat NSF/INRIA *C*IT* qui finance la collaboration entre l'équipe ParaDigne du LIP et le groupe de Phil Hatcher dans le New Hampshire.

Des travaux assez similaires ont été menés à Lille au début de sa thèse par Julien Soula qui cherchait à définir une machine abstraite à base de processus légers [120]. L'objectif de cette machine était de servir de support intermédiaire pour les langages à parallélisme de données irrégulier et dynamique, et s'exécutant en contexte distribué. Ce support était basé sur la notion de processeurs virtuels (entité d'exécution et de mémoire) qui masque au programmeur l'architecture distribuée sous-jacente. Ces processeurs virtuels sont regroupés en collections selon une topologie de voisinage, le tout constituant un objet data-parallèle (DPO).

3.5.3 PM^2 comme support de régulateurs de charge

Les applications fortement irrégulières, comme celles d'optimisation combinatoire, ont comme caractéristique majeure d'être potentiellement génératrices d'un parallélisme massif en terme de tâches et d'être extrêmement irrégulières et imprévisibles au niveau de la durée d'exécution des tâches générées. Cela exige donc du support exécutif des capacités évoluées de régulation et des aptitudes à supporter efficacement un nombre de tâches très important. Ces contraintes ont guidé les choix de conception de PM^2 . La plate-forme PM^2 s'est ainsi construite autour d'un noyau de multiprogrammation très efficace (Marcel) capable de supporter un grand nombre d'activités. Un mécanisme original de migration est également proposé pour offrir des possibilités de régulation très évoluées. Yves Denneulin a pendant sa thèse [42] étudié des mécanismes de régulation de charge génériques basés sur l'opération de migration.

Un autre projet de recherche sur la régulation de charge a été mené à Lille par Zouhir Hafidi

[71] pendant sa thèse. C'est le projet Mars¹⁶ qui définit des stratégies de régulation adaptatives sur des réseaux de stations partagées entre des utilisateurs actifs et des applications parallèles. Le principe de l'ordonnanceur Mars est d'utiliser la puissance de calcul des stations de travail pendant les périodes d'inactivité¹⁷.

3.5.3.1 LBMP : Load Balancing with Migration and Priorities

Le module d'ordonnancement LBMP a constitué une des contributions de la thèse d'Yves Denneulin [42]. Le principe de LBMP est d'étendre la notion de priorité définie dans la bibliothèque Marcel (sur un processeur) à une configuration parallèle constituée d'un ensemble de nœuds. Nous aborderons dans cette section quelques-uns des problèmes posés pour le respect de ces priorités.

Une configuration PM² est composée d'un ensemble de nœuds, où sur chacun d'entre eux, un ordonnanceur local (Marcel) de threads s'exécute. Pour simplifier la présentation, nous ne considérons que des configurations homogènes (même type de processeur, même puissance de calcul). Sur chaque nœud, les processus légers sont ordonnancés en respectant leur priorité locale, sans tenir compte des processus légers des autres nœuds. Dans un contexte parallèle, la vitesse du thread est donnée par la formule suivante :

$$sp(x) = \frac{pr(x)}{\sum_{y \in \text{nœud où } x \text{ s'exécute}} pr(y)}$$

Cela veut donc dire que PM² n'assure pas un respect *global* des propriétés sur les priorités. En d'autres termes, la priorité d'un processus léger n'a de sens que comparée aux autres processus légers qui s'exécutent sur le même nœud. Ce non-respect des priorités n'est pas satisfaisant, spécialement pour des applications qui ont des besoins particuliers sur le contrôle de l'exécution de l'application. La figure 3.5 illustre le problème général sur une configuration PM² avec trois nœuds.

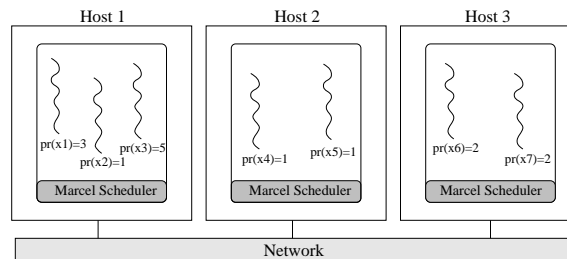


FIG. 3.5: Situation où un ordonnanceur global de threads serait utile !

Les problèmes suivants peuvent être observés.

- Le thread x_1 a une vitesse de $sp(x_1) = 1/3$. La vitesse de x_6 est de $sp(x_6) = 1/2$. Alors que le thread x_1 a une plus grande priorité que le thread x_6 , il s'exécute plus lentement.
- Le thread x_3 a une vitesse de $sp(x_3) = 5/9$. Le thread x_3 s'exécute plus vite que x_6 , mais pas $5/2$ plus vite.

¹⁶Multi-user Adaptive paRallel Scheduler.

¹⁷Principalement quand l'utilisateur interactif est absent, la nuit ou les week-ends.

LBMP est un module de régulation et d'ordonnancement développé au dessus de PM². LBMP offre aux applications parallèles un moyen d'agir sur l'ordonnancement des tâches grâce à des priorités qui sont choisies par le programmeur. Il essaie à tout instant (ou presque) d'assurer que les propriétés sont respectées. LBMP prend en charge les créations de processus en choisissant le nœud où débutera l'exécution. LBMP s'occupe également des migrations de threads pour respecter l'ordre dans les priorités et les vitesses. Pour l'exemple de la figure 3.5, une possibilité pour respecter à nouveau l'ordre est d'effectuer des migrations comme nous le montrons sur la figure 3.6 (pour obtenir une somme des priorités de 5 sur chacun des nœuds).

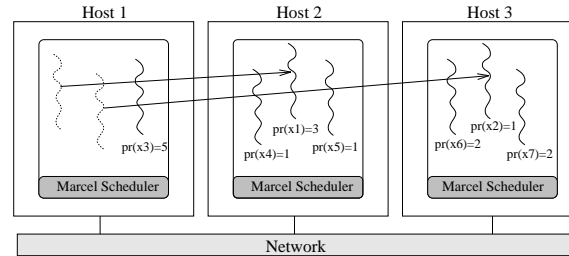


FIG. 3.6: Opérations de migration pour rétablir la propriété P .

Le choix des activités à migrer pour rétablir l'équilibre est un problème non trivial à résoudre car il revient au problème bien connu du « sac à dos ». En effet il convient de choisir des processus légers dont la somme des priorités doit être la plus proche possible d'une valeur sans la dépasser. Cela est équivalent à remplir un sac à dos avec des objets dont la somme des poids ne doit pas dépasser une valeur donnée.

Yves Denneulin a choisi ici de ne pas garantir en permanence la solution optimale mais de procéder de manière itérative, sans « retour en arrière¹⁸ ». Ce choix est dicté par des soucis de performance car cette opération ne doit pas prendre trop de temps, la situation sur laquelle elle agit fluctuant fortement au cours du temps. La méthode qui a été adoptée¹⁹ est la suivante : les processus légers sont triés par ordre décroissant de priorité puis stockés dans un tableau. Les threads sont choisis au fur et à mesure tant que leur choix ne dépasse pas la valeur fixée. Lorsque c'est le cas, le thread n'est pas choisi, mais le thread suivant est essayé et ainsi de suite jusqu'à avoir épuisé la liste complète. Ainsi on ne trouve pas forcément la solution optimale mais le coût de la recherche est en $O(n)$ seulement, n étant le nombre de processus légers candidats, le coût du tri étant en $O(n \log n)$.

3.5.3.2 Mars

Mars est un environnement de programmation [71] supportant l'exécution simultanée de plusieurs applications pouvant appartenir à différents utilisateurs. Il comprend essentiellement un ordonnancement adaptatif et des mécanismes de régulation de charge inter-applications. Mars a été développé au dessus des premières versions de PM² où la couche de communication était encore PVM.

Le style de programmation préconisé par Mars est de type Maître/Travailleurs (SPMD). La tâche maître génère du travail (des tâches applicatives) dont l'exécution sera déléguée aux tâches travailleuses. Cette tâche maître assure le contrôle du bon déroulement de l'application et de sa

¹⁸Backtracking en anglais.

¹⁹Il s'agit de la méthode « gloutonne » classique pour le problème du sac à dos.

terminaison. Bien que ce modèle souffre de graves limitations (extensibilité, goulot d'étranglement des communications), il se prête bien à un environnement évolutif où de nouveaux nœuds peuvent être ajoutés à la configuration sans que l'application s'en aperçoive.

Des applications d'algèbre linéaire [91] (Gauss-Jordan), des méthodes numériques hybrides [13] et des applications d'optimisation combinatoire (recherche tabu [124]) ont été développés au dessus du système Mars et PM².

3.6 Bilan des projets

Comme nous le mentionnions dans l'introduction, les recherches autour de ces supports exécutifs ont permis à six étudiants de préparer des thèses de doctorat, des DEA et pour la plupart d'entre eux de poursuivre dans le milieu académique. Je voudrais aussi souligner que ces étudiants ont eu à assurer à la fois un travail de recherche mais également des développements et même parfois de l'assistance aux utilisateurs. Une des difficultés de ce domaine de recherche en France provient très certainement du manque criant d'ingénieurs et d'assistance pour les chercheurs et projets de recherche dans les centres universitaires.

Les choix de conception et de réalisation ont été guidés par les technologies et par nos différentes expériences. Pour PVC, nous avons délibérément choisi de construire un environnement d'exécution au-dessus du système d'exploitation en utilisant les fonctionnalités fournies par le système Helios et les processeurs Transputers. Un des inconvénients de cette approche a été que le portage de PVC sur d'autres architectures a posé de nombreux problèmes techniques et que les performances ont été assez décevantes. Nos choix pour PM² du projet ESPACE ont été dirigés par ces conclusions et nous avons essayé de mettre l'accent sur la portabilité de l'environnement et un haut niveau de performance. Nous nous sommes aussi efforcés de promouvoir et de faire utiliser cet environnement dans la communauté scientifique, principalement en France, grâce à l'opération Stratagème. Un des regrets que l'on pourrait avoir est peut-être de ne pas avoir réussi à créer une dynamique commune sur ce thème en France permettant de mettre en commun nos forces de travail avec d'autres projets, en particulier avec le projet Apache. Les raisons sont d'abord d'ordre géographique et aussi qu'il est difficile de remettre en cause des développements logiciels lourds. J'espère pouvoir dans les prochaines années réussir à mieux coordonner en France ces activités sur les supports exécutifs, en particulier en synchronisant mieux nos activités sur de nouvelles thématiques.

Dans le premier chapitre de ce document, nous avons identifié deux contextes architecturaux. C'est certainement au niveau des grappes de PC et des réseaux à haut débit que nos efforts se sont surtout portés ces dernières années, en particulier avec PM² et sa couche de communication Madeleine. Dans le contexte du métacomputing, nos travaux ont débuté avec les travaux de Benoît Planquelle [105] qui cherche à fournir de nouvelles fonctionnalités autour de la fédération de plusieurs grappes hétérogènes. Il s'est intéressé en particulier à définir des mécanismes performants de conversion de données et la gestion efficace de plusieurs protocoles de communication en utilisant des processus légers pour la scrutation. Des collaborations plus rapprochées sont en cours avec les projets INRIA sur Rhône-Alpes pour faire évoluer nos environnements et construire une plate-forme régionale autour du métacomputing. Des expérimentations sont également en cours en utilisant l'environnement Globus pour une bibliothèque de calcul scientifique parallèle Scilab avec Frédéric Desprez.

Chapitre 4

Conclusions et perspectives

Les recherches que nous avons menées depuis dix ans ont été motivées par la volonté d'offrir aux utilisateurs des supports exécutifs simples à exploiter et leur permettant également de tirer profit du potentiel et des performances des architectures parallèles ou distribuées. C'est ainsi que les supports et environnements que nous avons conçus, en particulier PM^2 , ont été utilisés par plusieurs équipes de recherche dans le cadre de thèses, de stages de DEA ou de projets de fin d'études. Je suis aussi particulièrement fier d'avoir dans ce contexte encadré six étudiants en thèse, qui ont presque tous poursuivi une carrière académique dans différents centres universitaires français. Je tiens à leur rendre hommage en soulignant qu'ils ont assuré la presque totalité des développements, qui auraient pu pour l'essentiel l'être par des ingénieurs et programmeurs, comme cela se fait dans des centres de recherches étrangers. D'un point de vue plus personnel, je pense avoir réussi à mener une activité cohérente sur le thème des supports d'exécution à base de processus légers. Je suis également assez fier d'avoir réussi un équilibre, certes pas parfait, entre les trois types d'activités que doivent assumer les enseignants-chercheurs en France, à savoir l'enseignement avec un volume annuel de 200 heures, des responsabilités administratives assez diverses en enseignement/recherche et la poursuite d'une activité de recherche. L'ensemble de ces différentes activités est par ailleurs résumé dans un curriculum-vitae détaillé.

Les expérimentations et utilisations de nos travaux sont résumées dans la dernière partie du chapitre 3 avec des références (articles, mémoires de thèse, DEA) présentant plus en détail ces travaux. J'aimerais également au niveau de cette conclusion insister sur l'originalité de notre démarche dans la conception des supports exécutifs. Alors que beaucoup d'équipes du domaine mettent plutôt l'accent sur la portabilité des applications et des supports, nous nous sommes plus particulièrement intéressés à fournir un haut niveau de performance et nous nous sommes efforcés de porter nos réalisations sur un nombre important de plates-formes. Cet effort a un coût en heures de développement qu'il est assez difficile de chiffrer bien évidemment.

J'espère dans les prochaines années poursuivre dans cette même thématique en me focalisant plus particulièrement sur les points suivants : d'abord créer une dynamique de recherche et développement en France autour du calcul distribué à haute performance (métacomputing) et ensuite utiliser les techniques et technologies du parallélisme pour la mise en œuvre des logiciels et serveurs sur Internet.

Le métacomputing constitue une nouvelle voie à explorer pour le parallélisme. Cela concerne à la fois les aspects algorithmiques et méthodologiques pour la conception d'applications, mais également les supports exécutifs adaptés aux nouvelles contraintes posées par ce type d'architecture. Plusieurs opérations sont en cours sur ce sujet ; j'ai monté avec Yves Robert un groupe de travail au LIP sur le thème du métacomputing qui rassemble la presque totalité des chercheurs du projet ReMaP. Le

principe de ce groupe de travail est de coordonner nos travaux sur ce thème, aussi bien au niveau algorithmique du métacomputing (Yves Robert) que celui des bibliothèques de calcul scientifique (Frédéric Desprez) ou bien celui des supports exécutifs. Je participe également au niveau de la région Rhône-Alpes à la mise en place d'une action incitative sur le thème du métacomputing autour des équipements et grappes de stations qui sont opérationnelles sur Lyon et Grenoble. Ce projet devrait également concerner de nouveaux aspects applicatifs, comme la bio-informatique en collaboration avec les projets universitaires et INRIA sur la région Rhône-Alpes.

Depuis mon arrivée au LIP en octobre 1998, je participe à deux projets de recherche et développement autour de l'Internet à haut débit. Dans le cadre du projet CHARM (Caches à Haut Débit pour les AutoRoutes Multimédias), je m'intéresse à la conception d'un cache WWW performant pour servir de tête de réseau de boucles locales. Ce projet est mené en collaboration avec plusieurs industriels, comme Matra Systèmes et Information et Rhône-Vision Câble pour la partie réseau câblé. L'architecture matérielle de la solution que nous proposons se construit autour d'une grappe de PC et de l'utilisation des technologies issues du parallélisme et de nos travaux. En collaboration avec Loïc Prylli et Jean-Christophe Mignot, nous nous intéressons à un meilleur support des entrées-sorties distribuées en exploitant les possibilités d'asynchronisme et en réalisant un couplage plus fort entre les interfaces réseaux et disques. Le projet SPIHD (Services et Programmes pour l'Internet à Haut Débit) sur cette même thématique est en cours de démarrage avec d'autres partenaires industriels, en particulier du domaine de l'audio-visuel.

Annexe A

Exemple de programme PVC

A.1 L'application somme

L'application *somme* calcule en parallèle la somme des entiers de 1 à n. Un seul comportement est défini : le comportement *B_SOM*. Celui-ci reçoit en paramètre l'intervalle d'entiers dont il doit calculer la somme. Si l'intervalle est de cardinalité strictement supérieure à 1, le composant crée deux nouveaux composants de même comportement chargés de calculer chacun la somme pour une moitié de l'intervalle. Le composant attend les deux réponses, en effectue la somme et envoie le résultat à son créateur.

L'application est découpée en deux types de modules : *Module_Dia* qui est chargé de démarrer l'application et de récupérer le résultat, et *Module_Somme* qui abritera les CAC de comportements *B_SOM*. Le *Module_Somme* sera placé sur un des nœuds de la machine parallèle, tandis que les modules *B_SOM* seront dupliqués sur l'ensemble des nœuds.

A.2 Le module *Module_Dia*

```
/*
*****
*/
/*          Module_Dia.c          */
/*          */
/*          */
*****
*/

#include <Behavior.h>
#include <Prim_Cac.h>

typedef struct sum_param {
    Component reply;
    int inf;
    int sup;
} sum_param;

void lance(Component me, int *arg)
{
    /* comportement demarrant l'application */
    Component root;
    int limite, result, tps;
}
```

```

Message_Ptr mess;
sum_param parameters;

PrintString(me, "\n*****APPLI***** \n\n");
PrintString(me, "Limite :"); Flush(me);
ReadInt(me, &limite);

parameters.reply = me;
parameters.inf = 1;
parameters.sup = limite;

/* creation du premier composant de comportement B_SOM */
root = NewComponent(me, B_SOM, VOID, sizeof(sum_param), (char *)&parameters);

/* attente du resultat */
GetMessage(me, me, &mess);
result = *(int *)GetFirstArg(mes, sizeof(int));

PrintString(me, "result summ of 1 to ");
PrintInt(me, limite);
PrintString(me, " = ");
PrintInt(me, result);
PrintString(me, "\n\n\n");

/* arret de l'application */
StopModules(me);
}

int main(int argc, char ** argv)
{
word box_main;

/* initialisation du module PVC */
InitModule(argc, argv, 1);

/*
Creation d'un comportement a partir de la
fonction 'lance'
*/
InsertBehavior(lance, B_DIA);

box_main = NewBox(0);

/*
creation d'un cac avec le comportement DIA
(fonction lance est executee
*/

NewComponent(box_main, B_DIA, LOCAL, 0, NULL);

EndModule();
}

```

A.3 Le module *Module_Somme*

```

/*****
/*
/*          Module_Somme.c
/*
/*
/*****
#include <Behavior.h>
#include<Prim_Cac.h>

typedef struct sum_param {
    Component reply;
    int inf;
    int sup;
} sum_param;

void sum(Component me, sum_param *arg)
{
    Component left, right;
    int result;
    Message_Ptr message;
    sum_param parameters;

    if (arg->inf == arg->sup)
        /* envoi du resultat au CAC createur */
        Send(me,arg->reply,sizeof(int),(char *) &(arg->inf));
    else {
        /* creation d'un CAC pour calculer la premiere partie de la somme */
        parameters.reply = me;
        parameters.inf = arg->inf;
        parameters.sup = (arg->inf+arg->sup)/2;
        left = NewComponent(me,B_FAC,VOID,sizeof(fac_param),(char *) &parameters);

        /* creation d'un CAC pour calculer la deuxieme partie de la somme */
        parameters.inf = ((arg->inf+arg->sup)/2)+1;
        parameters.sup = arg->sup;
        right = NewComponent(me,B_FAC,VOID,sizeof(fac_param),(char *) &parameters);

        /* attente du resultat de l'un des CAC */
        GetMessage(me,me,&message);
        result = *(int *)GetFirstArg(message,sizeof(int));
        FreeMessage(message);

        /* attente du resultat du dernier CAC */
        GetMessage(me,me,&message);
        result += *(int *)GetFirstArg(message,sizeof(int));
        FreeMessage(message);

        /* envoi du resultat au CAC createur */
        Send(me,arg->reply,sizeof(int),(char *) &result);
    }
}

int main(int argc, char ** argv)
{
    InitModule(argc,argv,1);
    InsertBehavior(sum,B_SOM);
    EndModule();
}

```

}

Annexe B

Exemple de programme PM²

B.1 Principe d'implémentation de l'application somme avec PM²

Comme exemple illustrant l'utilisation de l'environnement PM², nous avons repris l'exemple du programme somme qui avait été détaillé pour PVC. Contrairement à PVC, nous avons choisi pour PM² un mode SPMD où l'ensemble du code est répliqué sur l'ensemble des processeurs.

Nous utilisons dans cet exemple l'appel de procédure à attente différée. L'appel à la fonction distante se fait par l'opération LRP_CALL et l'attente du résultat par le LRP_WAIT. Les paramètres du LRP_CALL permettent donc de désigner la fonction à exécuter, la taille de pile nécessaire pour le thread et les paramètres en entrée et retour de la fonction. La fonction `next_module` choisit de manière cyclique le numéro du nœud où sera exécutée la fonction.

B.2 L'application somme avec PM²

```
#include "rpc_defs.h"

/* ***** */

int *les_modules, nb_modules, module_courant = 0;

int next_module(void)
{ int res;

  lock_task();
  module_courant = (module_courant+1) % (nb_modules);
  res = les_modules[module_courant];
  unlock_task();
  return res;
}

BEGIN_SERVICE(DICHOTOMY)
int i;

if(req.inf == req.sup) {
  res.res = req.inf;
} else {
  int mid = (req.inf + req.sup)/2;
```

```

LRPC_REQ(DICHOTOMY) req1, req2;
LRPC_RES(DICHOTOMY) res1, res2;
pm2_rpc_wait_t att[2];

req1.inf = req.inf; req1.sup = mid;

/* Appel du service DICHOTOMY sur le noeud suivant */
LRP_CALL(next_module(), DICHOTOMY, STD_PRIO, DEFAULT_STACK,
          &req1, &res1, &att[0]);

req2.inf = mid+1; req2.sup = req.sup;
LRP_CALL(next_module(), DICHOTOMY, STD_PRIO, DEFAULT_STACK,
          &req2, &res2, &att[1]);

/* Attente des deux resultats */
for(i=0; i<2; i++)
    LRP_WAIT(&att[i]);

res.res = res1.res + res2.res;
}
END_SERVICE(DICHOTOMY)

BEGIN_SERVICE(LRPC_START)
LRPC_REQ(DICHOTOMY) req;
LRPC_RES(DICHOTOMY) res;

tfprintf(stderr, "Entrez un entier raisonnable "
            "(0 pour terminer) : ");
scanf("%d", &req.sup);

req.inf = 1;

LRPC(pm2_self(), DICHOTOMY, STD_PRIO, DEFAULT_STACK, &req, &res);
tfprintf(stderr, "1+...+%d = %d\n", req.sup, res.res);
END_SERVICE(START)

int main(int argc, char **argv)
{
    pm2_init_rpc();

    DECLARE_LRPC_WITH_NAME(LRPC_START, "start", OPTIMIZE_IF_LOCAL);
    DECLARE_LRPC_WITH_NAME(DICHOTOMY, "fac", OPTIMIZE_IF_LOCAL);

    pm2_init(&argc, argv, ASK_USER, &les_modules, &nb_modules);

    if(pm2_self() == les_modules[0]) { /* master process */

        LRPC(pm2_self(), LRPC_START, STD_PRIO, DEFAULT_STACK, NULL, NULL);

        pm2_kill_modules(les_modules, nb_modules);
    }

    pm2_exit();
    return 0;
}

```

Bibliographie

- [1] J.-F. ABRAMATIC, J. HUGON ET R. MAHL, éditeurs. *SM90 : une architecture modulaire et ses applications* (Versailles, décembre 1985), Eyrolles, INRIA, CNET et Agence de l'informatique. Actes des journées SM90.
- [2] G. Agha. *Actors. A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
- [3] P. America. « POOL-T : a parallel object-oriented language ». Dans *Object-Oriented Concurrent Programming* (1990), A. Yonezawa et M. Tokoro, éditeurs, Computer systems series, MIT Press, pp. 199–220.
- [4] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu et W. Zwaenepoel. « ThreadMarks : Shared Memory Computing on Network of Workstations ». *IEEE Computer* **29**, numéro 2 (février 1996), 18–28.
- [5] T. Anderson, B. Bershad, E. Lazowska et H. Levy. « Scheduler Activations : Effective Kernel Support for the User-Level management of parallelism ». *ACM Transactions on Computer Systems* **10**, numéro 1 (février 1992), 53–79.
- [6] G. Antoniu. « Allocation iso-adresse pour une migration préemptive de processus légers. Mise en œuvre dans PM2 pour la compilation data-parallèle ». Rapport de stage, DEA d'Informatique de Lyon, ENS Lyon, France, juin 1998.
- [7] G. Antoniu, L. Bougé et C. Perez. « Generic load balancing for HPF programs : Application to the Flame Simulation Kernel ». Dans *The 3rd Annual HPF User Group Meeting (HUG '99)* (Redondo Beach, California, août 1999). <http://www.ens-lyon.fr/~bouge/Biblio/Perez/AntoniouBougePerezHUG99.ps>.
- [8] G. Antoniu et C. Perez. « Allocation iso-adresse pour une migration préemptive de processus légers : application à un compilateur HPF ». Dans *Actes des Rencontres francophones du parallélisme (RenPar 11)* (IRISA, Rennes, France, juin 1999), pp. 43–48.
- [9] D. K. Arvind et D. Yokotsuka. « Debugging Concurrent Programs using Static Analysis and Run-time Hardware Monitoring ». Dans *Proceedings of IEEE Symposium on Parallel and Distributed Processing* (Dallas, Texas, décembre 1991).
- [10] O. Aumage. « Une librairie de communication portable et adaptative pour réseaux haut-débit ». Rapport de stage de DEA, DEA d'informatique de Lyon, Univ. Claude Bernard, Lyon 1, France, juin 1999.
- [11] T. Beisel, E. Gabriel et M. Resch. « An Extension to MPI for Distributed Computing on MPP's ». Dans *EuroPVM/MPI '97 : Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Cracow, Pologne, novembre 1997), M. Buback, J. Dongarra et J. Wasniewski, éditeurs, volume 1332 des *Lecture Notes in Computer Science*, Springer Verlag, pp. 75–83.

- [12] L. Bellisard, S. B. Atallah, F. Boyer et M. Riveill. « Distributed application configuration ». Dans *Proc. of the 16th International Conference on Distributed Computing Systems* (Hong-Kong, mai 1996), IEEE Computer Society, pp. 579–585.
- [13] G. Bergère. *Contribution à une programmation parallèle hétérogène des méthodes numériques hybrides*. Thèse de doctorat, Université de Lille 1, septembre 1999. Accessible à <http://www.lifl.fr/~bergere/these.ps>.
- [14] P.-E. Bernard, B. Plateau et D. Trystram. « Using threads for developing applications : Molecular Dynamics as a case study ». Dans *Parallel Numerics 96* (Gozd Martuljek, Slovenia, septembre 1996), Trobec, éditeur, pp. 3–16.
- [15] N. Boden, D. Cohen, R. Feldermann, A. Kulawik, C. Seitz et W. Su. « Myrinet : A Gigabit per second Local Area Network ». *IEEE-Micro* **15-1** (février 1995), 29–36.
- [16] J. Boudec. « The Asynchronous Transfer Mode : A tutorial ». *Computers Networks and ISDN Systems* **24** (1992), 279–309.
- [17] L. Bougé, P. Hatcher, R. Namyst et C. Perez. « A multithreaded runtime environment with thread migration for a HPF data-parallel compiler ». Dans *The 1998 Intl Conf. on Parallel Architectures and Compilation Techniques (PACT '98)* (Paris, France, octobre 1998), IFIP WG 10.3 and IEEE, pp. 418–425.
- [18] L. Bougé, J.-F. Méhaut, R. Namyst et L. Prylli. « Using the VI Architecture to build distributed, multithreaded runtime systems : a case study ». Dans *Proc. 2000 ACM Symposium on Applied Computing (SAC 2000)* (Villa Olmo, Como, Italy, mars 2000), ACM Special Interest Group on Applied Computing (SIGAPP), ACM. À paraître.
- [19] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson et R. Stevens. *Portable Programs for Parallel Processing*. Holt, Rinehart, and Winston, New York, 1987.
- [20] J. Briat, J.-M. Geib et J.-F. Méhaut. « Technologie et objectifs des supports d'exécution pour la régulation ». Dans *Actes de l'école d'hiver Icare* (Aussois, France, décembre 1997), pp. 125–160. École financée par le CNRS et le GDR-PRC PRS.
- [21] J. Briat, I. Ginzburg, M. Pasin et B. Plateau. « Athapascan Runtime : Efficiency for Irregular Problems ». Dans *Proceedings of the Euro-Par '97 Conference* (Passau, Germany, août 1997), volume 1300 des *Lectures Notes in Computer Science*, Springer Verlag, pp. 590–599.
- [22] R. M. Butler et E. L. Lusk. « Monitors, Messages, and Clusters : The P4 Parallel Programming System ». Rapport technique MCS-P362-0493, Argonne National Laboratory, 1993.
- [23] R. Buyya. *High Performance Cluster Computing. Volume 1 : Architecture and Systems*. Prentice Hall, 1999.
- [24] R. Buyya. *High Performance Cluster Computing. Volume 2 : Programming and Applications*. Prentice Hall, 1999.
- [25] P. Cadinot, N. Dorta, B. Folliot et P. Sens. « Mécanisme de va-et-vient en mémoire dans un environnement réseau haut débit ». Dans *Proceedings of MPR'96 (Journées sur la Mémoire Partagée Répartie)* (Bordeaux, mai 1996). Accessible à : <http://www-src.lip6.fr/homepages/Philippe.Cadinot/documents/Philippe.Ca%dinot.1.ps.gz>.
- [26] D. Caromel. « Concurrency and reusability : from sequential to parallel ». *Journal of Object Oriented Programming* **3**, numéro 3 (septembre 1990).
- [27] N. Carriero et D. Gelernter. « Linda in context ». *Communications of ACM* **32**, numéro 4 (1989), 444–458.

- [28] N. Carriero et D. Gelernter. *How to Write Parallel Programs : A First Course*. The MIT Press, 1990.
- [29] E. Cecchet. « SciOS : a distributed shared memory for SCI clusters ». Dans *Proc. of the third European Research Seminar on Advances on Distributed Systems (ERSADS'99)* (Madeira Island, Portugal, avril 1999).
- [30] M. Christaller. *Vers un support d'exécution portable pour applications parallèles irrégulières*. Thèse de doctorat en informatique, Université Joseph Fourier, Grenoble I, France, novembre 1996.
- [31] COMPAQ, INTEL, AND MICROSOFT CORPORATIONS. *Virtual Interface Architecture. Version 1.0*, décembre 1997. Available from <http://www.viarch.org/>.
- [32] P. Courtois, F. Heymans et D. Parnas. « Concurrent Control with Readers and Writers ». *Communications of ACM* **14**, numéro 10 (octobre 1971), 667–668.
- [33] L. Courtrai. *Les composants actifs de communication : outils pour la conception et l'implantation de langages parallèles à objets actifs pour machines MIMD*. Thèse de doctorat, Université des Sciences et Technologies de Lille, octobre 1992.
- [34] L. Courtrai, J.-F. Roos, C. Dumoulin, P. Merle, J.-M. Geib et J.-F. Méhaut. « Communicating Active Components : Support for Parallel Object Languages on Distributed Architectures ». Dans *Proceedings of EUSUG '92 European Sun User Group Conference* (Wiesbaden, novembre 1992).
- [35] CRAY RESEARCH, SGI. *Application Programmer's Library Reference Manual*, 1997. Documentation technique SR-2165.
- [36] C. Cruz-Neira, D. Sandin, T. DeFanti, R. Keynyon et J. Hart. « The CAVE : Audio visual experience automatic virtual environment ». *Communications of ACM* **35**, numéro 6 (1992), 65–72.
- [37] V. Danjean. « Extending the Linux kernel with Activations for better support of multithreaded program and integration in PM2 ». Rapport de stage de magistère, seconde année, ENS Lyon, France, septembre 1999. Stage à l'université du New Hampshire.
- [38] A. DARTE ET G.-R. PERRIN, éditeurs. *The Data-Parallel Programming Model*, volume 1136 des *Lectures Notes Computer Science*. Springer Verlag, 1996.
- [39] R. D. de Bernonville. « Support d'exécution à base de processus légers : régulation dynamique de charge pour des algorithmes irréguliers d'algèbre linéaire creuse ». Mémoire de DEA informatique, ENSERB, Bordeaux, juillet 1999.
- [40] E. Delattre. *Un bilan du projet OMPHALE*. Habilitation à diriger des recherches en informatique, Université des Sciences et technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, février 1990.
- [41] A. Denis. « Variable Reliable Protocol in Globus-Nexus ». Rapport de stage MIM2, Magistère d'informatique et modélisation (MIM), ENS Lyon, LIP, ENS Lyon, septembre 1999. Stage effectué sous la direction de Carl Kesselman à l'USC/ISI, accessible à <http://www.ens-lyon.fr/~denisa/work/stage2/vrp.ps.gz>.
- [42] Y. Denneulin. *Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin*. Thèse de doctorat, Université de Lille 1, janvier 1998.
- [43] Y. Denneulin, B. L. Cun, T. Mautor et J.-F. Méhaut. « Exact solution of large Quadratic Assignment Problems in parallel ». Dans *Proceedings of the 5th Inform's Computer Science Technical Section Conference, CTSS'96* (Dallas, Texas, janvier 1996).

- [44] Y. Denneulin, J.-F. Méhaut et R. Namyst. « Customizable thread scheduling directed by priorities ». Dans *Proc. Workshop on Multithreaded Execution and Compilation (MTEAC 99)* (Orlando, Florida, janvier 1999), G. Gao et D. Tullsen, éditeurs, Held in conjunction with 5th Intl Symp. On High Performance Computer Architecture (HPCA-5), IEEE.
- [45] F. Desprez, P. Ramet et J. Roman. « Optimal grain size computation for pipelined algorithms ». Dans *Euro-Par '96* (août 1996), volume 1123 des *Lectures Notes in Computer Science*, Springer Verlag, pp. 165–172.
- [46] R. Dimitrov et A. Skjellum. « An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing ». Dans *Proc. of the third MPI developer's Conference* (mars 1999). Accessible à <http://www.mpi-softtech.com/publications/mpipro-via.html>.
- [47] DOLPHIN INTERCONNECT SOLUTIONS, INC. *PCI-SCI Adapter Programming Specification*, mars 1997.
- [48] B. Ducourthial. *Les réseaux associatifs : un modèle de programmation à parallélisme de données, pour algorithmes et données irréguliers, à primitives de calculs asynchrones*. Thèse de doctorat, Université de Paris-Sud (Orsay), janvier 1999.
- [49] C. Dumoulin. « Un ramasse-miettes pour les composants actifs de communication ». Mémoire de DEA, Université de Lille 1, septembre 1992.
- [50] C. Dumoulin, J.-F. Roos et J.-F. Méhaut. « Le Ramasse-Miettes du projet PVC-BOX ». Dans *Actes des Rencontres francophones du parallélisme (RenPar 5)* (mai 1993), L. d'Informatique de Brest, éditeur, pp. 105–108.
- [51] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merrit, E. Gronke et C. Dodd. « The Virtual Interface Architecture ». *IEEE Micro* (mars 1998), 66–75.
- [52] G. Eddon et E. Henry. *Inside Distributed COM*. Microsoft Press, 1998.
- [53] A. Fagot. *Réexécution déterministe pour un modèle procédural parallèle basé sur les processus légers*. Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, septembre 1997.
- [54] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy et C. Thekkath. « Implementing Global Memory Management in a Workstation Cluster ». Dans *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (décembre 1995), ACM. Accessible à <http://www.cs.washington.edu/homes/levy/opal/sosp.ps>.
- [55] I. Foster, J. Geisler, W. Geisler, W. Nickless, W. Smith et S. Tuecke. « Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment ». Dans *Proc. of 5th IEEE Symposium on High Performance Distributed Computing* (1997), pp. 562–571. Accessible à : <ftp://ftp.globus.org/pub/globus/papers/isoft.ps.gz>.
- [56] I. Foster, J. Geisler, C. Kesselman et S. Tuecke. « Managing Multiple Communication Methods in High-performance Networked Computing Systems ». *Journal of Parallel and Distributed Computing* **40** (1997), 35–48.
- [57] I. Foster et N. Karonis. « A Grid-Enabled MPI : Message Passing in Heterogeneous Distributed Computing Systems ». Dans *Proceedings of SuperComputing'98*. ACM Press, 1998.
- [58] I. Foster et C. Kesselman. « The Globus Project : A Progress Report ». Dans *Proceedings of the Heterogeneous Computing Workshop* (mars 1998).

- [59] I. FOSTER ET C. KESSELMAN, éditeurs. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, juillet 1998.
- [60] I. Foster, C. Kesselman et S. Tuecke. « The Nexus approach to integrating multithreading and communication ». *Journal of Parallel and Distributed Computing* **37** (1996), 70–82.
- [61] I. Foster, S. Tuecke et S. Taylor. « A Portable Run-Time System for PCN ». Rapport technique ANL/MCS-TM-137, Argonne National Laboratory, 1991. Accessible à : ftp://info.mcs.anl.gov/pub_tech_reports/reports/TM137.ps.Z.
- [62] H. Franke, P. Hochschild, P. Pattnaik et M. Snir. « MPI-F : An efficient implementation of MPI on IBM-SP1 ». Dans *Proceedings of International Conference on Parallel Processing* (1994), volume III, CRC Publishers, pp. 197–201.
- [63] N. Furmento. *Schooner : Une encapsulation orientée objet de supports d'exécution pour applications réparties*. Thèse de doctorat, Université de Nice-Sophia Antipolis, mai 1999.
- [64] T. Gautier, J.-L. Roch et G. Villard. « Regular versus irregular problems and algorithms ». Dans *Proc. of IRREGULAR'95, Lyon, France* (septembre 1995), Springer-Verlag.
- [65] J.-M. Geib. *L'architecture du système réparti OMPHALE*. Thèse de doctorat, Université des Sciences et Technologies de Lille, janvier 1989.
- [66] J.-M. Geib. *Contributions aux systèmes à objets pour architectures distribuées*. Habilitation à diriger les recherches, Université des Sciences et Technologies de Lille, février 1993.
- [67] J.-M. Geib, C. Gransart et P. Merle. *CORBA : des concepts à la pratique*. Inter-Editions, 1997.
- [68] R. Gillet et R. Kaufmann. « Using the Memory Channel Network ». *IEEE Micro* **17**, numéro 1 (janvier 1997), 19–25.
- [69] C. Gransart. *BOX : un modèle et un langage à objets pour la programmation parallèle et distribuée*. Thèse de doctorat, Université des Sciences et Technologies de Lille, janvier 1995.
- [70] C. Grenot et P. Merle. « Le langage BOX : étude et réalisation ». Mémoire de DEA, Université de Lille 1, septembre 1992.
- [71] Z. Hafidi. *MARS : un environnement de programmation parallèle adaptative dans les réseaux de machines hétérogènes multi-utilisateurs*. Thèse de doctorat, Université des Sciences et Technologies de Lille, octobre 1998.
- [72] F. Hémery. *Étude de la répartition dynamique d'activités sur architectures décentralisées*. Thèse de doctorat, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, juin 1994.
- [73] High Performance Fortran Forum. « High Performance Fortran Language Specification Version 1.0 ». Rapport technique CRPC-TR9225, Center for Research on Parallel Computation, Rice University, 1993. Accessible à : <ftp://ftp.softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR9225.ps.gz>.
- [74] C. Hoare. « Communicating Sequential Processes ». *Communications of ACM* **21**, numéro 8 (août 1978), 666–677.
- [75] IBM CORP. *The IBM RS/6000 SP High-Performance Communication Network*. Available with IBM-SP technical documentation.
- [76] IBM CORPORATION. *IBM LoadLeveler User's Guide*. Manuel technique numéro SH26-7220.
- [77] IEEE. *Standard for Scalable Coherent Interface (SCI)*, août 1993. Standard IEEE numéro 1596.

- [78] H. Jamrozik. *Aide à la mise au point des applications parallèles et réparties à base d'objets persistants*. Thèse de doctorat en informatique, Université Joseph Fourier, Grenoble, mai 1993.
- [79] J. Jézéquel, F. André et F. Bergheul. « A Parallel Execution Environment for a Sequential Object-Oriented Language ». Dans *Proceedings of International Conference on SuperComputing* (July 1992), ACM.
- [80] W. Johnston. « High-Speed, Wide Area, Data Intensive Computing : A Ten Year Retrospective ». Dans *Proceedings of the High Performance Distributed Computing* (Chicago, July 1998), IEEE Computer Society, pp. 280–291.
- [81] D. Kafura et K. Lee. « ACT++ : Building a concurrent C++ with actors ». *Journal of Object Oriented Programming* **3**, numéro 1 (mai 1990).
- [82] S. Krakowiak, M. Meysembourg, H. Nguen, M. Riveill, C. Roisin et X. Rousset. « Design and implementation of an Object-Oriented strongly typed languages for distributed applications ». *Journal of Object Oriented Programming* **3**, numéro 3 (septembre 1990), 11–22.
- [83] K. Langendoen, R. Bhoedjang et H. Bal. « Models for Asynchronous Message Handling ». *IEEE Concurrency* **5**, numéro 2 (avril 1997), 28–38.
- [84] D. Lazure. « Faisabilité de l'implémentation de PVC sur Transputer sous Helios ». Mémoire de DEA informatique, Université des Sciences et Technologies de Lille, septembre 1990.
- [85] B. Le Cun. *Structures de données parallèles*. Thèse de doctorat, Université Pierre et Marie Curie – Paris VI, janvier 1996.
- [86] M. Lewis et A. Grimshaw. « The Core Legion Object Model ». Dans *Proc. of the Fifth IEEE International Symposium on High Performance Distributed Computing* (1996), IEEE Computer Society Press.
- [87] M. MacBeth, K. MacGuigan et P. Hatcher. « Executing Java Threads in Parallel in a Distributed-Memory Environment ». Dans *Proceedings of the IBM Centre for Advanced Studies Conference* (Toronto, Canada, novembre 1998). see also <http://www.cas.ibm.com/cascon>.
- [88] E. Mascarenhas et V. Rego. « Ariadne : Architecture of a portable threads system supporting mobile processes. ». Rapport technique CSD-TR-95-017, Purdue University, mars 1995.
- [89] C. E. McDowell et D. P. Helmbold. « Debugging Concurrent Programs ». *ACM Computing Surveys* **21**, numéro 4 (décembre 1989), 593–622.
- [90] J.-F. Méhaut. *Implantation du système réparti OMPHALE*. Thèse de doctorat, Université des Sciences et Technologies de Lille, février 1989.
- [91] N. Melab, E. Talbi et S. Petiton. « A parallel adaptive version of the block-based Gauss-Jordan algorithm ». Dans *Proc. of IEEE International Parallel Processing Symposium (IPPS'99)* (San Juan, Puerto Rico, avril 1999), J. Rolim, éditeur, pp. 350–354.
- [92] F. Mueller. « A Library Implementation of POSIX Threads under UNIX. ». Dans *Proceedings of the USENIX Conference* (1993), pp. 29–41.
- [93] F. Mueller. « On the design and implementation of DSM-Threads ». Dans *Proc. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '97)* (Las Vegas, NV, juin 1997), pp. 315–324.
- [94] F. Mueller, E. Giering et T. Baker. « Implementing Ada 9X features using POSIX Threads : Design Issues ». *TRI-Ada* (septembre 1993).

- [95] MYRICOM, INC. *The GM Message Passing System*, décembre 1998. Documentation technique accessible à <http://www.myri.com/GM/index.html>.
- [96] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières*. Thèse de doctorat, Univ. de Lille 1, janvier 1997.
- [97] R. Namyst, Y. Denneulin, J.-M. Geib et J.-F. Méhaut. « Utilisation des processus légers pour le calcul parallèle distribué : l'approche PM2 ». *Calculateurs Parallèles, Réseaux et Systèmes répartis* **10**, numéro 3 (juillet 1998), 237–258.
- [98] S. Pakin, V. Karamcheti et A. Chien. « Fast Messages : Efficient, Portable Communication for Workstation Clusters and MPPs ». *IEEE Concurrency* **5**, numéro 2 (avril 1997), 60–73.
- [99] S. Pakin, M. Lauria et A. Chien. « High Performance Messaging on workstations : Illinois Fast Messages (FM) for Myrinet ». Dans *Proceedings of Supercomputing '95* (San Diego, California, décembre 1995). Accessible à <http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps>.
- [100] C. Perez. *Compilation des langages à parallélisme de données : gestion de l'équilibrage de charge par un exécutif à base de processus légers*. Thèse de doctorat, ENS Lyon, décembre 1999.
- [101] P. Pierce et G. Regnier. « The Paragon [TM] Implementation of the NX message passing interface ». Dans *Proceedings of Scalable High Performance Computing Conference* (Knoxville, TN, mai 1994), IEEE Computer Society Press, pp. 184–190.
- [102] J.-M. Place. *Les performances d'une architecture d'ordinateur destinée au support de la communication par message*. Thèse de doctorat, Université des Sciences et Technologies de Lille, février 1990.
- [103] B. Planquelle et J.-F. Méhaut. « Communications multi-protocoles et réseaux à haut débit ». Dans *11es Rencontres francophones du parallélisme (RenPar 11)* (IRISA, Univ. Rennes 1, Rennes, INRIA, juin 1999), pp. 73–78.
- [104] B. Planquelle, J.-F. Méhaut et N. Revol. « Multi-protocol communications and high speed networks ». Dans *Euro-Par '99 : Parallel Processing* (Toulouse, France, août 1999), volume 1685 des *Lecture Notes in Computer Science*, Springer-Verlag, pp. 139–143.
- [105] B. Planquelle, J.-F. Méhaut et N. Revol. « Multi-Cluster Approach with PM2 ». Dans *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99), Technical Session on Cluster Computing Technologies, Environments and Applications* (Las Vegas, NV, juin 1999), volume 2, pp. 779–785.
- [106] F. Potter. *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs*. Thèse de doctorat, Université Pierre et Marie Curie – Paris VI, Avril 1996.
- [107] L. Prylli et B. Tourancheau. « Protocol design for high performance networking : A Myrinet experience ». Rapport technique RR97-22, Laboratoire d'Informatique du Parallélisme, École Nationale Supérieure de Lyon, 69364 Lyon Cedex 07, July 1997.
- [108] L. Prylli et B. Tourancheau. « BIP : A new protocol designed for High-Performance networking on Myrinet ». Dans *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)* (Orlando, USA, mars 1998), volume 1388 des *Lecture Notes in Computer Science*, Held in conjunction with IPPS/SPDP 1998. IEEE, Springer-Verlag, pp. 472–485.
- [109] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr et R. Sanzi. « Mach : A Foundation for Open Systems ». Dans *Proceedings of the Second Workshop on Workstation Operating System, WWOS2* (septembre 1989). Accessible à : <ftp://mach.cs.cmu.edu/doc/published/intro.ps>.

- [110] C. René et T. Priol. « MPI Code Encapsulation using Parallel CORBA Object ». Dans *Proc. of the IEEE International Symposium on High Performance Distributed Computing* (Redondo Beach, California, août 1999).
- [111] R. v. Renesse, H. v. Staren et A. Tanenbaum. « Performance of the Amoeba distributed operating system ». *Journal of Software Practice and Experience* **19** (mars 1989), 223–234.
- [112] O. Reymann. *Support d'exécution parallèle fondé sur des mécanismes de mémoire distribuée virtuellement partagée*. Thèse de doctorat, ENS Lyon, juillet 1999.
- [113] J.-F. Roos. *Mise au point d'applications distribuées pour environnement de développement basé sur une technologie objet*. Thèse de doctorat, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, février 1994.
- [114] J.-F. Roos, L. Courtrai, J.-M. Geib et J.-F. Méhaut. « Les Composants Actifs de Communication : manuel de programmation (V 2.0) ». Rapport technique ERA-115, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, septembre 1992.
- [115] J.-F. Roos, L. Courtrai et J.-F. Méhaut. « Execution Replay of Parallel Programs ». Dans *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing* (janvier 1993), IEEE, pp. 429–434.
- [116] C. Roucairol. « Stratagème : une méthodologie de programmation parallèle pour les problèmes non structurés ». Rapport de Recherche, PRiSM, Versailles, décembre 1995. Accessible à : http://torquenada.prism.uvsq.fr/rapports/1997document_1997_4.ps.gz.
- [117] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Hermann, C. Kaiser, S. Kanglois, P. Leonard et W. Neuhauser. « The Chorus Distributed Operating System ». *Computing Systems* **1**, numéro 4 (1988), 305–370.
- [118] R. Russell et P. Hatcher. « Efficient kernel support for reliable communications ». Dans *13th ACM Symposium on Applied Computing* (Atlanta, GA, février 1998).
- [119] A. Skjellum et T. McMahon. « Interoperability of Message-Passing Interface (MPI) Implementations : A Position Paper ». Rapport technique, MPI Software Technology, Inc., décembre 1997. Accessible à <http://www.mpi-softtech.com/publications/interop121897.ps>.
- [120] J. Soula. « Support d'exécution multi-threadé pour le parallélisme de données dynamique et irrégulier ». Dans *Actes des 9èmes Rencontres francophones du parallélisme (RenPar 9)* (Lausanne, Suisse, mai 1997), A. Schiper et D. Trystram, éditeurs, École Polytechnique Fédérale de Lausanne.
- [121] M. Steen, A. Tanenbaum, I. Kuz et H. Sips. « A scalable middleware solution for advanced Wide-Area Web services ». Dans *Proc. of Middleware'98* (The Lake District, UK, septembre 1998).
- [122] SUN MICROSYSTEMS. *Systems Services Overview Chapter 6 : Lightweight Processes*, 1988. Documentation technique du système SunOS.
- [123] V. Sunderam. « PVM : A framework for Parallel Distributed Computing ». *Concurrency : Practice and Experience* **2**, numéro 4 (1990), 315–339.
- [124] E. Talbi, Z. Hafidi et J.-M. Geib. « Parallel Tabu search for large optimization problems ». Dans *Meta-heuristics - Advances and Trends in Local Search Paradigms for Optimization* (1999), S. Voss, S. Martello, I. Osman et C. Roucairol, éditeurs, Kluwer Academic Press, pp. 345–358.

- [125] T. von Eicken, A. Basu, V. Buch et W. Vogels. « U-Net : A User-Level Network Interface for Parallel and Distributed Computing ». Dans *Proc. of 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, décembre 1995).
- [126] T. von Eicken, D. Culler, S. Goldstein et K. Schauser. « Active Messages : a mechanism for integrated communication and computation. ». Dans *Proc. 19th Int'l Symposium on Computer Architecture* (mai 1992).
- [127] A. Yonezawa et M. Yokote. *Object-Oriented Concurrent Programming*. The MIT Press, 1987.