Méthodes et outils pour les communications dans les applications de calcul distribué

Tawfik Es-Sqalli

▶ **To cite this version:**

Tawfik Es-Sqalli. Méthodes et outils pour les communications dans les applications de calcul distribué. Informatique [cs]. Université Henri Poincaré - Nancy 1, 2000. Français. NNT : 2000NAN10141 . tel-01754352

HAL Id: tel-01754352
https://hal.univ-lorraine.fr/tel-01754352

Submitted on 30 Mar 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

53 125

SCN2000 141B

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Méthodes et outils pour les communications dans les applications de calcul distribué

S.C.D. - U.H.P. NANCY 1 SCIENCES BIBLIOTHÈQUE

THÈSE

présentée et soutenue publiquement le 13 Novembre 2000

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Tawfik Es-sqalli

Composition du jury

Président : Claude Godart

Rapporteurs : Claude Godart, Professeur, Université Henri Poincaré, Nancy 1
Michel Trehel, Professeur, Université Franche-Comté, Besançon
Xavier Rousset de Pina, Professeur, INPG-Enserg, Grenoble

Examinateurs : Jacques Guyard, Professeur, Université Henri Poincaré, Nancy 1
[illegible] recherche, INRIA-Lorraine

BIBLIOTHEQUE SCIENCES NANCY 1

D 095 139505 1

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

53125

Remerciements

Je tiens à remercier toutes les personnes qui ont rendu cette thèse possible par leur aide et leurs contributions.

Mes premiers remerciements sont adressés aux membres du jury pour l'intérêt qu'ils ont porté à ce travail. Un merci tout particulier à Jacques Guyard pour son encadrement, son soutien et son respect tout au long de mes trois années de thèse.

Je remercie également Eric Fleury qui s'est montré intéressé par mon travail et a apporté de nombreux conseils qui m'ont permis d'approfondir ma réflexion sur les différents sujets traités.

J'adresse mes sincères remerciements à André Schaff de m'avoir accepté au sein de son équipe, pour sa gentillesse, son encouragement et la motivation qu'il insuffle à l'ensemble de son équipe.

Merci aussi aux autres membres de mon équipe pour leur soutien, en particulier à Eric Dillon avec qui j'ai commencé mes premiers travaux, à Carlos, Olivier ainsi qu'à mes collègues de bureau à différentes époques : Emmanuel, Philippe et Ghassan. Un merci spécial à Josiane, notre sympathique secrétaire.

Merci à tous mes amis du LORIA pour leur soutien et surtout pour les matches de baby-foot :-) qu'on a joués ensemble.

Enfin, un grand merci à mes parents, mes proches et toute ma famille pour m'avoir soutenu tout au long de mes études.

Je dédie cette thèse à mes parents,
à mes frères et sœurs,
à toute ma famille, en particulier la nouvelle née, ma nièce Zineb,
à mes fidèles amis.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

1 Contexte général

Les progrès en matière de technologie constituent un élément dominant en informatique depuis plusieurs années. En effet, l'apparition de processeurs de plus en plus petits et puissants, et de réseaux de transmission de données très rapides et fiables ont conduit les concepteurs de machines à imaginer des machines fort variées, s'éloignant notablement de la machine classique de *Von Neumann*. En particulier, il est devenu courant de faire coopérer plusieurs machines dans une même architecture, ceci essentiellement pour améliorer les performances obtenues grâce au fonctionnement simultané de plusieurs machines, c'est le domaine du parallélisme et des architectures parallèles.

Ce domaine a renouvelé profondément la discipline informatique, non seulement par les performances qu'il procure, mais aussi par la nécessaire prise en compte de dimensions supplémentaires : la concurrence, la communication et la coopération. Par conséquent, de nombreux domaines de recherche ont émergé ; par exemple, les langages parallèles, la parallélisation automatique, les architectures distribuées, les environnements de programmation, l'analyse de la complexité, l'algorithmique parallèle.

Aujourd'hui, son utilisation dans les domaines des gros consommateurs de calcul a commencé à se banaliser, notamment pour la majorité des applications de simulation de phénomènes physiques, les problèmes d'analyse financière, les bases de données, les serveurs de fichiers, le traitement d'images, et bien d'autres. Cependant, la programmation parallèle demeure une tâche difficile et fait intervenir de nouvelles approches algorithmiques très différentes de celles séquentielles. Ce problème est renforcé par la difficulté d'adéquation des modèles de programmation parallèles aux besoins des applications et des utilisateurs et par le manque de standardisation des environnements de programmation parallèle.

2 Les communications dans les applications parallèles

La résolution d'un problème en utilisant plusieurs processeurs implique inévitablement la distribution de données, l'échange des résultats intermédiaires ou la diffusion de solutions de sous-problèmes. Toutes ces opérations nécessitent des communications qui apparaissent ainsi comme un point crucial dans le calcul parallèle. Leur réalisation fait appel à plusieurs modèles de communication parallèles dont les plus utilisés sont :

- la communication par échange de messages ;
- l'utilisation des langages data-parallèles ;
- la communication par files de contrôle « thread » et mémoire partagée ;
- les systèmes de mémoire virtuellement partagée.

Chacun de ces modèles est orienté vers une architecture matérielle définie, et propose ses propres supports logiciels de communication qui à leur tour fournissent plusieurs primitives de communication. Face à cette diversité de modèles et à leur richesse, la phase d'implantation des communications est devenue ardue. Elle nécessite une connaissance, voir une maîtrise, des primitives de communication utilisées pour atteindre de bonnes performances. Ce qui rend leur utilisation souvent limitée à une communauté restreinte de spécialistes. En effet, il n'existe pas actuellement un outil de communication qui soit capable d'exploiter une part substantielle de la puissance des machines parallèles, sans un effort que peu d'utilisateurs sont prêts à fournir.

Un autre problème vient du fait que les infrastructures de communication parallèles ne cessent de se développer et de nouveaux modèles ont vu le jour. De ce fait, l'utilisateur est amené à suivre cette évolution pour profiter davantage de ce domaine. Cependant, à chaque fois que l'infrastructure matérielle ou logicielle de communication est modifiée (passage d'une machine mémoire partagée à une machine mémoire distribuée, changement de la bibliothèque de communication, etc.), il faut souvent réadapter le code à cette nouvelle infrastructure.

L'un des thèmes de recherche du projet *RESEDAS* (*Concepts et outils logiciels pour les télécommunications et les systèmes distribués*) est de libérer l'utilisateur de toutes contraintes d'ordre matériel ou logiciel liées aux communications dans les applications parallèles, et lui garantir de bonnes performances. Ce travail de recherche s'est traduit par la définition d'un nouveau formalisme appelé le langage `MeDLey` (*Message Defintion Language*) dont l'approche originale est de décrire tout ce qui concerne les échanges entre les tâches d'une application parallèle. À partir de cette description, un outil dérive de façon automatique une implantation des échanges sur le support de communication choisi par l'utilisateur.

3 L'apport original de la thèse

Le travail de thèse a été guidé par le besoin de faire évoluer le langage `MeDLey` pour couvrir un champ d'application plus vaste et de l'expérimenter sur des applications réelles.

Nous focalisons nos recherches sur les architectures parallèles à mémoire distribuée, en particulier celles composées d'un réseau de stations. Dans ce type d'architecture, la communication entre les tâches est explicite et le modèle de programmation le plus utilisé est celui du passage de messages (*Message Passing*). Sa mise en œuvre s'appuie essentiellement sur l'utilisation de bibliothèques telles que *PVM* ou *MPI* (voir chapitre 2) qui fournissent les fonctions de communication entre processus.

Dans ce contexte, une première version du langage a été développée et s'appuie sur une implantation de communications en utilisant la bibliothèque d'échange de messages *MPI*. La première partie du travail de thèse nous a conduit a étendre cette version de `MeDLey`, d'abord pour permettre des communications collectives, puis pour utiliser les méthodes de décomposition de domaine.

Une autre manière d'implanter les échanges entre les tâches d'une application parallèle est de modéliser ces tâches par des objets dans un système d'objets distribués. La communication entre ces tâches se fait via des invocations de méthodes sur ces objets basées sur l'approche *put/get*. C'est l'idée de base de la deuxième partie de notre travail dont le but est de concevoir une nouvelle approche du langage `MeDLey` qui repose sur l'utilisation de l'*ORB* (*Object Request Broker*) de *CORBA* (*Common Object Request Broker Architecture*) comme support de communication.

Ce choix est justifié d'une part par les performances de *CORBA* au niveau du temps de communication au vu des tests que nous avons réalisés ; et d'autre part par les avantages de cet environnement en terme de services, en particulier celui d'interopérabilité.

L'implantation de cette extension de MeDLey nous a conduit au développement d'une bibliothèque d'échange de messages dans un environnement *CORBA* appelée *MPC* (*Message Passing in CORBA*). Cette bibliothèque repose sur l'utilisation du mécanisme d'invocation de méthode sur des objets distants du système *CORBA*.

L'objectif d'une telle bibliothèque est, d'une part, de profiter des avantages de *CORBA* en terme d'interopérabilité et de performance des communications tout en masquant la difficulté de son utilisation en proposant une interface « message passing » semblable à celles des bibliothèques d'échange de messages existantes, et dont la plupart des utilisateurs du parallélisme sont habitués, notamment *MPI* et *PVM* ; et d'autre part, offrir une variété des primitives de communications utilisées dans le cadre du calcul parallèle et qui ne sont pas directement disponibles par l'utilisation du mécanisme d'invocation de méthode de *CORBA*. Ces primitives couvrent en particulier les communications point à point et celles collectives ainsi que la synchronisation.

Enfin, nos derniers travaux ont porté sur la validation et l'expérimentation des outils développés dans le cadre de cette thèse.

4 Plan

Ce rapport de thèse est divisé en cinq parties :

La première partie, composée de trois chapitres, est consacrée principalement à l'état de l'art :

Nous commençons par présenter globalement le domaine du parallélisme (chapitre 1), nous présentons les motivations pour le parallélisme et faisons un rapide tour d'horizon des différents architectures et modèles de programmation parallèles. Nous mettons l'accent également sur l'évolution de ce domaine en termes d'architectures et de modèles.

Nous détaillons ensuite au chapitre 2 le paradigme de programmation par échange de messages. Nous distinguons les différents modèles et primitives de communication, et nous terminons par une présentation de quelques bibliothèques de communication.

L'objectif du troisième chapitre est de passer en revue les composants de *CORBA* et le modèle de communication associé. Nous abordons également les avantages et les principes de programmation d'une application dans un environnement réparti à objets de type *CORBA*.

La seconde partie présente le contexte dans lequel s'inscrit notre travail :

Nous abordons au chapitre 4 la problématique du besoin d'environnements pour le développement des applications de calcul distribué et parallèle. Nous présentons brièvement l'environnement développé dans l'équipe pour aider l'utilisateur à développer des applications parallèles basées sur des modèles de programmation à mémoire distribuée et à les exécuter en utilisant au mieux les ressources machine et réseau disponibles.

Un des composants majeur de cet environnement est le langage MeDLey destiné à spécifier les communications pour des applications parallèles basées sur le paradigme de communication par échange de messages. Nous présentons au chapitre 5 l'approche originale de ce langage et un résumé de sa syntaxe.

La troisième partie présente les extensions apportées au langage MeDLey :

Dans le chapitre 6, nous traitons les communications collectives et l'extension de MeDLey permettant de couvrir les besoins pour réaliser ce type de communication dans un modèle SPMD.

La deuxième partie de l'extension du langage MeDLey a concerné les méthodes de décomposition de domaine. Dans le chapitre 7, nous présentons le principe de ces méthodes et l'extension de MeDLey proposée dans ce domaine et la façon abordée pour la parallélisation sur des architectures parallèles à mémoire distribuée.

La quatrième partie de ce rapport présente l'utilisation de MeDLey dans un environnement *CORBA*:

Nous étudions dans le chapitre 8 les avantages de *CORBA* en terme de communication et de services et comment les applications parallèles peuvent en tirer profit. Nous présentons également les différents tests de performances de communications de quelques bus *CORBA*.

Nous présentons dans le chapitre 9 l'approche proposée pour spécifier les communications avec le langage MeDLey dans un environnement *CORBA*. Nous distinguons les différentes étapes de construction d'une applications reposant sur cette approche.

Dans le dernier chapitre de cette partie (chapitre 10), nous présentons la bibliothèque *MPC*. Nous décrivons son interface de communication et les protocoles de communication mis en œuvre pour l'implanter.

La dernière partie de ce rapport est consacrée à la validation et à l'expérimentation des outils développés :

Nous présentons dans le chapitre 11 l'expérimentation du langage MeDLey basée sur l'implantation du code « Vlasov », un code de simulation numérique utilisé dans le domaine de physique des plasmas.

Enfin, dans le dernier chapitre de ce rapport (chapitre 12), nous utilisons le langage *TLA+* pour la validation formelle du protocole de communication point à point de la bibliothèque *MPC*.

Première partie

Les systèmes parallèles et distribués : de l'architecture aux modèles

Chapitre 1

Parallélisme et architectures parallèles

Dans ce chapitre, nous présentons les motivations pour le parallélisme et faisons un rapide tour d'horizon des différents architectures et modèles de programmation parallèles. Nous mettons l'accent également sur l'évolution de ce domaine en termes d'architectures et de modèles.

1.1 Introduction : pourquoi le parallélisme ?

L'apparition sur le marché des machines multiprocesseurs et des réseaux de stations de travail a engendré un regain d'intérêt pour la programmation parallèle. Sous le vocable « Parallélisme » se sont regroupés tous les projets susceptibles de mettre à profit ces nouveaux matériels. Cela a créé une certaine confusion car les raisons, ainsi que les méthodes utilisées, pour réaliser des systèmes parallèles, peuvent être très différentes, voir antagonistes. Mais d'une manière générale, on peut citer deux grandes raisons motivant l'utilisation des systèmes parallèles :

1. amélioration des performances de calcul : c'est certainement la première motivation qui a poussé les scientifiques à concevoir des machines parallèles ;
2. résolution des problèmes de grandes tailles : les problèmes gourmands en place mémoire posent une difficulté majeure aux programmeurs et aux constructeurs car la réalisation de machines possédant de très grandes mémoires est coûteuse. Il peut être plus simple et plus rentable de construire une machine contenant 1024 processeurs ayant chacun 128 Mbytes de mémoires qu'une machine séquentielle possédant plus de 130 Gbytes de mémoire centrale ;

Ces deux motivations ne sont, en principe, pas incompatibles. Toutefois, chacune d'elles conduit à des solutions informatiques différentes, puisqu'on ne dispose pas dans le domaine du parallélisme d'un modèle général de machine qui soit l'équivalent de ce qu'est la machine de *Von Neumann* pour le calcul séquentiel. De même, il n'existe pas de modèle formel universel permettant de faire des preuves de calculabilité et de complexité, analogue au modèle séquentiel de *Turing*.

La classification des machines et applications parallèles repose sur trois thèmes souvent complémentaires :

- architecture ;
- modèle d'exécution ;
- modèle de programmation.

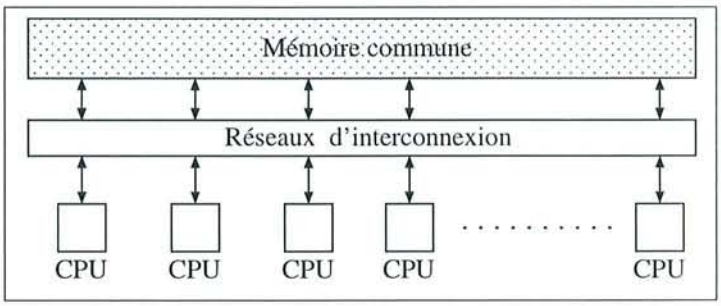

FIG. 1.1 – *Les multiprocesseurs à mémoire partagée*

1.2 Les architectures parallèles

L'architecture désigne la façon dont la machine physique est organisée. Elle regroupe des critères tels que les techniques d'interconnexion des processeurs, l'organisation de la mémoire, les mécanismes de synchronisation disponibles et la disposition des unités de calcul et des unités de contrôle. La grande diversité des machines parallèles appelle à une classification de ces dernières selon ces critères. Il existe de nombreuses taxonomies des machines parallèles parmi lesquelles on trouve :

- les machines vectorielles multiprocesseurs ;
- les multiprocesseurs à mémoires distribuées ;
- les machines synchrones.

1.2.1 Les machines vectorielles multiprocesseurs

Ces machines sont développées à partir des machines vectorielles monoprocesseur des années 70 [Gengler 96]. Les processeurs sont très puissants, 50-300 Mflops et plus pour les machines les plus récentes, mais en faible nombre. Toutes ces machines disposent d'une mémoire partagée accessible à tous les processeurs (figure 1.1). Ces machines peuvent atteindre des puissances de 1 à 100 Gflops, mais elles sont très chères et n'atteignent en fait jamais des rapports coût/performance vraiment intéressants.

1.2.2 Les multiprocesseurs à mémoires distribuées

Ces machines se caractérisent par l'utilisation de processeurs ordinaires en grand nombre, 16 à 1024 ou plus. Chaque processeur dispose d'une mémoire locale. La communication entre processeurs se fait par envois de messages à travers des réseaux de communication (figure 1.2). La caractéristique essentielle de ces machines se situe dans l'autonomie accordée à chaque processeur. Tous les processeurs disposent de leur propre séquenceur et peuvent exécuter différents programmes indépendamment les uns des autres. Entre elles, ces machines se distinguent surtout par leurs réseaux de communication. Les multiprocesseurs à mémoires distribuées sont, en général, peu chers et peuvent atteindre des rapports coût/performance très intéressants. Citons, dans cette classe, les machines basées sur le *Transputer*, les *IPSC*, le *FPS*, le *Supernode*, le *T3D*, le *Paragon* et le *SP-2*.

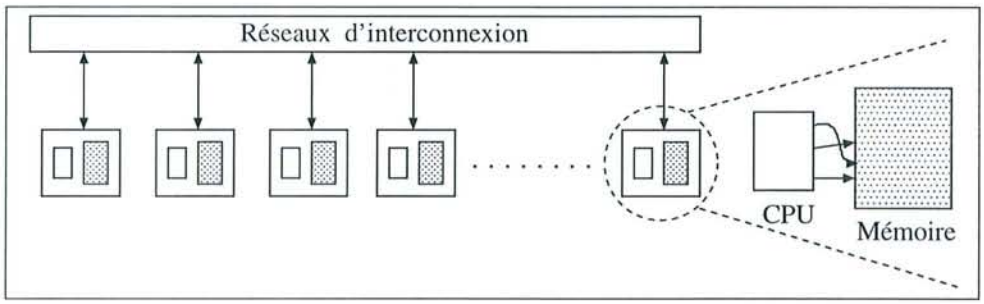

FIG. 1.2 – *Les multiprocesseurs à mémoire distribuée*

1.2.3 Les machines synchrones

Les machines de ce troisième type sont les descendantes directes de l'*ILLIAC IV* [Gengler 96]. La plus connue d'entre elles est la *Connexion Machine 2*. Ces machines comportent un très grand nombre d'éléments de calcul, de 4096 à 65536, qui ont de faible puissance, de 1 à 4 bits, et qui disposent tous d'une petite mémoire locale. Tous ces éléments de calcul sont alimentés par un unique séquenceur et exécutent de manière synchrone une même instruction sur des données différentes se trouvant dans leurs mémoires locales respectives.

1.3 Evolutions des architectures parallèles

1.3.1 Machines à mémoire distribuée virtuellement partagée

Les deux classes d'architectures qui s'opposent actuellement dans le domaine de calcul parallèle sont les machines vectorielles multiprocesseurs et les multiprocesseurs à mémoire distribuée. La première classe, dans laquelle tous les processeurs accèdent à une seule mémoire physique, est la plus ancienne. Malheureusement, en dépit de quelques tentatives [Cheriton 91] la plupart de ces machines restent limitées en nombre de processeurs en raison de la centralisation des accès mémoire. Par contre leur programmation est facile et toutes les phases de communications sont transparentes pour l'utilisateur. Chaque processus accède aux données à tour de rôle, d'où le risque de goulots d'étranglement.

A l'opposé, les architectures à mémoire distribuée peuvent contenir un grand nombre de processeurs et permettre ainsi d'obtenir de bonnes performances. Leur handicap réside dans la difficulté de les utiliser efficacement : toutes les communications doivent être programmées (et optimisées!) par l'utilisateur. C'est la raison pour laquelle ce marché est resté longtemps limité au monde scientifique, gros consommateur de puissance de calcul et peu rebuté par les difficultés liées à la programmation parallèle [Guide 96].

Pour pallier ces difficultés, en 1986, *Kai Li* publie les premiers travaux concernant un nouveau système de programmation parallèle : une mémoire virtuellement partagée sur une architecture parallèle à mémoire distribuée [Li 86]. C'est un compromis entre les deux architectures citées auparavant. Fonctionnellement, ces systèmes essaient de combiner la facilité de programmation des machines à mémoire partagée avec l'efficacité des machines à mémoire distribuée. Un exemple de cette classe de machine est l'*Origin 2000* de *SGI*.

Un système de mémoire distribuée virtuellement partagée peut être vu de deux manières différentes (figure 1.3):

- niveau architectural : on s'appuie sur des architectures à mémoire distribuée en raison de leur potentiel en termes d'efficacité et d'extensibilité ;

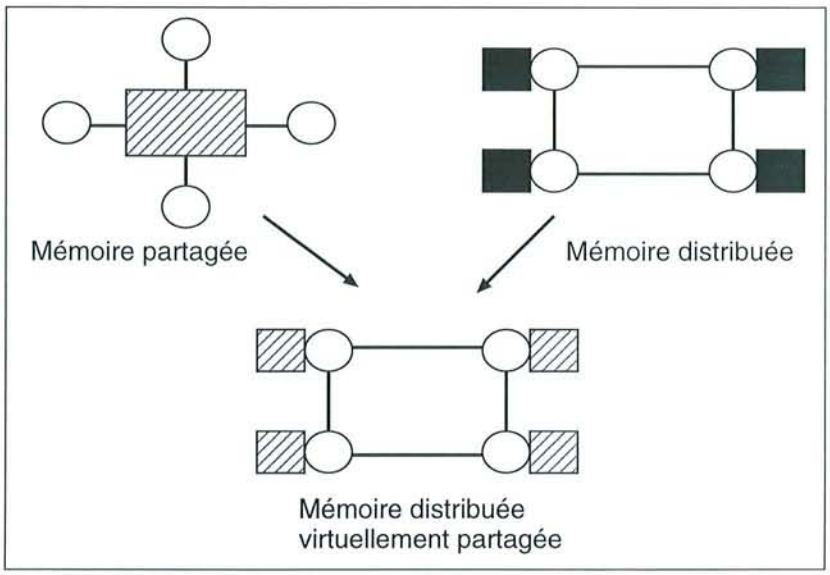

FIG. 1.3 – *Aboutir à une mémoire distribuée virtuellement partagée*

- niveau logiciel : on donne l'impression à l'utilisateur qu'il travaille avec une (importante) mémoire partagée entre tous les processeurs. Il n'a ainsi plus à se préoccuper de la localisation de ses données ni des communications entre les processeurs.

Une architecture à mémoire distribuée virtuellement partagée propose donc une vue unifiée de différentes zones de mémoires distribuées entre les processeurs. Ainsi, les processus accèdent à des zones mémoire distantes sans prendre en compte la localisation des données ni les moyens mis en œuvre pour y accéder. Les processus utilisent la mémoire sans savoir si elle leur appartient réellement [Lefevre 97].

1.3.2 Machines parallèles virtuelles : "network-computing"

Le domaine du parallélisme s'est étendu à une nouvelle architecture : les réseaux (homogènes ou hétérogènes) de stations de travail ou *NOW* (*Network Of Workstations*). On obtient ainsi une machine parallèle virtuelle. Un *NOW* est donc un agrégat de machines autonomes communicant au travers d'un réseau, ayant en plus des services de coopération.

L'idée d'utiliser un réseau de stations de travail pour exécuter des applications parallèles est née de deux constats. Le premier concerne la nécessité d'utiliser, dans un nombre croissant de domaines, le calcul parallèle pour la résolution de problèmes scientifiques. Cependant, l'emploi d'une machine parallèle classique reste trop onéreux pour de nombreux utilisateurs. Le *NOW* fournit alors une alternative sérieuse pour le traitement d'un certain nombre d'applications parallèles. Le second constat est lié plus particulièrement au type d'activités effectuées sur les stations de travail. En effet, ces dernières sont affectées principalement à une utilisation en mode interactif et restent donc sous utilisées pendant la nuit et les fins de semaine. Certaines études montrent, d'ailleurs, que même pendant les heures ouvrables, entre 60% et 90% des ressources des stations restent inutilisées [Cap 93] [Litzkow 88]. L'idéal consiste donc à exécuter sur ces stations des tâches habituelles (éventuellement interactives) et des applications parallèles, en évitant de gêner les utilisateurs principaux, mais en satisfaisant le maximum de monde [Anderson 93].

Ce nouveau type d'architecture ressemble néanmoins aux machines parallèles à mémoire distribuée. La différence majeure est liée à la spécialisation des interconnexions dans les machines

parallèles à hautes performances. Tous les aspects de l'architecture interne aux systèmes en passant par les protocoles de communication et le routage sont optimisés pour diminuer le coût de transfert d'un message entre deux noeuds de la machine et cela sans souci du prix élevé de l'ordinateur qui en résulte. En revanche, dans un réseau de stations de travail l'architecture des interconnexions est complexe, non régulière et générale. Aucune optimisation n'existe donc à la base dans un *NOW* pour augmenter les performances du trafic lié au parallélisme. Une autre différence entre une machine parallèle et un *NOW* se situe au niveau du système d'exploitation qui a connaissance de ce regroupement pour former une machine parallèle. De plus, le système d'exploitation possède soit des mécanismes pour gérer directement des programmes parallèles, soit il collabore facilement avec des outils destinés à la gestion des tâches parallèles [LSF 96] [NQE 97].

Pour remédier à ces problèmes, l'architecture des réseaux de stations de travail connaît depuis peu une évolution sur le plan du support d'interconnexion. La nouveauté réside dans l'utilisation de réseaux plus rapides diminuant les coûts de communication et permettant, ainsi, la construction des *NOW* à hautes performances dédiés ou non au calcul parallèle. En effet, l'utilisation des réseaux *ATM* (*Asynchronous Transfer Mode*) [Prycker 95], *FDDI* (*Fiber Distributed Data Interface*) [Jain 94], *Fast Ethernet*, *HiPPI* (*High Performance Parallel Interface*) [Kofman 96] ou *Myrinet* [Boden 95] comme liens de communication a pour conséquence que le débit entre les stations devient extensible (scalable).

Avant d'achever ce paragraphe sur l'évolution des architectures parallèles. Rappelons qu'un nouveau mode d'architecture est né, il s'agit des architectures parallèles hybrides. L'idée de base est de rassembler plusieurs machines parallèles de différentes architectures pour bâtir une machine parallèle hybride. Cependant l'utilisation d'une telle architecture demeure difficile et des recherches actuelles tentent de faciliter le déploiement des applications parallèles sur ces architectures [Globus 00].

1.4 Modélisation des architectures et de la programmation parallèles

1.4.1 Modèles algorithmiques

Nous venons de présenter plusieurs familles d'architectures parallèles. Cependant ces familles ne présentent qu'une vue très abstraite de l'architecture matérielle réellement utilisée. La grande diversité d'architectures et de réseaux d'interconnexion constituant les machines parallèles actuelles a généralement un impact très important sur les performances d'un algorithme.

Chaque architecture possède ses propres particularités desquelles dépendent largement ses performances. Dans le but d'adapter la façon de programmer les différents algorithmes aux différentes architectures, plusieurs modèles ont été élaborés pour décrire de façon de plus en plus détaillée le mode de fonctionnement de chaque machine. Ces modèles sont appelés des modèles algorithmiques. Bien entendu, de tels modèles ne seront intéressants et utilisables que s'ils permettent de décrire de façon précise toutes les architectures matérielles de façon abstraite (i.e. avec un nombre de paramètres peu élevé).

Pour modéliser les différentes architectures parallèles, il faut dégager des paramètres qui serviront de points de comparaison. Généralement des estimateurs sur différentes ressources sont utilisés. Une ressource étant alors définie comme un élément de l'architecture qui affecte les performances de la machine. Plusieurs estimateurs peuvent être envisagés. Parmi les plus utilisés on trouve :

- le nombre de processeurs, P . Ce nombre ne possède pas de bornes à priori ;
- l'organisation mémoire, permettant de préciser si la machine est basée sur une mémoire physiquement partagée ou distribuée ; l'estimation de ce paramètre reste difficile, cependant, les temps d'accès aux différents niveaux de mémoire peuvent être pris en compte ;
- la latence, c'est à dire le coût que représente un accès à une mémoire locale ou globale pour une machine à mémoire partagée ; ou bien le coût d'une communication pour une machine à mémoire distribuée ;
- le degré d'asynchronisme, permettant de dire comment les processeurs exécutent leurs codes les uns vis à vis des autres : de façon asynchrone, semisynchrone, ou complètement synchrone ;
- la bande passante, concernant à la fois les communications et les accès à la mémoire ;
- le surcoût de traitement d'un message pour un processeur, correspondant directement au temps nécessaire à l'envoi et la réception d'un message pour chaque processeur ;
- la hiérarchie mémoire, incluant les registres, les caches, la mémoire principale et secondaire ;
- la topologie du réseau d'interconnexion, que ce soit une grille, un anneau, un bus, etc.

A partir de cet ensemble de paramètres, plusieurs modèles décrivant le fonctionnement d'architectures parallèles ont été décris. Les premiers modèles étaient très simples, mais trop peu réalistes.

Parmi ces modèles, citons le modèle *PRAM* (*Parallel Random AccessMachine*), extension directe du modèle de machine séquentielle. Ce modèle décrit une architecture parallèle comme un ensemble de processeurs séquentiels partageant une mémoire globale. Dans ce modèle, le coût d'accès à la mémoire est uniforme, l'exécution se fait de façon synchrone, instruction par instruction, sur tous les processeurs. Une variante de ce modèle est le modèle *VRAM* décrivant un fonctionnement identique, mis à part que les processeurs utilisés sont vectoriels.

Les premières extensions de ces modèles concernent tout d'abord le degré d'asynchronisme des processeurs. Citons dans ce cadre le modèle *Phase PRAM* [Gibbons 89], qui l'étend avec une exécution semiasynchrone. En effet, une machine *Phase PRAM* est constituée d'une mémoire partagée globale, d'un ensemble de P processeurs possédant chacun une mémoire locale privée. L'exécution se fait sous forme de phases de calcul asynchrones suivies de phases de synchronisation explicite. Les accès mémoires sont toujours considérés comme uniformes. Une extension complètement asynchrone est présenté par le modèle *APRAM* (*Asynchronous PRAM*) [Gibbons 89].

Enfin, les dernières extensions ont intégré des paramètres indispensables pour le réalisme d'exécution sur les différentes architectures. La finesse de ces dernières extensions permet de décrire complètement la façon dont vont se dérouler les algorithmes sur chaque machine. Nous trouvons principalement deux modèles dans ce cadre :

1. le modèle *BSP* (*Bulk-Synchronous Parallel model*) [Skillicorn 96] ;
2. le modèle *LogP* [Culler 93].

Le modèle *BSP* est un modèle décrivant une architecture à mémoire distribuée. A la manière du modèle Phase *PRAM*, il fonctionne de façon semi-asynchrone, les phases de calculs étant ici appelées des « superpas ». Pendant ces « superpas » les processeurs peuvent fonctionner de manière totalement asynchrone ; après chaque « superpas » une synchronisation est effectuée.

Le modèle *LogP*, enfin, s'adresse plus particulièrement à la dernière famille de machines présentée dans les paragraphes précédents, à savoir les réseaux de stations de travail. Au contraire

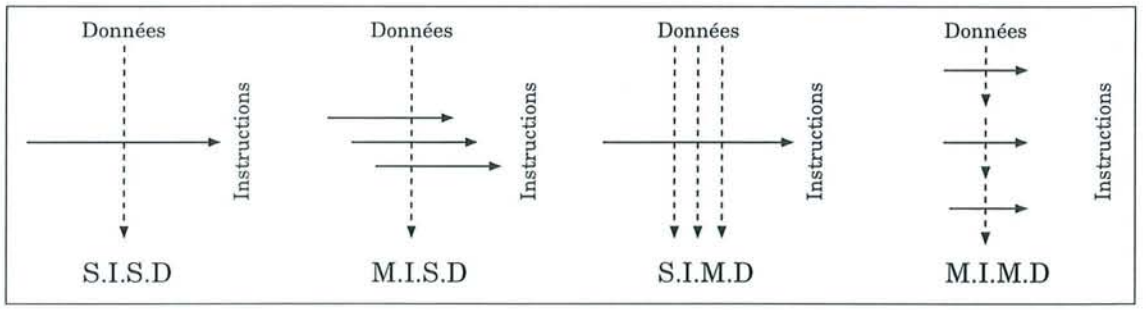

FIG. 1.4 – *Modèles d'exécution proposés par Flynn*

du modèle *BSP*, l'exécution se fait de manière totalement asynchrone correspondant ainsi directement aux réseaux de stations de travail. De plus, ce modèle considère que le réseau a une capacité finie. Cette capacité peut être atteinte si un processeur envoie des données à une vitesse plus rapide que celle à laquelle le destinataire peut les recevoir. Ce dernier modèle reste sans doute le plus réaliste concernant le mode d'exécution et les communications sur les architectures matérielles actuelles. Cependant, il souffre encore d'une limite concernant la quantification des accès mémoires. En effet, le modèle LogP permet d'exprimer de façon très fine le coût de chaque communication, mais les accès mémoires sont toujours supposés uniformes. Or toutes les architectures matérielles récentes sont basées sur l'utilisation intensive de caches mémoires, conduisant à des accès hiérarchisés à la mémoire. [Li 95] proposent de combiner certains modèles décrivant la hiérarchie mémoire comme *HMM* avec le modèle *LogP* pour obtenir un modèle complet. Ce modèle, baptisé *LogPHMM*, décrit une machine constituée d'un ensemble de processeurs asynchrones possédant chacun une mémoire locale non bornée. Cette mémoire locale est hiérarchisée en couches de taille croissante, dont les temps accès sont eux aussi croissant. Les communications entre processeurs suivent le modèle *LogP*.

Enfin, malgré la pertinence et la simplicité de chaque modèle, les utiliser est souvent une tâche fastidieuse. Cependant, certains algorithmes classiques ont déjà été modélisés, notamment avec *BSP*, ce qui permet de se faire une idée rapide des performances d'un programme, à condition que le modèle de programmation laisse suffisamment d'informations sur ce qui se passe réellement à l'exécution.

1.4.2 Modèle d'exécution

Le modèle d'exécution décrit la manière dont les programmes sont exécutés du point de vue du programmeur. La grande diversité des machines parallèles appelle à une classification de ces dernières, afin de pouvoir en dégager les concepts sous-jacents. Il existe de nombreuses classifications des machines parallèles, nous avons donc choisit la plus simple et la plus claire, celle de *FLYNN* [Flynn 95] introduite en 1966. Cette classification est fondée sur deux critères :

La machine a-t-elle un ou plusieurs flux de données? (*Single Data stream ou Multiple Data stream*)

La machine a-t-elle un ou plusieurs flux d'instructions? (*Single Instruction stream ou Multiple Instruction stream*)

Il est possible d'introduire le parallélisme en multipliant les flots d'instructions et/ou les flots de données. Nous obtenons alors les quatre classes des modèles suivantes (figure 1.4) :

Modèle séquentiel

Le modèle de fonctionnement séquentiel, appelé *SISD* (*Single Instruction Single Data*), correspondant à l'architecture dite de « Von Neuman », concerne encore la majorité des ordinateurs commercialisés à l'heure actuelle. Les éléments de base sont la mémoire et le processeur, lui-même constitué d'une unité de contrôle et d'une unité de traitement. L'unité de contrôle lit dans la mémoire les instructions à exécuter, puis envoie des ordres à l'unité de traitement. Celle-ci effectue alors les opérations demandées sur le flux de données en provenance de la mémoire. Le flux traité est ensuite réécrit en mémoire.

Modèle vectoriel

Le modèle vectoriel, appelé également *MISD* (*Multiple Instruction Single Data*), conserve la même structure. La différence est que les unités de traitement et de contrôle sont découpées en étages, chargée chacune d'une partie des opérations à effectuer. Le facteur d'accélération est donc égal (grossièrement) au nombre d'étages des unités. Il est à noter que bien qu'il ne s'agit pas ici de machines multiprocesseurs, le fonctionnement n'en est pas moins parallèle, le parallélisme se situant ici à un niveau plus bas (au sein même du processeur). Cette classe de machines utilise le mode de fonctionnement du travail à la chaîne ou pipeline.

Modèle S.I.M.D

Dans le modèle *SIMD* (*Single Instruction Multiple Data*), seules les unités de traitement sont dupliquées, elles sont commandées par une unité de contrôle unique (ce qui entraîne qu'à un instant donné, elles font toutes la même chose) et sont alimentées par des flux de données différents.

Modèle M.I.M.D

La classe de machines *MIMD*(*Multiple Instruction Multiple Data*) est de très loin la plus importante. Comme dans le cas de la classe SIMD, chaque processeur traite une donnée distincte, mais, en plus, il suit son propre flot d'instructions. Ici la localisation des données divise la classe en deux, les machines à mémoire partagée et les machines à mémoire distribuée.

La classification de *Flynn* est toujours utilisée de nos jours, cependant la plupart des scientifiques s'accordent pour dire qu'elle reste trop pauvre pour rendre compte de la diversité des matériels actuels. Dans ce cadre, une nouvelle classification vient de s'ajouter et qu'il est possible de qualifier de plus gros grain, où il n'est plus question d'instructions mais de programmes :

- *S.P.M.D* (*Single Program Multiple Data*)
- *M.P.M.D* (*Multiple Program Multiple Data*)
- *M-S.P.M.D* : plusieurs familles de programmes (ou tâches) qui traitent des données différentes.

1.4.3 Modèle de programmation

Deux points de vue sont actuellement envisagés pour programmer les machines parallèles. Le premier point de vue est celui du parallélisme implicite : utiliser un langage non spécifique au parallélisme (Fortran, Prolog ou un langage fonctionnel) et générer par compilation un code où le parallélisme est explicite. Le deuxième point de vue est celui du parallélisme explicite au premier abord, où le programmeur subdivise le problème afin de le paralléliser. Cette subdivision

peut être guidée par un découpage de données (parallélisme de données) ou par un découpage des calculs (parallélisme de contrôle et de flux).

Un domaine nouveau semble actuellement s'ouvrir avec l'apparition des modèles de programmation hybrides, permettant de concilier dans un même cadre un parallélisme de contrôle et de données. Mais d'une manière générale, le choix du modèle de programmation est fortement guidé par le matériel utilisé ou par l'application à réaliser.

Parallélisme de données

L'approche du parallélisme de données consiste à découper les données alors que le code de traitement est transformé de façon automatique à l'aide de directives en code parallèle. Cette approche est bien adaptée aux calculs sur des structures de données régulières.

L'expression du parallélisme au niveau des données a l'avantage de conserver un flot unique de contrôle. Un algorithme data-parallèle consiste donc, en une séquence d'instructions élémentaires appliquées à des données scalaires ou vectorielles : une instruction n'est déclenchée que si l'instruction précédente est terminée.

Le parallélisme de données apparaît aujourd'hui comme un modèle de programmation particulièrement adapté à la conception, la compilation et l'optimisation des logiciels massivement parallèles. Ce modèle, qui met l'accent sur la régularité des traitements parallèles, permet de conduire des études très précises sur les comportements des programmes [Gengler 96], et donc de donner naissance à des outils particulièrement performants.

On distribuera, par exemple, une matrice de taille n*n sur p processeurs de manière à ce que chaque processeur possède n/p lignes de la matrice. Le processeur sera le propriétaire (*OWNER*) de ces lignes de la matrice et sera responsable de toutes les opérations les concernant.

Parallélisme de contrôle

L'approche du parallélisme de contrôle consiste à isoler sous forme de modules des calculs qui peuvent se faire en parallèle : dans cette classe on trouve l'approche procédurale (objet, fonctionnelle, etc.) et l'approche des processus ou tâches communiquants. Cette dernière est proche de la réalité physique de l'exécution parallèle.

L'extraction d'un parallélisme de contrôle d'un programme séquentiel implique une première phase de découpage du programme en tâches. Cette découpe peut s'effectuer de manière automatique ou manuelle pour un programme donné. Ensuite on étudie l'ensemble des tâches pour trouver celles qui doivent s'exécuter simultanément et celles qui doivent s'exécuter en séquence l'une par rapport à l'autre. L'exemple le plus simple est le calcul d'une expression comme (a + b) * (c - d). Cette expression comporte deux sous expressions qui peuvent être évaluées indépendamment l'une de l'autre. Le découpage en tâches ne peut évidemment pas se faire de façon quelconque si l'on cherche la plus grande efficacité. Les critères importants à considérer sont le nombre de tâches que l'on veut obtenir, le grain du parallélisme, mais aussi les dépendances entre les différentes tâches générées.

Parallélisme de flux : *pipeline*

Il correspond au concept du travail à la chaîne : une série d'opérations sont effectuées en cascade sur un flux de données.

Ce découpage en trois modèles reflète bien le cheminement entre le modèle d'exécution et le parallélisme associé. En effet, chaque type de parallélisme correspond à l'abstraction d'un

modèle d'exécution : le parallélisme de données correspond aux architectures *SIMD* et *SPMD*, le parallélisme de contrôle au *MIMD* et le parallélisme de flux au *MISD*.

Une conséquence directe de cet état de choses est la bonne adaptation (bonne efficacité d'exécution) entre un modèle de programmation et l'architecture associée. Par exemple un programme data parallel est a priori plus efficace sur une machine *SIMD* que sur une *MIMD*, car les coûts de synchronisation et de communication sont plus faibles dans le premier cas. De plus, la liaison entre le modèle de programmation et le modèle d'exécution entretient la confusion entre les noms des classes de *Flynn* (*SIMD*, *MISD* et *MIMD*) et l'approche de parallélisme associé.

1.5 Conclusion

Plusieurs tentatives de définition d'un modèle général de calcul parallèle ont été faites mais, contrairement à ce qui s'est passé avec les ordinateurs séquentiels et le modèle de calcul de *Von Neumann*, aucune n'a réussi à s'imposer.

Dans la pratique, ce sont les buts poursuivis par l'utilisation de systèmes parallèles qui vont déterminer les moyens, modèles et méthodes utilisés. Du point de vue de l'utilisateur, le parallélisme s'exprime au moyen d'un paradigme de programmation. Ce dernier se base de son côté sur un modèle de programmation et effectue ainsi le lien entre le programme et le modèle de programmation. Parmi les paradigmes de programmation les plus utilisés, citons l'échange de messages (message-passing), la mémoire partagée, l'approche fonctionnelle et la programmation parallèle orientée objets. De nouveau, un paradigme convient plus ou moins bien à un type de parallélisme.

Notons, par ailleurs, qu'une application n'est pas forcément restreinte à un seul type de parallélisme : suivant le sous-problème à résoudre, un parallélisme spécifique peut être choisi. La programmation par combinaison de plusieurs types de parallélismes reste cependant difficile à utiliser, car de nombreux aspects restent encore à la charge de l'utilisateur.

Dans le reste de ce rapport, nous allons nous intéresser plus particulièrement aux machines *MIMD* à mémoire distribuée, notamment au cas des machines parallèles virtuelles réalisées par un réseau de stations. Chaque processeur est donc une entité de calcul autonome, possédant toutes les caractéristiques d'une machine séquentielle, à savoir un processeur et une mémoire. Les processeurs peuvent communiquer entre eux, explicitement, à travers un réseau d'interconnexion. Dans ce cadre, le paradigme de programmation et de communication par passage de messages (*Message Passing*) est le plus répandu. Nous détaillerons ce paradigme dans le chapitre suivant.

Chapitre 2

Paradigme de programmation par passage de messages

Dans ce chapitre, nous analysons les formes que peuvent prendre les communications dans une machine à mémoire distribuée basées sur le paradigme de communication par passage de messages. Nous distinguons les différents modèles et primitives de communication, et nous terminons par une présentation de quelques bibliothèques de communication.

2.1 Introduction

Si les années 1970 avaient vu émerger les réseaux longues distances (*WAN : Wide Area Network*), le débit de ces réseaux (quelques Koctets/s) et leurs prix confinaient leur utilisation au transfert de fichiers et au courrier électronique. Dans les années 1980, l'apparition de réseaux locaux, a permis d'entrevoir de nouvelles possibilités technologiques. Parmi ces possibilités la notion de distribution des calculs. Il s'agit d'utiliser les différents processeurs et mémoires d'un ensemble de machines pour faire du calcul intensif. Cette notion entraîne alors d'autres besoins : comment intégrer aux systèmes d'exploitation des moyens simples de communication qui permettent de gérer cette distribution de calcul? En effet, les systèmes d'exploitation n'ont pas été conçus à l'origine pour intégrer l'aspect réseau de manière transparente pour l'utilisateur. Pour répondre à ce besoin, on a alors ajouté, aux systèmes d'exploitation, des couches logicielles de communication par passage de messages. Les machines participant à une telle distribution sont appelées des machines à passage de messages.

2.1.1 Les machines à passage de messages

Il est à noter que ces machines appartiennent à la classe *NORMA* : « NO Remote Memory Access » (l'accès à une mémoire distant est impossible). Cela signifie d'abord que la mémoire est distribuée sur les processeurs et ensuite qu'il n'existe pas de mécanisme permettant d'adresser les mémoires non locales. La mémoire n'est donc pas partageable. Les espaces d'adressage sont physiquement et logiquement distincts. L'absence de mémoire partagée annule la possibilité d'échanger des informations à travers la mémoire.

C'est donc en utilisant les entrées-sorties et plus particulièrement un dispositif de communication que l'échange d'information peut avoir lieu. Avec une mémoire partagée, les communications sont implicites. Dans le cas d'une machine à mémoire distribuée les accès aux mémoires distantes sont explicites. C'est le programmeur qui doit les spécifier dans son programme en coordonnant

le processeur qui demande l'information distante et celui qui la possède. La spécification des communications impose dans ce cas une contrainte supplémentaire par rapport à la mémoire partagée. Le programmeur peut gérer la distribution des données sur les mémoires. Cette distribution a une influence déterminante sur les performances. Elle détermine le nombre et le volume des communications. Les communications étant très coûteuses, il faut les réduire autant que possible. La recherche d'une distribution favorable devient donc une contrainte qui n'existe pas pour la mémoire partagée.

Comme pour la mémoire partagée, le programmeur doit en plus gérer la distribution des calculs (c'est-à-dire équilibrer les charges de calculs sur tous les processeurs) et contrôler la synchronisation des processeurs. Pour ce dernier point, le programmeur ne dispose pas de variables partagées pour les contrôles d'accès ou les contrôles de séquence. Toutes les synchronisations sont par conséquent réalisées par messages.

2.1.2 Les interfaces de programmation

Un programme parallèle pour une machine à passage de messages est généralement écrit à partir d'un langage séquentiel classique (C, C++, Fortran) et utilise les fonctions des bibliothèques de communication. Ces bibliothèques s'appellent aussi des interfaces de programmation d'applications (*API* : *Application Programming Interface*). Il existe plusieurs *API* pour les machines à passage de messages. Les fonctionnalités assurées par les *API* constituent un deuxième critère de distinction. Il existe principalement deux niveaux d'*API*. Les *API* de haut niveau comme *PVM* [Team 94], *MPI* [Forum 95] [J.Chergui 97], etc. Ces *API* limitent l'intervention du programmeur à l'utilisation des fonctions de communication. L'intégralité des protocoles de communication nécessaires pour la transmission des messages est masquée au programmeur.

D'autres *API*, plus rudimentaires, n'assurent pas certains traitements de protocoles comme le contrôle de flux, le contrôle d'erreur, le réordonnancement ou l'allocation de tampons. Les sockets [Quinton 97], les messages actifs [Thorsten 92] et les « fast messages » [Scott 95], sont des exemples d'*API* de bas niveau. L'objectif de ces *API* est d'abord de fournir un support de développement pour les *API* de plus haut niveau. Leurs performances sont bien meilleures mais le programmeur doit intégrer dans son programme les traitements nécessaires pour son application qui ne sont pas assurés par l'*API*. C'est pourquoi l'utilisation de ces *API* est réservée aux spécialistes du domaine car en matière de communication dans un système distribué, la complexité des phénomènes et la pauvreté des environnements de déverminage (déboggage) rendent la programmation à ce niveau quasiment impossible pour le programmeur non-spécialiste.

Dans le reste de ce rapport, nous nous limitons au modèle de programmation reposant sur les *API* de haut niveau qui sont les plus utilisées dans le cadre des applications du calcul distribué.

2.2 Concepts de base de la programmation par échange de messages

Le paradigme de programmation par échange de messages est principalement destiné au développement d'applications basées sur le parallélisme de contrôle avec une architecture de type *MIMD* à mémoire distribuée.

La programmation dans ce paradigme est basée sur deux étapes distinctes effectuées par le programmeur. La première consiste à découper l'application en tâches s'exécutant en parallèle. La seconde étape consiste à définir les interactions entre ces tâches sous forme de messages explicitement échangés. Nous commençons d'abord par définir les modèles de tâche et de message

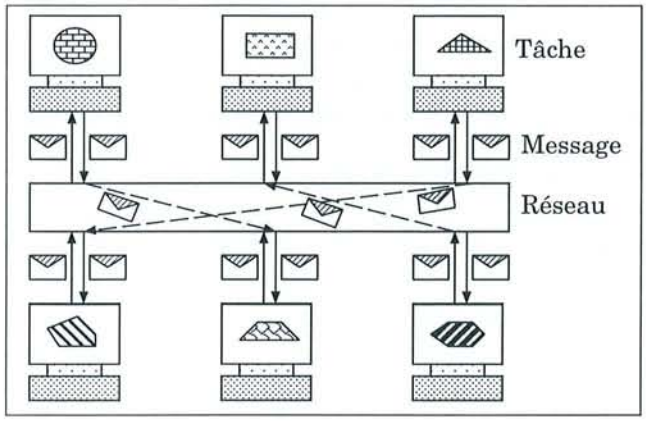

FIG. 2.1 – *Modèles de tâche et de message de la programmation par échange de messages*

sur lesquels ce paradigme ropose (figure 2.1).

2.2.1 Le modèle de tâches

Le service de passage de messages est offert à des tâches qui s'exécutent de façon concurrente. Dans ce contexte, une tâche doit être considérée comme un flot d'exécution, et regroupe donc indifféremment les notions de thread ou de processus.

Lorsqu'on utilise la programmation par échange de messages, on fait implicitement les hypothèses suivantes :

- les données utilisées dans une tâche sont privées (i.e. chaque tâche dispose de son propre espace mémoire) ;
- toute coopération entre tâches doit se faire par le biais d'une communication ;
- toutes les parties participant à une communication doivent appeler explicitement une (resp.des) primitive(s) spécifiques(s) pour que l'échange soit réalisé.

Ces hypothèses signifient que si au cours de l'exécution d'une tâche, son algorithme nécessite l'accès aux données d'une autre tâche, les deux tâches doivent invoquer des primitives de communication de manière à échanger les données en question.

Globalement, on peut donc considérer que chaque tâche s'exécute indépendamment des autres, et que, au fur et à mesure des calculs, elles échangent des informations à l'aide de messages.

La gestion des tâches

Ce paragraphe fait une présentation générale des fonctionnalités des bibliothèques d'échange de messages, en particulier des moyens offerts pour lancer, synchroniser et tuer des tâches concurrentes.

Deux possibilités se présentent concernant le lancement des tâches :

- la gestion statique des tâches : toutes les tâches concurrentes doivent être précisées lors du démarrage de l'application à travers un fichier de configuration. Le nombre de tâches concurrentes ne variera pas pendant toute la durée de l'exécution de l'application ;

- la gestion dynamique des tâches : une ou plusieurs tâches peuvent être lancées par l'utilisateur, mais ces tâches peuvent elles-mêmes relancer d'autres tâches, ou en tuer. Le nombre de tâches impliquées dans l'application peut donc varier au cours de l'exécution de celle-ci.

Pour ce qui est de la synchronisation, chaque bibliothèque d'échange de messages propose en général une fonction permettant de synchroniser tout ou partie des tâches concurrentes. Cette fonction bloque chaque tâche jusqu'à ce que l'ensemble des tâches devant se synchroniser aient toutes appelé cette primitive. La portée de la synchronisation (i.e. le nombre de tâches devant se synchroniser) dépend des bibliothèques d'échange de messages. Généralement la primitive de synchronisation utilise un paramètre pour résoudre ce problème :

1. soit le paramètre est un nombre correspondant au nombre de tâches devant participer à la synchronisation. L'appel de chaque tâche à la primitive de synchronisation a pour effet de décrémenter ce compteur global. Toutes les tâches sont débloquées lorsque le compteur passe à zéro ;
2. soit le paramètre est un identificateur définissant un groupe de tâches. Ce qui signifie que toutes les tâches doivent être bloquées par la primitive de synchronisation tant que l'ensemble du groupe n'a pas appelé cette primitive.

Ces deux méthodes ont leurs avantages et inconvénients. La première permet une gestion très souple des synchronisations, puisqu'il n'est pas nécessaire de préparer cette synchronisation par la création d'une structure spécifique comme un groupe. En revanche, la deuxième possibilité permet que deux groupes de tâches se synchronisent de manière indépendante, ce qui n'était pas possible dans le premier cas.

2.2.2 Le modèle de messages

Le concept de base de la programmation par échange de messages repose sur les messages eux-mêmes. Il est donc important de préciser la forme et le contenu de ceux-ci.

Il convient d'abord de distinguer deux types de messages :

- les messages de données, ou encore messages utilisateurs ;
- les messages de contrôle ou de service, ou encore messages utilisés par le système de façon transparente pour l'utilisateur.

La première catégorie de messages correspond aux messages que l'utilisateur manipule avec les différentes primitives de communications offertes par la bibliothèque d'échange de messages.

La seconde catégorie regroupe tous les messages utilisés par la bibliothèque pour implanter les différents protocoles de communication ou de synchronisation (l'utilisateur ne les voit pas, et il n'a aucun contrôle sur eux). Par exemple, nous trouvons notamment les messages d'acquittement, de synchronisation, de notification d'envoi, etc.

Nous nous limiterons ici aux messages utilisateurs, car l'implantation des messages de service dépend directement de chaque architecture matérielle et logicielle : ces choix d'implantation ne servent qu'à garantir un service d'échange de messages à l'utilisateur.

Un message utilisateur est constitué de deux parties distinctes (figure 2.2) :

1. l'enveloppe du message ;
2. le contenu du message.

L'enveloppe regroupe toutes les informations de service indispensables à l'acheminement et à la réception du message. On y trouve en particulier un champ pour :

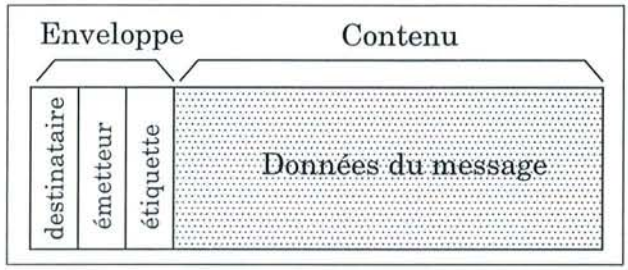

FIG. 2.2 – *Représentation schématique d'un message utilisateur*

- le destinataire du message ;
- l'émetteur du message ;
- l'étiquette éventuelle du message, permettant notamment de distinguer des messages sans en connaître le contenu.

Enfin, la seconde partie regroupe le cœur du message, i.e. la suite d'octets qui doit effectivement être transmise d'une tâche à l'autre. Après avoir introduit les concepts de tâches et de messages, il s'agit maintenant de détailler la liste des primitives de communication entre tâches.

Les communications peuvent être réalisées de diverses manières et offrent différentes propriétés au programmeur. On distingue les primitives de communication simples qui n'impliquent que deux processeurs, appelées des communications point-à-point, et les primitives de communication complexes, qui peuvent concerner un sous-ensemble de processeurs.

Chacun de ces types suit un modèle de communication qui implique ou non de synchronisation entre les tâches participantes. On parle des modèles synchrone/asynchrone. En outre, l'émetteur (resp le récepteur) peut attendre ou non l'achèvement de la communication pour pouvoir accéder à la zone mémoire d'envoi (resp de réception), ce qui permet de distinguer deux autres modèles : bloquant et non bloquant.

2.2.3 Les modèles de communication

Les modèles synchrones et asynchrones

L'envoi d'un message concerne un expéditeur qui réalise l'opération d'envoi et un destinataire qui réalise l'opération de réception [Gengler 96]. Si ces deux opérations sont faites en »même temps » on parle de mode de communication synchrone ou encore de rendez-vous. Ce mode de communication est comparable aux communications par téléphone qui mettent en présence les interlocuteurs. Si, par contre, les opérations d'envoi et de réception se font à des instants quelconques, on parle de mode de communication asynchrone.

La communication par rendez-vous se fait à un moment où les deux sites sont prêts à communiquer (figure 2.3 a). Le rendez-vous engendre souvent des temps d'attente longs. En contrepartie, il permet à l'expéditeur d'être sûr que le message est arrivé au destinataire.

Dans le mode de communication asynchrone (figure 2.3 b,c,d), l'expéditeur envoie son message dès que celui-ci est prêt. La réception de ce message du côté du destinataire se fait plus tard. L'avantage du mode asynchrone consiste à ne jamais faire attendre l'émetteur. Le récepteur doit éventuellement attendre si le message n'est pas encore arrivé au moment où il appelle la primitive de réception. Au contraire, si le message est arrivé avant l'appel de primitive de réception, le destinataire n'a pas besoin d'attendre. Le mode asynchrone engendre toujours des temps d'attente moins importants que le mode synchrone. Mais cette accélération de l'exécution du programme

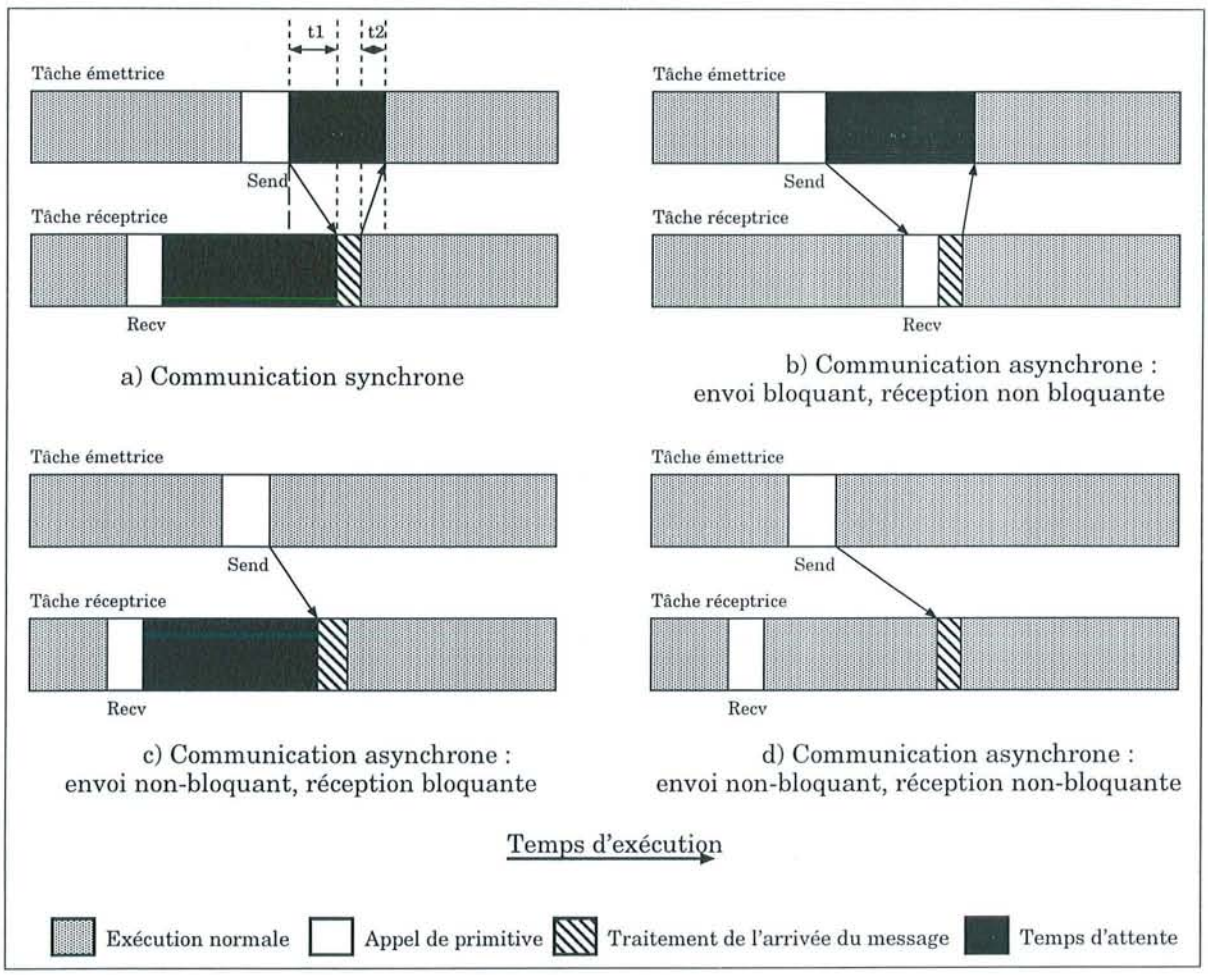

FIG. 2.3 – *Les modèles de communication*

se paie par le fait que l'expéditeur ne sait pas, à un moment donné, si le message est déjà pris en compte par le destinataire. Le modèle asynchrone offre donc des propriétés plus faibles que le modèle synchrone.

Le mode de communication asynchrone est offert par toutes les bibliothèques de communication alors que le mode synchrone est rare. Ceci s'explique par le fait qu'il est très facile de réaliser les communications synchrones à l'aide de communications asynchrones. Il suffit que le destinataire renvoie à l'expéditeur un accusé de réception pour confirmer la réception.

Les modes bloquant et non bloquant

Les modes de communication bloquant et non bloquant sont considérés dans le modèle des communications asynchrones (figure 2.3 b,c,d). Avec le mode bloquant, l'appel d'une primitive d'envoi ou de réception se termine lorsque l'opération en question a effectivement été réalisée. L'appel de primitive d'expédition se termine quand le message a quitté l'expéditeur, ce qui ne signifie pas forcement que le message est déjà arrivé au destinataire. L'appel de la primitive de réception se termine quand le message est arrivé et a été recopié dans le tampon de réception.

Le mode non bloquant tente de diminuer le temps d'attente des primitives de communication en donnant au programmeur la possibilité de demander l'exécution d'une communication sans devoir attendre qu'elle soit effectivement réalisée.

L'utilisation des communications non bloquantes n'est évidemment intéressante que sur les machines qui permettent de réaliser des calculs et des communications simultanément. L'emploi de communications non bloquantes permet alors de masquer presque totalement les durées des communications. La quasi-totalité des bibliothèques de communication offrent à la fois les modes de communication bloquant et non bloquant.

La difficulté d'utilisation des primitives non bloquantes réside dans le fait que le programmeur doit veiller à ce que les opérations soient terminées en temps voulu. A cette fin, on lui offre de nouvelles primitives, typiquement une primitive d'attente qui permet de bloquer l'exécution jusqu'à ce que la communication demandée soit terminée, et une primitive de test qui permet de savoir si la communication a été effectuée entre temps ou non.

2.2.4 Les types de communication

Les communications peuvent être réalisées de diverses manières et offrent différentes propriétés au programmeur. On distingue les primitives de communication simples, qui n'impliquent que deux processeurs, et les primitives de communication complexes, qui peuvent concerner un sous-ensemble de processeurs.

Les communications point à point

Les communications les plus simples sont l'envoi d'un message d'un processeur donné à un autre. Une telle communication est dite point à point. Elle nécessite deux opérations explicites : le processeur expéditeur envoie le message et le processeur destinataire le reçoit. Suivant le modèle de communication retenu, le fait que le destinataire ne veuille pas considérer le message peut bloquer l'expéditeur ou non.

La primitive d'envoi est fournie par une bibliothèque de communication sous une forme proche de la suivante :

Send(tampon, taille, destinataire, type)

- tampon est un pointeur en mémoire, qui indique l'endroit de la mémoire auquel se trouve le premier octet de l'information à envoyer ;
- taille, exprimée en général en octets, est un entier indiquant la taille du message ;
- destinataire est représenté le plus souvent par un entier naturel ;
- type est un type de message, la plupart du temps un entier naturel. Le type de message permet de classer le message du point de vue de l'application ;

La primitive de réception est elle aussi fournie par la bibliothèque de communication. Sa forme typique est la suivante :

Recv(tampon, taille, expéditeur, type)

- tampon est un pointeur en mémoire qui indique l'endroit de la mémoire auquel devra être déposé le premier octet du message ;
- taille, exprimée en général en octets, est un entier indiquant la taille maximale que pourra avoir le message à recevoir ;
- expéditeur est l'identification du processus en provenance duquel on attend le message ;
- type est le type du message attendu.

Les primitives de communication élémentaires que nous venons de présenter sont l'envoi et la réception de messages. Elles sont à la base d'un certain nombre d'algorithmes réalisant des fonctions de communication plus complexes. Ces algorithmes, appelés primitives de communications collectives ou macro-communication, fournissent des solutions toutes faites pour un certain nombre de schémas de communication que l'on utilise souvent. Ces primitives concernent un sous-ensemble des processeurs de la machine parallèle.

Les communications collectives

Les primitives Les primitives classiques des communications collectives utilisées en algorithmique parallèle sont parmi les suivantes :

La synchronisation : c'est la macro-communication la plus simple qui fait intervenir tous les processus, il n'y a pas d'échange d'information à proprement parler, mais les processus sont simplement assurés du fait que tous ont rallié un certain point de leur exécution, à savoir le point de synchronisation.

La diffusion (brodcast ou one to all) : elle consiste en l'envoi d'un même message depuis un processeur distingué à tous les autres processeurs.

La distribution (scattering) : dans l'algorithme de distribution ou diffusion personnalisée, un processeur distingué envoie un message distinct à chacun des autres processeurs.

Le rassemblement (gathering) : il consiste dans l'opération inverse de la distribution. Un processeur distingué reçoit un message distinct de chaque autre processeur.

Le commérage (allgather) : chacun des processus possède une information spécifique, et à la fin de l'algorithme, tous les processus connaissent toutes les informations.

La transposition (all to all ou multi-scattering) : il s'agit de l'algorithme dans lequel chaque processeur effectue simultanément une distribution.

La réduction : elle consiste à synthétiser, sur un processus distingué, une donnée d à partir de n données di détenues respectivement par les n processus. A cet effet, on utilise une opération commutative et associative >, appelée opérateur de réduction, et on calcul la valeur de d comme étant d = >i di. Souvent, on considère une variante de la réduction dans laquelle tous les processus reçoivent la valeur finale d. Ce résultat peut être obtenu par une réduction d'après la première

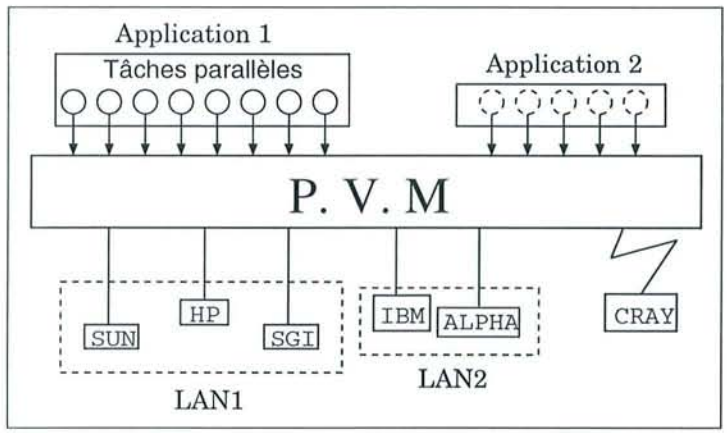

FIG. 2.4 – *Le système* PVM

définition, suivie d'une diffusion. Mais il est en général plus intéressant de fusionner ces phases en une seule étape de calcul appelée réduction-diffusion.

2.3 Quelques bibliothèques de communication

Dans ce paragraphe, nous allons introduire brièvement le principe des deux bibliothèques de communication parmi les plus utilisées actuellement, à savoir *PVM* (*Parallel Virtual Machine*) et *MPI* (*Message Passing Interface*).

2.3.1 *PVM*

PVM est un logiciel du domaine public qui permet d'exploiter un ensemble de machines hétérogènes (*DEC/Alpha, CRAY, HP, IBM/RS6000, SUN, SGI...*) et/ou un réseau de processeurs (*Thinking Machines, CM-5, Intel iPSC/860 et Paragon, CRAY T3D...*) comme une ressource unique de calcul. C'est un projet qui a commencé en été 1989 à l'initiative de chercheurs du centre d'*ORNL* (*Oak Ridge National Laboratory*) et de l'Université d'Emory (*Atlanta*), financé par des fonds publics des Etats-Unis à savoir l'*U.S. Department of Energy*, *NSF* (*National Science Foundation*) et l'état du Tennessee [Team 94].

Sur le plan architectural, *PVM* désigne une collection flexible de machines, ou hôtes, monoprocesseurs, multiprocesseurs à mémoire partagée ou distribuée, qu'un programme d'application va voir comme une machine logique unique à mémoire distribuée supportant le modèle de passage par messages. Suivant ce modèle (figure 2.4), le programme d'application, écrit en Fortran, C ou C++ se compose d'un ensemble de tâches qui vont coopérer via des primitives de communication et de synchronisation. Dans ce cadre, *PVM* gère tous les problèmes de formats différents entre les machines.

Le système *PVM* proprement dit est constitué de deux parties (figure 2.5), une partie « démon », référencée comme *pvmd3*, installée sur l'ensemble des hôtes, et une bibliothèque de primitives (libpvm3.a pour C et C++, libfpvm3.a pour Fortran) nécessaires à l'initialisation des tâches, à leur communication et leur synchronisation. Une interface utilisateur sommaire est disponible avec la version de base, mais il existe des couches graphiques de plus haut niveau comme *XPVM* [Kohl 96] ou *HENCE* [Beguelin 94].

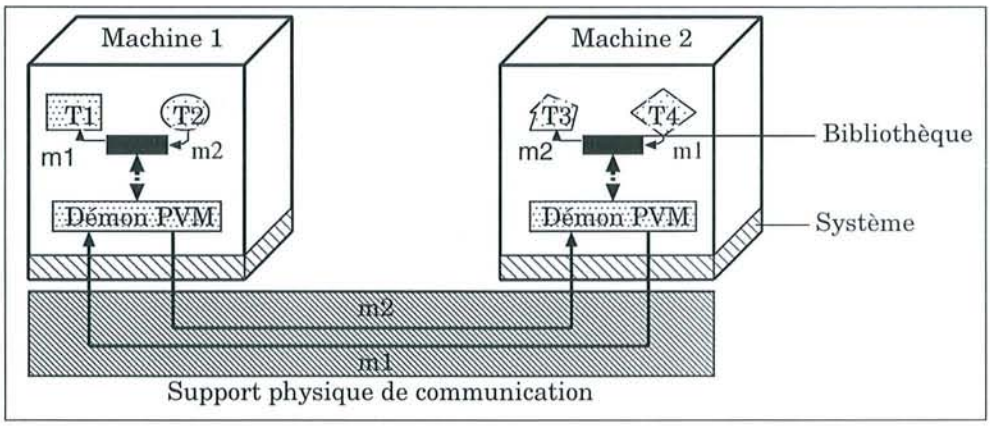

FIG. 2.5 – PVM *en action*

Le Principe

PVM permet:

- de définir une machine virtuelle composée d'un ensemble de machines (homogène ou non) ;
- de démarrer automatiquement des tâches sur la machine virtuelle ;
- aux tâches de communiquer et de se synchroniser.

Composantes

Une bibliothèque de primitives (libpvm3.a pour C, libfpvm3.a pour Fortran) nécessaires à l'initialisation des tâches, à leur communication et leur synchronisation.

La console *PVM* : la console permet de contrôler la machine virtuelle.

Le démon pvmd3: le démon *pvmd3* est lancé sur chaque machine qui fait partie de la machine virtuelle, il permet de contrôler les échanges entre les tâches. Soit il est lancé à partir de la console *PVM*, soit il est lancé à partir d'un fichier de configuration qui permet de décrire la machine virtuelle.

Développement

PVM permet d'utiliser tous les modèles de l'extension de la classification de *Flynn*. Néanmoins les plus utilisés sont ceux basés sur le *SPMD* et sur le *MPMD* notamment celui du *maître/esclave*.

Le modèle *maître/esclave* utilise deux programmes :

- Un *maître* qui :
  - lance les tâches que doivent exécuter les esclaves ;
  - distribue les données aux esclaves ;
  - reçoit les données traitées ;
  - peut exécuter des opérations séquentielles ou d'Entrées/Sorties.
- Des *esclaves* qui :
  - reçoivent des données ;
  - traitent ces données ;
  - émettent les données traitées.

Les tâches

Création des tâches : De la même manière que la construction de la machine virtuelle est entièrement dynamique, les tâches *PVM* constituant une application distribuée peuvent être lancées/arrêtées dynamiquement. En conséquence, le nombre de tâches fonctionnant sur une machine virtuelle peut varier. Leur lancement peut être fait soit par l'utilisateur (comme une commande UNIX classique), soit par une tâche elle-même (*pvm-spawn*).

Identification des tâches : Leur identification se fait par un « Task IDentifier » (TID), qui inclut à la fois le numéro de noeud sur lequel s'exécute la tâche, et le numéro de processus UNIX correspondant à la tâche.

Placement des tâches : Il se fait sur les différents noeuds en précisant où lancer la tâche (paramètres de pvm-spawn), mais par défaut *PVM* choisit lui même les noeuds en appliquant un simple algorithme de placement cyclique : la première tâche sur le premier noeud, la deuxième sur le second, etc.

Groupement de tâches : *PVM* permet de rassembler dynamiquement les tâches dans des groupes, afin de leur appliquer des opérations collectives. Ici encore, le nombre de groupes est dynamique. N'importe quelle tâche peut créer un nouveau groupe, et n'importe quelle tâche peut le joindre (*pvm_joingroup*), en précisant le nom du groupe.

Au sein de chaque groupe, chaque tâche se voit attribuer un numéro d'instance unique. Ce numéro correspond exactement à l'ordre d'entrée dans le groupe. Le couple (groupe, instance) permet donc d'identifier une tâche *PVM* de façon unique. Il existe d'ailleurs une fonction *PVM* permettant de faire le lien directement avec le TID de la tâche.

Les communications

Communications point à point : Elles sont basées sur une primitive d'envoi non bloquante (pvm-send) et une primitive de réception bloquante (pvm-recv). Cependant, avant de pouvoir envoyer un message, il est nécessaire de réserver explicitement un buffer (*pvm-initsend*) pour stocker le message avant l'envoi, et de construire le contenu du message lui-même (*pvm-pack*). Symétriquement, la réception du message doit être suivie par un dépaquetage du message (pvm-unpack). Une fonction permettant de scruter l'arrivée de messages (*pvm-probe*) permet d'attendre un message sans se bloquer, et offre donc la possibilité d'une réception non bloquante.

Communications collectives : permettent de faire des diffusions (*pvm-bcast*) ou des multi-diffusions (*pvm-mcast*), ainsi que des opérations de réduction (*pvm-reduce*) ou de distribution et regroupement (*pvm-scatter et pvm-gather*).

2.3.2 *MPI*

La bibliothèque de communication *MPI* est la spécification d'un standard d'interface d'échange de messages sur les machines à mémoires distribuées.

Comme pour toute bibliothèque, le but de ce standard consiste à avoir accès à la même interface facile à utiliser sur toutes les machines du même type. La bibliothèque résultante de *MPI* pourra être optimisée par chaque constructeur de machines parallèles afin de tenir compte des spécificités de ses machines en terme de communications. De nombreux groupes de travail, réunissant chercheurs, utilisateurs et constructeurs, ont permis à *MPI* de couvrir une grande variété d'applications et de tenir compte des succès et des échecs rencontrés par les définitions et les réalisations des autres bibliothèques.

MPI contient des routines de communication point à point, des routines globales, ainsi que des routines comportant des calculs comme la sommation ou le calcul de maximum.

Deux notions importantes dans *MPI* sont la notion de contexte de communication et la notion de groupe de processus. Les contextes et les groupes permettent de définir des partitions dans l'espace des messages et respectivement dans l'espace des processeurs. Des routines permettent de les manipuler (création, modification, destruction). Les contextes sont particulièrement importants pour la construction de bibliothèques parallèles puisqu'elles permettent d'éviter un mélange des messages entre différentes bibliothèques, voir entre différentes instances d'une même bibliothèque. Les groupent et contextes forment un objet communicateur et on envoie un message à un processeur d'un groupe et ce dans un contexte particulier.

La machine virtuelle

Tout ce qui concerne la machine virtuelle n'est jamais abordé dans *MPI*, puisque ces considérations relèvent de l'implantation de *MPI*. En conséquence, la gestion de la machine virtuelle diffère suivant l'implantation de *MPI* que l'on choisit. En se limitant aux implantations domaine public, on trouve les deux cas de figures :

1. *MPICH* propose une gestion entièrement statique des noeuds de la machine virtuelle. En effet, il est nécessaire de préciser dans un fichier, la liste des stations participant à la machine virtuelle. Cette machine est construite avant de lancer toute application *MPI*, et sa configuration sera active jusqu'à sa mort.
2. LAM-*MPI* propose un mécanisme plus proche de ce qui existe dans *PVM*, dans le sens ou l'utilisateur peut modifier à la main sa machine virtuelle. Par contre, les tâches elles-mêmes n'ont pas la possibilité d'agir sur ce paramètre.

En résumé, le concept de machine virtuelle n'existe pas à proprement parler au sein de la norme *MPI*. Il dépend directement de l'implantation. Pour simplifier, tout ce qui pouvait être géré dynamiquement dans *PVM* sera statique dans le standard *MPI*. La verion MPI2 [Gropp 99] comble cette lacune et propose une gestion dynamique de tâches.

Création des tâches : Le nombre de tâches constituant une application programmée avec *MPI* sera fixé au départ par un moyen dépendant de l'implantation, généralement en donnant « en extension » une liste d'exécutables qui seront lancés. Aucune tâche ne pourra être créée par la suite.

Identification des tâches : Le nommage des tâches par défaut est entièrement numérique, mais la liaison avec le numéro de processus n'apparaît pas comme dans le TID de *PVM*. Les numéros sont attribués selon l'ordre d'apparition dans le fichier de déclaration des tâches de l'application.

MPI propose aussi une définition logique de la topologie des tâches en précisant quels sont les liens entre tâches. Cette définition reste purement logique, et ne donnera en particulier pas lieu à un mapping des tâches sur une architecture physique spécifique. Elle offrira simplement un moyen d'identification différent des tâches (on parlera alors de voisins, par exemple).

Placement des tâches : En ce qui concerne le placement des tâches, il sera lui-aussi statique. Il sera, ici encore, dépendant de l'implantation, et sera en général précisé en même temps que la liste des tâches à lancer.

Groupement des tâches : *MPI* offre une notion de groupes dans lesquels on peut regrouper les tâches, mais ces groupes ne sont utilisées que comme un moyen de construire des communicateurs *MPI*, qui sont détaillés dans la suite de cette section.

Les communications

Les communications offrent beaucoup plus de possibilités avec *MPI* qu'il n'y en avait avec *PVM*.

Communications point à point Concernant les primitives d'envoi et de réception tout d'abord, alors que *PVM* n'offrait schématiquement qu'un envoi non-bloquant et une réception bloquante, *MPI* décline les fonctions de communication en quatre modes d'échange :

1. Le mode *standard send*, où l'implantation *MPI* se charge d'acheminer le message de la source à la destination sans préciser le "comment" des choses. En pratique, l'implantation est chargée de choisir l'un des trois modes ci-dessous suivant la disponibilité des ressources ;
2. le mode *ready send*, où le message ne peut être envoyé que si le « receive » correspondant a déjà été déclenché. Dans le cas contraire une erreur apparaît ;
3. le mode *buffered send*, où le message peut être envoyé sans que le receive correspondant n'aie eu lieu. Le message sera par contre stocké dans un buffer d'envoi, envoi qui pourra avoir lieu en tâche de fond, si le système le permet ;
4. le mode *synchronous send*, où le message ne sera envoyé qu'au moment où le receive correspondant sera appelé. Dans ce cas, la communication se fait donc par rendez-vous.

Enfin, dans *MPI*, ces quatre modes peuvent être déclinés de façon bloquante ou non. Pour le standard *MPI*, rendre une primitive non-bloquante signifie simplement la scinder en deux parties : l'une pour initier l'envoi (resp. la réception), l'autre pour s'assurer que l'envoi (resp. la réception) s'est correctement terminé.

Cependant, la grande particularité de *MPI* réside dans le fait que chaque communication est associée à un contexte, qui est représenté par le communicateur *MPI*. Un communicateur représente en fait une espèce de canal de communication qui relie un ensemble déterminé de tâches *MPI*. Chaque tâche possède un numéro d'identification au sein d'un communicateur. Ce communicateur, associé à un numéro d'identification pourra alors être utilisé pour échanger des messages entre tâches faisant partie du communicateur.

Il est possible de définir autant de communicateurs que l'on veut et une tâche peut appartenir à un nombre illimité de communicateurs. Ce mécanisme permet de définir plusieurs liaisons logiques entre deux mêmes tâches, qui pourront être utilisées pour transmettre des informations différentes.

Mais, lorsqu'une tâche attend par exemple deux messages associés à des traitements différents, ce mécanisme permet facilement de différencier deux messages. Toute la gestion des communicateurs s'appuie directement sur des opérations ensemblistes telles que l'union, l'intersection, la différence ensembliste, etc. De ce fait, les phases de création des communicateurs sont plus « propres » qu'avec les groupes *PVM*, puisque tout le monde connaît à tout moment le contenu exact du communicateur, évitant ainsi les perpétuelles erreurs de synchronisation sur des groupes *PVM* incomplets, lorsque les tâches *PVM* n'ont pas encore toutes rejoint un groupe (inconvénient de l'aspect dynamique du ralliement au groupe).

Communications collectives : Les opérations collectives au sein de *MPI* sont très classiques. Elles sont néanmoins un peu plus variées notamment concernant les échanges complets (MPI-Alltoall) que celles proposées dans *PVM*. De plus grâce aux contextes *MPI*, une communication collective ne peut pas interférer avec une communication point à point.

2.4 Conclusion

Au premier abord, la mise en œuvre de la programmation par échange de messages ne semble pas très compliquée. En effet, il existe beaucoup de bibliothèques de communication, les principes utilisés sont simples et ne sont en fait que des extensions des notions utilisées pour les programmes séquentiels.

Cependant, si le but initial était de fournir des bibliothèques d'échange de messages capables de franchir le degré de difficulté rencontré avec les mécanismes de communication de bas niveau (Socket, RPC,..), le résultat final est que ce but n'a été que partiellement atteint. Ce mode de programmation est considéré toujours comme un assembleur du parallélisme.

En outre, obtenir une exécution correcte, déterministe, et efficace reste un réel problème. Il est impératif de maîtriser à la fois le paradigme de programmation et les mécanismes utilisés (bibliothèque de communication et architecture matérielle) pour obtenir les performances escomptées.

Par ailleurs, et pour répondre aux besoins multiples des applications distribuées, d'autres environnements de développement de ce type d'applications ont vu le jour. Ces environnements fournissent tout un ensemble de mécanismes pour rendre au maximum transparents les problèmes de communication et de répartition. Parmi ces environnement, on peut citer *CORBA* [Geib 97], *DCE* [Group. 98] ou bien *DCOM* [Thai 99]. Il est particulièrement étonnant de voir à quelle vitesse ces standards sont entrés dans le monde industriel. Nous présenterons dans le prochain chapitre l'un des ces environnements à savoir l'architecture CORBA et nous justifierons le choix de cette architecture.

Chapitre 3

Vers un nouveau modèle de programmation pour les applications distribuées : CORBA

L'objectif de ce chapitre est de passer en revue les composants de CORBA et le modèle de communication associé.

3.1 Introduction

Les besoins en matière d'application de type client/serveur ont conduit à la définition d'une norme CORBA [OMG 98a] standardisant le support de l'exécution d'application répartie à objets. Cette norme définit l'ensemble des services et des supports dont a besoin une application à objets pour s'exécuter dans un environnement réparti. De nombreuses implémentations, plus ou moins complètes de la norme CORBA sont disponibles.

En étudiant cette architecture, nous avons dégagé plusieurs avantages :

- d'ordre qualitatif : mécanismes évolués de transparence à la localisation, portabilité, interopérabilité, gestion d'hétérogénéité matérielle et logicielle, etc ;
- d'ordre quantitatif : les temps de communication réalisés sur certaines implantations de *CORBA* sont comparables à ceux obtenus en utilisant des bibliothèques d'échange de messages (voir chapitre 8).

Après une brève présentation de l'approche orientée objet, nous mettons l'accent sur l'architecture CORBA et ses composants. Nous abordons ensuite les principes de programmation d'une application dans un environnement réparti à objets de type CORBA. Nous détaillons ensuite le modèle de communication que cette architecture propose.

3.2 L'approche orientée objet

Un objet associe des données et des traitements dans une même entité en ne laissant que l'interface de l'objet, c'est-à-dire les opérations que l'on peut effectuer dessus. L'objet facilite l'analyse et la programmation [Chauvet 97]. L'analyse est facilitée car les objets permettent une modélisation directe des objets réels et de leurs comportements. Le processus de programmation

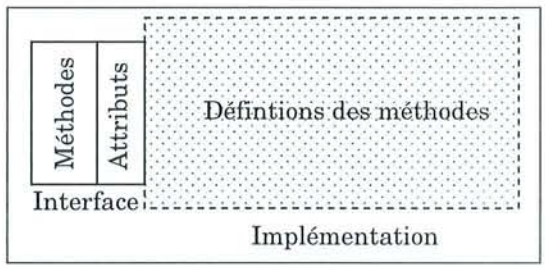

FIG. 3.1 – *Les composants d'un objet*

est simplifié car ces objets sont réutilisables, faciles à adapter et à assembler, ce qui facilite le prototypage rapide de l'application.

Dans la terminologie habituelle des technologies orientées objet, les traitements sont appelés les méthodes de l'objet et les données sont appelées propriétés, attributs, champs ou quelques fois variables de l'objet (figure 3.1).

Ces objets font d'excellents modules logiciels. En effet, ils peuvent être construits et programmés indépendamment les uns des autres, chaque objet forme une unité autonome, caractérisée entièrement par ses attributs et ses méthodes. Ils sont un moyen naturel de découper la tâche de programmation des applications en étapes indépendantes et plus aisément mises en œuvre.

3.2.1 Les interactions entre objets

Si chaque objet représente une partie des objets physiques mis en jeu dans les opérations réelles, une application concrète, contiendra de nombreux objets et une grande richesse d'interactions entre ceux-ci. Le message, la seconde clé pour la compréhension de l'orienté objet, est la représentation des interactions entre objets. Dans la terminologie généralement acceptée, on dit que lorsqu'il y a interaction entre deux objets l'un a envoyé un message à l'autre, ou encore qu'un objet a reçu un message d'un autre objet. Notons qu'ici les termes peuvent varier. Dans certains modèles de systèmes d'objets on parle de requêtes plutôt que de messages, mais le sens original est identique. Un message est simplement constitué du nom de l'objet qui en est destinataire, du nom d'une méthode de cet objet et d'une liste, éventuellement vide, de données également appelées paramètres ou encore arguments du message.

3.3 Une vision globale de la construction d'applications objets répartis

3.3.1 L'OMG

L'Object Management Group (OMG) est un consortium international créé en 1989 et regroupant actuellement plus de 850 acteurs du monde informatique : des constructeurs (IBM, Sun), des producteurs de logiciel (Netscape, Inprise ou ex-Borland/Visigenic, IONA Tech.), des utilisateurs (Boeing, Alcatel) et des institutionnels et universités (NASA, INRIA, LIFL) [Geib 97]. L'objectif de ce groupe est de faire émerger des standards pour l'intégration d'applications distribuées hétérogènes à partir des technologies orientées objet. Ainsi les concepts clés mis en avant sont la réutilisabilité, l'interopérabilité et la portabilité de composants logiciels. L'élément clé de la vision de l'OMG est CORBA (Common Object Request Broker Architecture) : une architecture logicielle orientée objet. Ce bus d'objets répartis offre un support d'exécution masquant les couches techniques d'un système réparti (système d'exploitation, processeur et réseau)

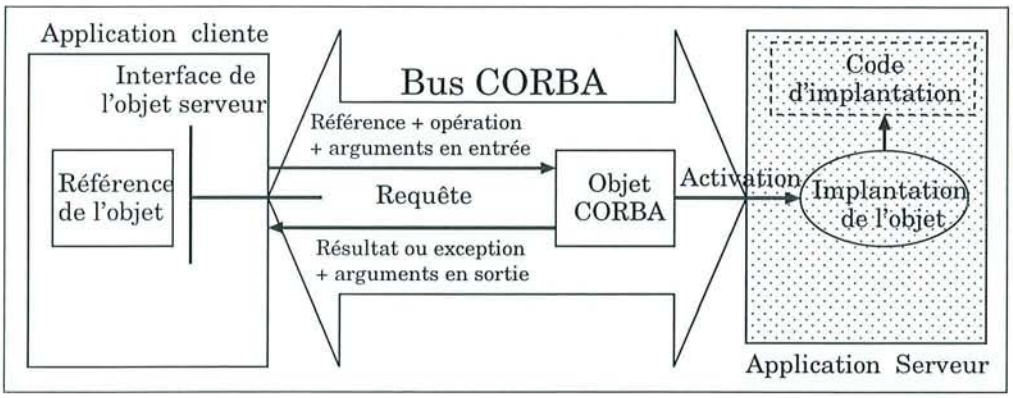

FIG. 3.2 – *Le modèle objet client/serveur de CORBA*

et il prend en charge les communications entre les composants logiciels formant les applications réparties hétérogènes.

3.3.2 Le modèle objet client/serveur

Le but de CORBA est de permettre un développement plus simple de programmes utilisant l'architecture client/serveur. Jusqu'à présent, un programmeur qui souhaitait mettre en place une application fonctionnant en client/serveur disposait des solutions techniques suivantes :

- utilisation des RPC (Remote Procedure Call) [Gressier 95], qui permettent à un programme écrit en C d'appeler des procédures distantes. Un tel système ne permet pas de faire communiquer un ensemble des modules écrits dans divers langages de programmation ;
- l'utilisation directe, dans le programme, des sockets [Quinton 97] . C'est là une des méthodes les plus classiques, mais elle requiert de gérer soi même les envois de messages.

CORBA propose une nouvelle approche : un modèle orienté objet client/serveur d'abstraction et de coopération entre les applications réparties [Geib 97]. Chaque application peut exporter certaines de ses fonctionnalités (services) sous la forme d'objets CORBA : c'est la composante d'abstraction (structuration) de ce modèle. Les interactions entre les applications sont alors matérialisées par des invocations à distance des méthodes sur des objets : c'est la partie coopération. La notion client/serveur intervient uniquement lors de l'utilisation d'un objet : l'application implantant l'objet est le serveur, l'application utilisant l'objet est le client. Bien entendu, une application peut tout à fait être à la fois cliente et serveur.

La figure 3.2 présente les différentes notions intervenant dans ce modèle objet client/serveur :

- l'application cliente est un programme qui invoque les méthodes des objets à travers le bus CORBA ;
- la référence d'objet est une structure désignant l'objet CORBA et contenant l'information nécessaire pour le localiser sur le bus ;
- l'interface de l'objet est le type abstrait de l'objet CORBA définissant ses opérations et attributs. Celle-ci se définit par l'intermédiaire du langage OMG-IDL (voir section 3.4) ;
- la requête est le mécanisme d'invocation d'une opération ou d'accès à un attribut de l'objet ;
- le bus CORBA achemine les requêtes de l'application cliente vers l'objet en masquant tous les problèmes d'hétérogénéité (langages, systèmes d'exploitation, matériels, réseaux) ;

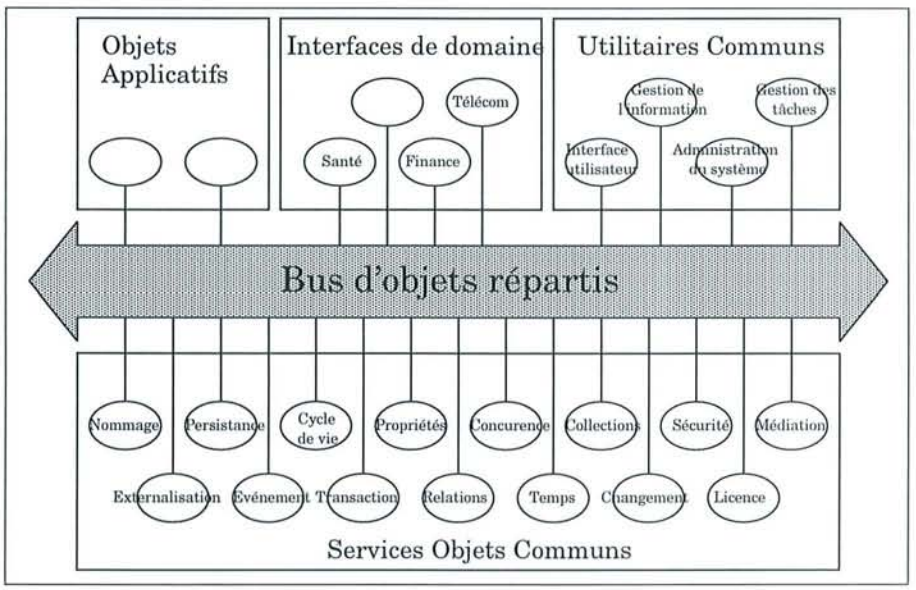

FIG. 3.3 – *L'architecture globale de l'OMG*

- l'objet CORBA est le composant logiciel cible. C'est une entité virtuelle gérée par le bus CORBA ;
- l'activation est le processus d'association d'un objet d'implantation à un objet CORBA ;
- l'implantation de l'objet est l'entité codant l'objet CORBA à un instant donné et gérant un état de l'objet temporaire. Au cours du temps, un même objet CORBA peut se voir associer des implantations différentes ;
- le code d'implantation regroupe les traitements associés à l'implantation des opérations de l'objet CORBA. Cela peut être par exemple une classe Java aussi bien qu'un ensemble de fonctions C ;
- l'application serveur est la structure d'accueil des objets d'implantation et des exécutions des opérations. Cela peut être par exemple un processus Unix.

Dans la suite de ce chapitre, nous verrons que ces notions abstraites se traduisent par des composantes technologiques fournies par la norme CORBA.

L'architecture globale

L'OMG définit aussi une vision globale de la construction d'applications réparties : l'Object Management Architecture Guide [Soley 95]. Cette architecture globale, appelée aussi l'OMA (figure 3.3), vise à classifier les différents objets qui interviennent dans une application en fonction de leurs rôles :

- Le bus d'objets répartis est la clé de voûte de l'architecture globale de l'OMG. Il assure le transport des requêtes entre tous les objets CORBA. Il offre un environnement d'exécution aux objets masquant l'hétérogénéité liée aux langages de programmation, aux systèmes d'exploitation, aux processeurs et aux réseaux. Ce bus est plus amplement détaillé dans la suite ;
- les services objet communs (CORBA services) fournissent sous forme d'objets CORBA, spécifiés grâce au langage OMG-IDL, les fonctions systèmes nécessaires à la plupart des

applications réparties. Actuellement, l'OMG a défini des services pour les annuaires (Nommage et Médiation), le cycle de vie des objets, les relations entre objets, les événements, les transactions, la sécurité, la persistance, etc [OMG 98b]. Au fur et à mesure des besoins, l'OMG ajoute de nouveaux services communs ;

- les utilitaires communs (CORBA facilities) sont des canevas d'objets ("Frameworks") qui répondent plus particulièrement aux besoins des utilisateurs. Ils standardisent l'interface utilisateur (e.g. OpenDoc), la gestion de l'information, l'administration, le Workflow, etc ;
- les interfaces de domaine (Domain Interfaces) définissent des objets de métiers spécifiques à des secteurs d'activités comme la finance (e.g. monnaie électronique), la santé (e.g. dossier médical) et les télécoms (e.g. transport multimédia). Leur objectif est de pouvoir assurer l'interopérabilité sémantique entre les systèmes d'informations d'entreprises d'un même métier : les " Business Object Frameworks " (BOFs) ;
- les objets applicatifs (Application Objects) sont ceux qui sont spécifiques à une application répartie et ne sont donc pas standardisés. Toutefois, dès que le rôle de ces objets apparaît dans plus d'une application ils peuvent alors rentrer dans une des catégories précédentes et donc être standardisés par l'OMG.

Le bus d'objets répartis CORBA

Les caractéristiques Le bus CORBA est donc l'intermédiaire/négociateur à travers lequel les objets vont pouvoir dialoguer. Il fournit les caractéristiques suivantes :

- la liaison avec "tous" les langages de programmation : cependant, actuellement l'OMG a seulement défini officiellement cette liaison pour les langages C, C++, SmallTalk, Ada, COBOL et Java ;
- la transparence des invocations : les requêtes aux objets semblent toujours être locales, le bus CORBA se chargeant de les acheminer en utilisant le canal de communication le plus approprié ;
- l'invocation statique et dynamique : ces deux mécanismes complémentaires permettent de soumettre les requêtes aux objets. En statique, les invocations sont contrôlées à la compilation. En dynamique, les invocations doivent être contrôlées à l'exécution ;
- un système auto-descriptif : les interfaces des objets sont connues du bus et sont aussi accessibles par les programmes par l'intermédiaire du référentiel des interfaces ;
- l'activation automatique et transparente des objets : les objets sont en mémoire uniquement s'ils sont utilisés par des applications clientes ;
- l'interopérabilité entre bus : à partir de la norme CORBA 2.0, un protocole générique de transport des requêtes (GIOP pour General Inter-ORB Protocol) a été défini permettant l'interconnexion de bus CORBA provenant de fournisseurs distincts, une de ses instanciations est l'Internet Inter-ORB Protocol (IIOP) fonctionnant au-dessus de TCP/IP ;

Les composantes Le bus CORBA fournit les composantes suivantes (figure 3.4) :

- ORB (Object Request Broker) est le noyau de transport des requêtes aux objets. Il intègre au minimum les protocoles GIOP et IIOP. L'interface au bus fournit les primitives de base comme l'initialisation de l'ORB ;
- SII (Static Invocation Interface) est l'interface d'invocations statiques permettant de soumettre des requêtes contrôlées à la compilation des programmes. Cette interface est générée à partir de définitions OMG-IDL ;

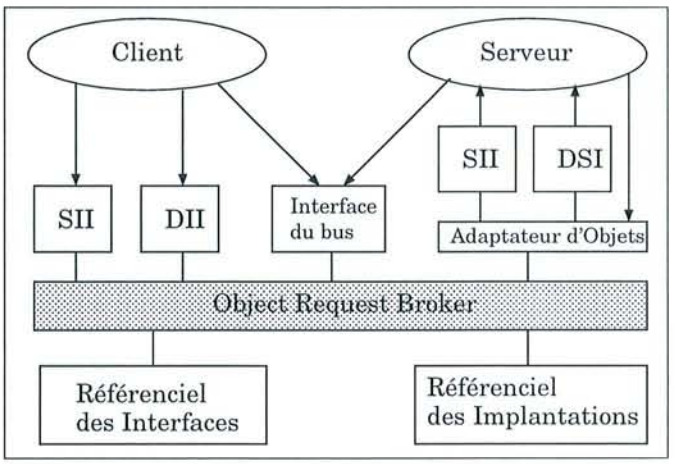

FIG. 3.4 – *Les composantes du bus CORBA*

- DII (Dynamic Invocation Interface) est l'interface d'invocations dynamiques permettant de construire dynamiquement des requêtes vers n'importe quel objet CORBA sans générer/utiliser une interface SII ;
- IFR (Interface Repository) est le référentiel des interfaces contenant une représentation des interfaces OMG-IDL accessible par les applications durant l'exécution ;
- SSI (Skeleton Static Interface) est l'interface de squelettes statiques qui permet à l'implantation des objets de recevoir les requêtes leur étant destinées. Cette interface est générée comme l'interface SII ;
- DSI (Dynamic Skeleton Interface) est l'interface de squelettes dynamiques qui permet d'intercepter dynamiquement toute requête sans générer une interface SSI. C'est le pendant de DII pour un serveur ;
- OA (Object Adapter) est l'adaptateur d'objets qui s'occupe de créer les objets CORBA, de maintenir les associations entre objets CORBA et implantations et de réaliser l'activation automatique si nécessaire ;
- ImplR (Implementation Repository) est le référentiel des implantations qui contient l'information nécessaire à l'activation. Ce référentiel est spécifique à chaque produit CORBA ;

Ces différentes composantes sont toutes décrites dans le langage OMG-IDL, ce qui les rend accessibles au travers du bus (e.g. le référentiel des interfaces).

Les protocoles réseaux Tout bus à la norme CORBA 2.0 doit fournir les protocoles GIOP et IIOP. Le protocole GIOP définit une représentation commune des données (CDR ou Common Data Representation), un format de références d'objet interopérable (IOR ou Interoperable Object Reference) et un ensemble de messages de transport des requêtes aux objets (Request, Reply, etc). Cependant, GIOP est seulement un protocole générique, IIOP fournit alors une implantation de GIOP au dessus de TCP/IP et donc d'Internet. Les IORs dans le contexte d'IIOP doivent contenir :

- le nom complet de l'interface OMG-IDL de l'objet ;
- l'adresse IP de la machine Internet où est localisé l'objet ;
- un port IP pour se connecter au serveur de l'objet ;

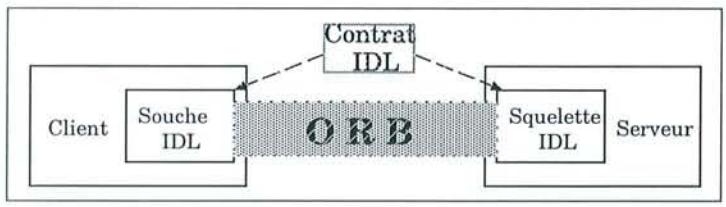

FIG. 3.5 – *Projection du contrat IDL en souches et squelettes*

- une clé pour désigner l'objet dans le serveur. Son format est libre et il est donc différent pour chaque implantation du bus CORBA.

Toutefois, un bus CORBA peut contenir d'autres protocoles de transport des requêtes aux objets.

La recherche d'objets Cette catégorie de services offre les mécanismes pour rechercher/retrouver dynamiquement sur le bus les objets nécessaires aux applications. Ce sont les équivalents des annuaires téléphoniques :

- le service Nommage (Naming Service) est l'équivalent des " pages blanches " : les objets sont désignés par des noms symboliques. Cet annuaire est matérialisé par un graphe de répertoires de désignation ;
- le service Vendeur ou médiation (Trader Service) est l'équivalent des "pages jaunes" : les objets peuvent être recherchés en fonction de leurs caractéristiques.

3.4 Le langage OMG-IDL

3.4.1 La notion du contrat IDL

Le langage OMG-IDL (Interface Definition Language) permet d'exprimer, sous la forme de contrats IDL, la coopération entre les fournisseurs et les utilisateurs de services, en séparant l'interface de l'implantation des objets et en masquant les divers problèmes liés à l'interopérabilité, l'hétérogénéité et la localisation de ceux-ci [Geib 97].

Un contrat IDL spécifie les types manipulés par un ensemble d'applications réparties, c'est-à-dire les types d'objets (ou interfaces IDL) et les types de données échangés entre les objets. Le contrat IDL isole ainsi les clients et fournisseurs de l'infrastructure logicielle et matérielle les mettant en relation à travers le bus CORBA.

Pour plus d'informations sur la syntaxe du langage OMG-IDL, nous vous conseillons de vous reporter à [Geib 97] et [OMG 98a].

3.4.2 La projection vers un langage de programmation

Les contrats IDL sont projetés en souches IDL (ou interface d'invocations statiques SII) dans l'environnement de programmation du client et en squelettes IDL (ou interface de squelettes statiques SSI) dans l'environnement de programmation du fournisseur (figure 3.5). Le client invoque localement les souches pour accéder aux objets. Les souches IDL construisent des requêtes, qui vont être transportées par le bus, puis délivrées par celui-ci aux squelettes IDL qui les délégueront aux objets. Ainsi le langage OMG-IDL est la clé de voûte du bus d'objets répartis CORBA.

Une projection est la traduction d'une spécification OMG-IDL dans un langage d'implantation. Pour permettre la portabilité des applications d'un bus vers un autre, les règles de projection sont normalisées et fixent précisément la traduction de chaque construction IDL en une ou des constructions du langage cible et les règles d'utilisation correcte de ces traductions. Actuellement, ces règles existent pour les langages C, C++, SmallTalk, Ada, Java et Cobol orienté objet. Nous ne détaillons pas ici ces règles, pour plus d'informations, reportez-vous à [OMG 98a].

La projection est réalisée par un pré-compilateur IDL dépendant du langage cible et de l'implantation du bus CORBA cible. Ainsi, chaque produit CORBA fournit un pré-compilateur IDL pour chacun des langages supportés. Le code des applications est alors portable d'un bus à un autre car les souches/squelettes générés s'utilisent toujours de la même manière quel que soit le produit CORBA. Par contre, le code des souches et des squelettes IDL n'est pas forcément portable car il dépend de l'implantation du bus pour lequel ils ont été généré.

3.5 Les mécanismes dynamiques de CORBA

Le bus CORBA permet de réaliser des applications composées d'objets répartis. Pour cela, le pré-compilateur OMG-IDL génère automatiquement les souches de communication avec les objets. Ces souches utilisent alors le bus pour réaliser la coopération des objets des applications. Cette approche statique est bien adaptée pour la conception et l'exécution d'applications dont les spécifications OMG-IDL sont stabilisées comme nous l'avons vu dans la section précédente. Néanmoins, dans la plupart des applications complexes, certaines spécifications évoluent au cours du temps par l'ajout de nouvelles opérations et/ou de nouveaux types d'objets ou par la modification des spécifications existantes. Dans ces contextes, l'approche statique, par pré-génération automatique de souches, ne convient plus. Elle crée un lien statique (à la compilation) entre les applications clientes et les interfaces IDL des objets utilisés. Ainsi, lorsque les interfaces évoluent, il faut modifier et recompiler toutes les applications clientes et serveurs.

D'un autre côté, CORBA offre un ensemble de mécanismes pour exploiter et implanter dynamiquement des objets répartis : le référentiel des interfaces (IFR), l'interface d'invocations dynamiques (DII) et l'interface de squelettes dynamiques (DSI). Ces mécanismes dynamiques permettent de construire des applications qui s'adaptent automatiquement aux changements/évolutions des spécifications IDL.

3.6 La mise en place d'une application CORBA

L'OMG n'impose pas de processus de conception et de développement d'applications distribuées : la norme CORBA laisse une entière liberté sur le choix des outils à mettre en œuvre. Néanmoins, la mise en place d'une application CORBA suit toujours à peu près le scénario ci-après (voir figure 3.6) :

1. **La définition du contrat IDL** : à partir du cahier des charges, il faut définir les objets composant l'application à l'aide d'une méthodologie orientée objet (e.g. OMT ou l'UML [OMG 95]). Cette modélisation est ensuite traduite sous la forme de contrats IDL composés des interfaces des objets et des types de données utiles aux échanges d'informations entre les objets.
2. **La projection vers les langages de programmation** : les interfaces des objets sont décrites dans des fichiers texte. Le pré-compilateur prend en entrée un tel fichier et opère un contrôle syntaxique et sémantique des définitions OMG-IDL contenues dans ce fichier.

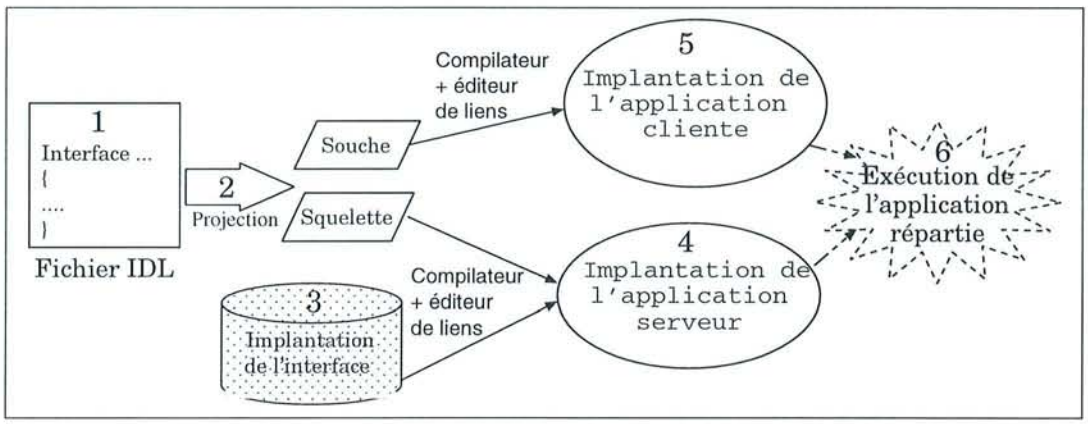

FIG. 3.6 – *La mise en place d'une application CORBA*

Ensuite il génère le code des souches qui sera utilisé par les applications clientes des interfaces décrites dans le fichier IDL, ainsi que le code des squelettes pour les programmes serveurs implantant ces types. Cette projection est spécifique à chaque langage de programmation : un environnement CORBA fournit un pré-compilateur IDL pour chacun des langages supportés.

3. **L'implantation des interfaces IDL** : en complétant et/ou en réutilisant le code généré pour les squelettes, le développeur implante les objets dans le langage de son choix ou dans le langage le mieux adapté à la réalisation de ses objets. Il doit tout de même se plier aux règles de projection vers ce langage.
4. **L'implantation des serveurs d'objets** : le développeur doit écrire les programmes serveurs qui incluent l'implantation des objets et les squelettes pré-générés. Ces programmes contiennent le code pour se connecter au bus, instancier les objets racines du serveur, rendre publiques les références sur ces objets à l'aide par exemple du service Nommage et se mettre en attente de requêtes pour ces objets.
5. **L'implantation des applications clientes des objets** : le développeur écrit un ensemble de programmes clients qui agissent sur les objets en les parcourant et en invoquant des opérations sur ceux-ci. Ces programmes incluent le code des souches, le code pour l'interface Homme-Machine et le code spécifique à l'application. Les clients obtiennent les références des objets serveurs en consultant par exemple le service Nommage.
6. **L'exécution répartie de l'application** : enfin, l'exploitation de l'application peut commencer. Le bus d'objets répartis CORBA assure alors les communications entre les programmes clients et les objets via le protocole IIOP.

3.7 Conclusion

De nombreuses autres solutions "middlewares" telles que DCE, Java et les propositions Microsoft pour la construction et l'exécution d'applications réparties existent. Cependant la norme CORBA se distingue car elle offre :

- une solution ouverte et évolutive : choisir CORBA revient à ne pas s'enfermer dans une solution propriétaire évoluant difficilement. L'OMG réagit très vite aux demandes du marché. Par exemple, CORBA a su très rapidement intégrer le langage Java et offre des passerelles

d'interopérabilité avec d'autres mondes comme DCE, COM et DCOM. De plus, la portabilité visée par l'OMG permet de changer plus ou moins facilement de fournisseurs de technologies CORBA ;

- une architecture modulaire : grâce à l'OMA, une application répartie n'est pas construite à partir de zéro mais réutilise des composants logiciels offrant des fonctions orientées système, utilisateur et/ou métier ;
- l'interopérabilité entre composants hétérogènes : CORBA fournit les abstractions et mécanismes offrant un bus orienté objet d'intégration de composants logiciels conçus avec des technologies hétérogènes (langages, systèmes d'exploitation, machines et réseaux). Ainsi, diverses technologies peuvent être mises en œuvre et coopérer dans la même application ;
- le libre choix des technologies d'implantation : aucune contrainte aussi bien au niveau des réseaux, des machines, des systèmes d'exploitation que des langages de programmation n'est imposée ;
- la portabilité du code : les fragments de code sont totalement portables d'une implantation d'un bus à une autre ;
- l'encapsulation de l'existant : la réutilisation de codes existants (voire même d'applications complètes) peut être réalisée par encapsulation de ceux-ci dans des objets CORBA. Ces objets sont alors utilisables pour bâtir de nouvelles applications ;
- la programmation multi-flot : elle est quasiment inhérente aux environnements d'exécution répartis objet. En effet, un objet est susceptible de recevoir simultanément plusieurs invocations en provenance d'objets distincts. L'environnement de programmation fournit l'ensemble des primitives nécessaires au contrôle et à la synchronisation des flots d'exécution (thread). Cette propriété permet de réaliser aisément du recouvrement calcul/communication.

Ainsi, CORBA s'impose progressivement comme le choix incontournable quel que soit le domaine d'applications réparties abordé.

Deuxième partie

Génie logiciel pour les communications dans les applications distribuées et parallèles

Chapitre 4

Le besoin d'environnements pour les applications du calcul distribué et parallèle

Nous abordons dans ce chapitre la problématique du besoin d'environnements pour le développement des applications de calcul distribué et parallèle. Dans ce contexte, nous présentons brièvement l'environnement développé dans l'équipe pour aider l'utilisateur à développer des applications parallèles basées sur des modèles de programmation à mémoire distribuée et à les exécuter en utilisant au mieux les ressources machine et réseau disponibles.

4.1 Introduction

Le développement des applications distribuées et parallèles se poursuit. Cependant, même si des progrès significatifs ont été accomplis dans les méthodes de génie logiciel, le caractère intrinsèquement parallèle et réparti des modèles de programmation mis en œuvre continue à poser des problèmes ardus pour ce type de programmation.

La maîtrise du développement des applications parallèles et distribuées passe donc par le renforcement des activités de conception, validation et test. Le besoin d'environnements ayant pour objectifs de faciliter les étapes de conception et de développement ainsi que de garantir au maximum le déroulement correct de l'exécution pour ce type d'applications reste d'une grande actualité.

L'enjeu actuel est double : d'une part développer de nouvelles méthodes radicalement différentes pour permettre un développement aisé d'applications parallèles, et d'autre part permettre d'utiliser efficacement les techniques actuelles telles que les langages data parallèles, l'échange de messages ou l'invocation de méthodes.

De nombreuses actions de recherches sont actuellement orientées dans cette direction. Comme la liste des projets de recherches dans ce sens est importante, on se contentera de donner quelques exemples. Le projet *Ptools* [Ptools 99] aux *USA* labellise un certain nombre d'outils dédiés au parallélisme dans le monde. Le projet *HPFIT* [HPFIT 99] de l'*INRIA* se propose d'intégrer les outils développés en *France* autour de *HPF* [Koelbel 94]. Plusieurs projets européens sont également consacrés à ce type d'action, qui étudient plusieurs aspects. Pour chacun, des motivations simples permettent de les justifier :

- il faut libérer le programmeur des contraintes syntaxiques des nouveaux langages et modèles

de programmation parallèles ;
- il faut lui proposer des paquetages dédiés à des classes d'algorithmes avec un souci de réutilisabilité ;
- il faut proposer des outils d'évaluation des performances des algorithmes parallèles ;
- il faut masquer les différences de fonctionnement des machines parallèles.

Cependant les propositions qui existent sont souvent limitées aux machines massivement parallèles et aux langages data-parallèles. Des efforts ont été menés pour les applications distribuées sur des réseaux d'ordinateurs mais de façon plus lente, la raison principale étant que les modèles de programmation utilisés dans ce cadre n'ont pas encore atteint la maturité suffisante et restaient l'apanage d'experts.

Notre contribution, dans l'équipe *RESEDAS*, concerne un environnement de développement des applications distribuées.

4.2 L'environnement proposé

L'environnement que nous proposons doit aider l'utilisateur à développer des applications parallèles basées sur des modèles de programmation à mémoire distribuée et à les exécuter en utilisant au mieux les ressources machine et réseau disponibles. Rappelons que nous nous plaçons dans un contexte de réseaux hétérogènes de stations de travail.

La figure 4.1 présente l'architecture de l'environnement proposé, constitué de quatre composantes principales :

1. **MeDLey** (Message Definition Language): générateur pour le langage de spécification de messages. L'outil MeDLey constitue le cœur de cet environnement, son premier prototype a été développé dans le cadre du travail de thèse d'*Eric Dillon* [Dillon 97b]. L'utilisateur développe son application parallèle dans un langage cible (C, C++) et spécifie les messages dans le langage MeDLey. À partir de cette spécification, le générateur MeDLey fournit un ensemble de fonctions d'échange de données spécifiques à l'application et le moyen de communication choisi pas l'utilisateur.
2. **PlaTo** : module de placement de tâches. Pour exécuter une application parallèle, l'outil PlaTo calcule un placement définissant quelles tâches s'exécutent sur quelles machines. Afin d'obtenir de bonnes performances, il détermine ce placement en tenant compte du graphe des communications, de la topologie du réseau et de l'état courant de la machine virtuelle associée à cette application ;
3. **Analyse** : traitement des traces d'exécution. Afin d'augmenter les données contenues dans le graphe des communications, une génération des traces d'exécution est possible. Ainsi le module d'analyse enrichit le graphe avec des informations de type quantitatif, permettant à PlaTo de déterminer un meilleur placement pour l'application en question lors d'une exécution ultérieure.
4. **MinT** : outil d'administration de réseaux simplifié. Cet outil joue le rôle de gestionnaire de réseaux et systèmes pour le compte de l'environnement. Son objectif est de fournir des services de gestion minimaux au placement, en récoltant principalement les informations relatives à la machine virtuelle, telles que la topologie physique ou l'état de charge des équipements.

Les trois derniers outils ont été proposés et développés dans le cadre du travail de thèse de *Carlos Gamboa Dos Santos* [Gamboa Dos Santos 98]. Le prochain chapitre sera consacré à la description du langage MeDLey qui situe la base et le point de départ de notre travail de thèse.

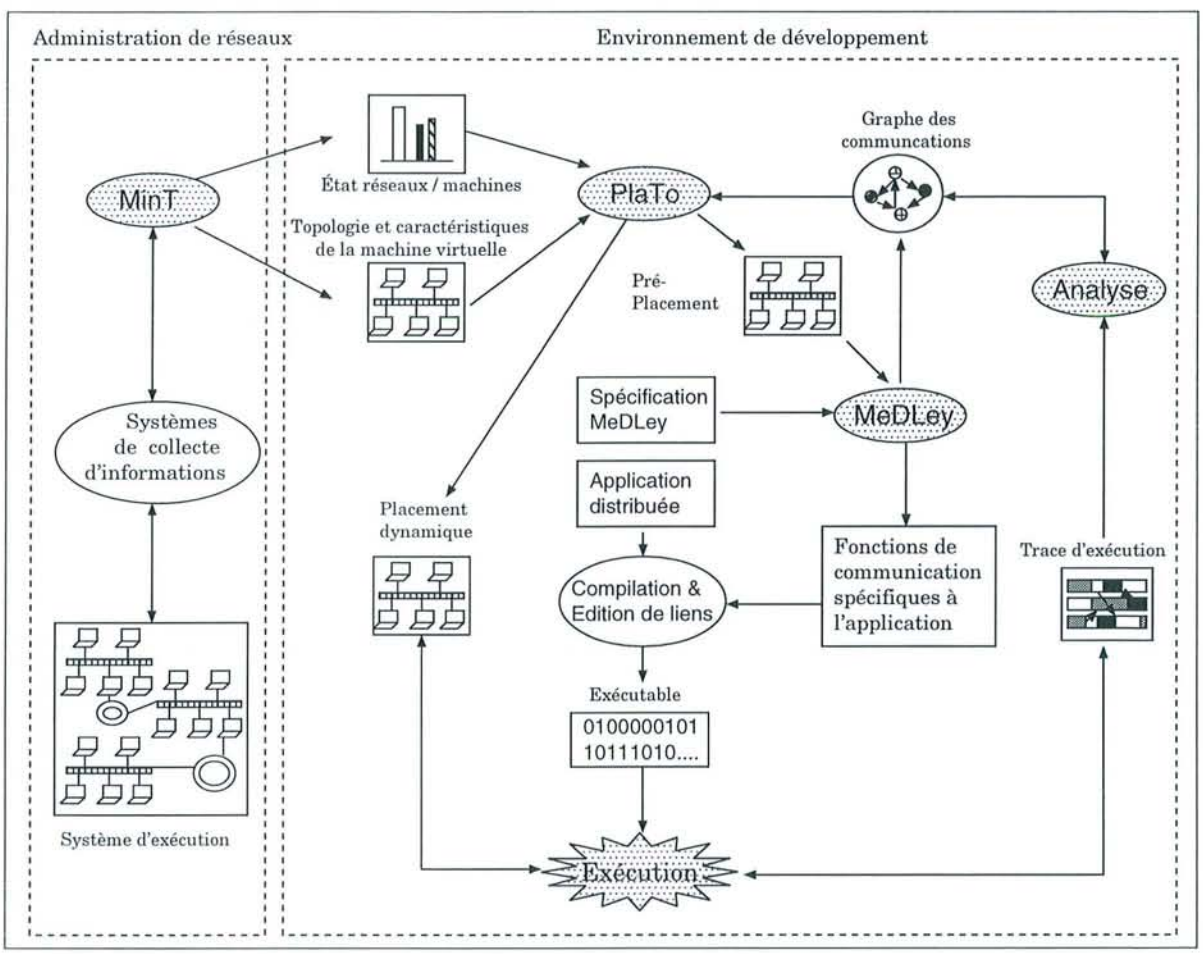

FIG. 4.1 – *L'architecture globale de l'environnement*

Chapitre 5

Le langage MeDLey

Ce chapitre a pour but de présenter brièvement l'approche MeDLey qui est destinée à spécifier les communications pour les applications de calcul distribué et parallèle. Nous allons présenter un résumé de la syntaxe et l'utilisation de ce langage. Pour plus d'informations sur sa syntaxe, le lecteur se reportera au document [Dillon 97b] ou au manuel de référence [Dillon 97a].

5.1 Introduction

Dés lors qu'une application parallèle suit le modèle des tâches communicantes, le programmeur est toujours confronté aux mêmes problèmes, quel que soit l'algorithme : structurer et subdiviser son application en tâches plus ou moins indépendantes, et implanter les communications entre elles selon les possibilités matérielles et logicielles.

Pour pouvoir l'aider dans sa tâche, il est nécessaire d'une part de lui offrir un moyen simple et abstrait de définir la structure de son application, et d'autre part de lui assurer des performances optimales lors des échanges entre ces tâches pour un temps de développement réduit.

Généralement, les applications parallèles basées sur le modèle de tâches communicantes sont programmées avec des langages séquentiels étendus par des mécanismes permettant les coopérations entre tâches séquentielles. Que ce soit par l'utilisation de bibliothèques de communication ou par d'autres mécanismes particuliers, les services offerts sont souvent identiques. On peut les regrouper en 3 catégories :

- les fonctionnalités permettant la gestion dynamique des tâches, i.e. naissance, mort, synchronisations ;
- les fonctionnalités d'identification de ces tâches, de la simple numérotation à d'autres mécanismes plus complexes comme des regroupements fonctionnels ou des topologies précises d'interconnexion ;
- les fonctionnalités d'échange d'informations entre tâches, depuis les échanges point à point à ceux collectifs.

La solution proposée par MeDLey est construite d'après cette constatation. Il s'agit de décrire certains de ces services de façon indépendante de tout langage ou implantation dans un formalisme simple qui pourra être utilisé par le programmeur dans la phase de conception de son application parallèle. Grâce à ce formalisme, il pourra décrire son application sans faire de choix a priori sur l'implantation future, en terme de tâches, et d'échange entre ces tâches. A partir de cette description de l'application, un outil dérivera de façon automatique une implantation de ces échanges entre tâches vers un langage séquentiel cible. En conséquence, il restera à la charge

du programmeur de réaliser la partie contrôle de son algorithme dans le langage cible choisi, toutes les parties concernant les extensions parallèles étant à la charge de l'outil.

Bien entendu, le but de cette approche ne doit pas se limiter à une implantation automatique des échanges entre les tâches d'une application. Le contexte de calcul hautes performances impose que cette implantation soit la plus efficace possible. En effet, les bibliothèques de communication offrent beaucoup de moyens d'optimiser les échanges grâce à une grande variété de fonctionnalités plus ou moins adaptées à chaque architecture. Cependant, le caractère exhaustif de ces bibliothèques oblige à avoir une certaine expérience avant d'obtenir des performances satisfaisantes. Dans ce cadre, pour obtenir des performances raisonnables, l'utilisateur d'une bibliothèque d'échange de messages doit réussir à concilier deux choses :

- d'une part les meilleures performances brutes de la bibliothèque de communication qu'il utilise, en choisissant au mieux parmi les primitives offertes en fonction de ses besoins, et de l'architecture matérielle dont il dispose ;
- d'autre part l'implantation de son algorithme de la façon la plus efficace pour obtenir le meilleur temps d'exécution possible.

Concernant les performances brutes des bibliothèques de communications, beaucoup de personnes s'intéressent à différentes grandeurs telles que le temps de latence, le débit, ou encore le surcoût des fonctions de communication [Keeton et al., 1995] [Liet al., 1995b] [Dillon et al., 1995b] . Cependant, dans le cas d'applications réelles, ces grandeurs peuvent parfois s'avérer difficiles à manipuler pour obtenir une véritable efficacité. En conséquence, l'efficacité globale d'une application n'est pas toujours aussi bonne qu'escomptée puisque l'interaction entre les communications et les calculs est parfois très forte. De plus, beaucoup de méthodes de communications existent, proposant des schémas de communications nouveaux, plus ou moins efficaces selon les cas, conduisant l'utilisateur à faire un choix qui ne pourra être bon que s'il possède, encore une fois, une certaine expérience.

Dans ce chapitre nous allons détailler le formalisme MeDL*ey*, en présentant auparavant le modèle de tâches, et d'échanges qu'il propose. A partir de ce formalisme, nous montrerons qu'il est possible de dériver de façon automatique une implantation des échanges ayant lieu au sein d'une application, selon l'architecture matérielle et logicielle disponible.

5.2 L'approche proposée par MeDL*ey*

L'approche proposée par MeDL*ey* est de décrire tout ce qui concerne les échanges entre les tâches d'une application parallèle, afin d'en dériver une implantation efficace de façon automatique. Il s'agit donc de fournir à l'utilisateur, une vision abstraite de son application en termes de tâches et d'échanges entre ces tâches. L'approche peut donc être présentée selon deux points de vue :

- la structuration d'une application basée sur le modèle de tâches communicantes ;
- la description des échanges entre ces tâches.

Nous allons commencer par présenter ces deux points de vue.

5.2.1 La structuration de l'application avec MeDL*ey*

La structuration d'une application parallèle est souvent basée sur le modèle *MPMD*, dans lequel différentes tâches s'exécutent en parallèle. Néanmoins, de nombreuses applications utilisent

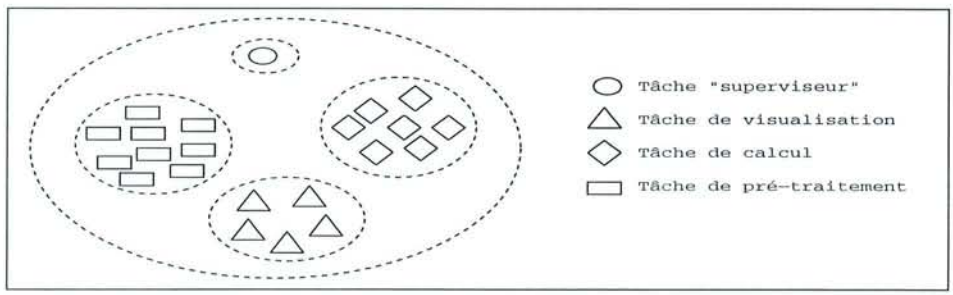

FIG. 5.1 – *Le modèle* M-SPMD

une approche plus simple, dans laquelle les tâches sont toutes identiques, c'est-à-dire de type *SPMD*.

Pour concilier ces deux types de programmation parallèle, MeDLey introduit le modèle Multiple -Single Program Multiple Data- (*M-SPMD*). Dans ce modèle, différents types de tâches peuvent s'exécuter simultanément, avec un nombre de tâches variable par type. L'idée est d'associer à chaque sous-application *SPMD* un représentant. Cette idée est à rapprocher de la terminologie que l'on trouve dans les langages orientés objets, ou l'on parle de classes et d'instances de ces classes ou objets. Nous allons utiliser une terminologie identique dans MeDLey. Une application sera décrite statiquement en termes de représentants de tâches qui seront instanciés à l'exécution.

Prenons l'exemple où l'on considère une application parallèle scindée en quatre parties distinctes :

- un sous ensemble de tâches effectuant un pré-traitement sur des données, obéissant à un modèle *SPMD* ;
- un second sous-ensemble de tâches effectuant un calcul sur ces données pré-traitées, ici encore, selon un modèle *SPMD* ;
- un troisième sous-ensemble de tâches chargées d'afficher les résultats finaux, toujours selon un modèle *SPMD* ;
- enfin, une tâche agissant comme un superviseur, permettant par exemple de suivre l'évolution du calcul global.

Cette application, à l'exécution, peut être représentée comme sur la figure 5.1. On en retrouve les quatre ensembles de tâches. Il est évident que selon la taille des données à traiter la cardinalité des familles de pré-traitement, calcul et affichage va varier.

Une description MeDLey de cette application va se faire en oubliant la cardinalité des différents sous-ensembles. En effet, nous allons associer à chacun de ces sous-ensembles un représentant MeDLey, de manière à obtenir une représentation statique de l'application. Cette représentation est schématisée sur la figure 5.2.

De cette manière, l'utilisateur dispose d'une représentation abstraite de son application, qui n'est en particulier pas surchargée par les différentes cardinalités dépendant du volume de données à traiter.

Enfin, MeDLey offre un raffinement concernant la structuration de l'application, en permettant de préciser la topologie d'interconnexion des tâches au sein de chaque sous-ensemble *SPMD*.

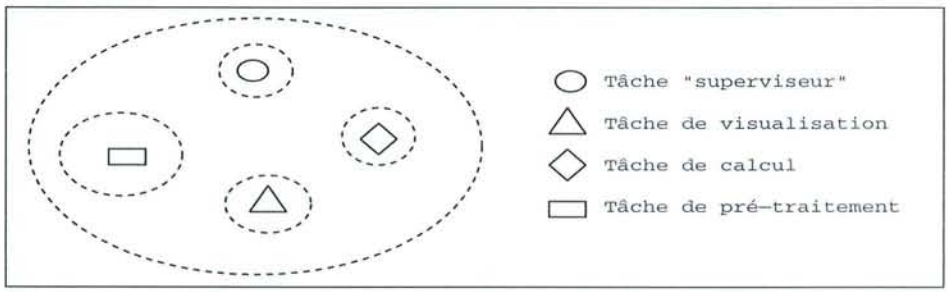

FIG. 5.2 – *Description* MeDLey *d'une application* M-SPMD

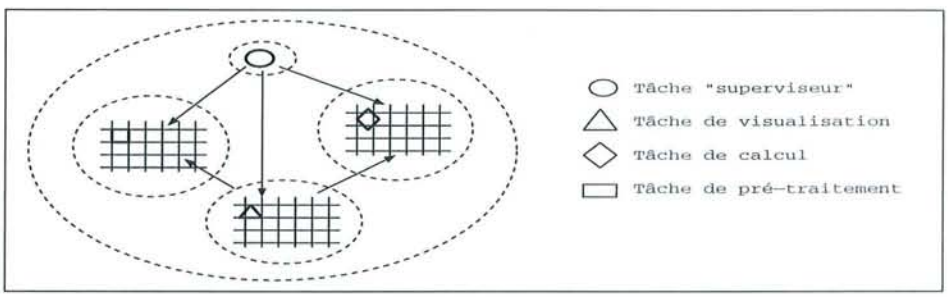

FIG. 5.3 – *Les échanges entre tâches avec* MeDLey

5.2.2 Les échanges entre tâches avec MeDL*ey*

La deuxième partie d'une description MeDL*ey* permet de décrire les interactions entre ces tâches, aussi bien au niveau des représentants que des instances elles-mêmes. En effet, le formalisme MeDL*ey* permet de préciser de façon statique comment les représentants de tâches sont interconnectés, i.e. quels groupes de tâches vont échanger des informations. De cette manière, la description MeDL*ey* inclut la notion de graphe de communication entre les représentants. Cette information supplémentaire est représentée sur la figure 5.3. Les noeuds du graphe correspondent aux représentants des tâches, alors que les arcs symbolisent les connexions entre ces représentants de tâches.

Enfin, en plus de cette description statique de l'application, MeDL*ey* permet de définir explicitement les interactions entre les représentants des tâches et entre leurs instances. En effet, pour chaque représentant de tâche MeDL*ey*, trois informations doivent être précisées :

- les variables privées impliquées dans les échanges pour ce représentant de tâches ;
- les envois de données vers les autres tâches ou représentants de tâches ;
- les réceptions de données en provenance d'autres tâches ou représentants de tâches.

De plus, puisque le formalisme se veut abstrait et complètement indépendant de toute architecture matérielle, comme de tout langage cible, MeDL*ey* propose ses propres notations pour déclarer les variables locales à chaque tâche. A partir de ces déclarations seront construits tous les échanges de la tâche. Ici encore, aucune supposition n'est faite sur la construction des échanges à partir des variables locales, la syntaxe proposée permet des définitions indépendantes de tout langage ou bibliothèque de communication. Chaque représentant de tâche MeDL*ey* comporte donc quatre champs :

- un préambule, précisant le nom du représentant que l'on définit, et ses interconnexions avec les autres représentants de tâches ;

- une zone permettant la définition des variables qui seront instanciées dans chaque instance de ce représentant de tâche ;
- deux zones définissant les données pouvant être envoyées et reçues par chaque instance de ce représentant de tâche.

5.3 Présentation du langage

La syntaxe d'une spécification de communication en utilisant le langage MeDLey comporte principalement deux points :

- tout ce qui permet de décrire la structure de l'application en termes de tâches communicantes. Cette partie fait référence à la définition des modules MeDLey ;
- tout ce qui permet de décrire explicitement les échanges entre ces tâches. Cette partie concerne le contenu des modules MeDLey.

Nous allons donc tout d'abord présenter le squelette général d'une spécification de communications avec MeDLey.

5.3.1 La définition d'une spécification MeDLey

MeDLey fournit une vue statique d'une application parallèle, sous forme de représentants de tâches qui seront instanciés à l'exécution. Ces représentants sont définis de façon purement statique, avec des structures de données privées locales à chaque instance de tâche, et des échanges de données pouvant être déclenchés depuis le langage cible choisi. Un représentant de tâche MeDLey est défini par son nom, et contient quatre parties :

1. un préambule, permettant de définir le représentant lui-même, ainsi que ses interactions avec les autres représentants de l'application ;
2. le bloc *uses*, qui permet de définir les données locales à chaque instance de la tâche ;
3. le bloc *sends*, qui permet de définir les envois de données vers d'autres instances de tâches ;
4. le bloc *receives*, qui permet de définir les réceptions de données provenant d'autres instances de tâches.

Le squelette d'une spécification de communications avec MeDLey se présente donc schématiquement comme l'exemple décrit dans la figure 5.4. Cet exemple définit les représentants de deux tâches constituant une application *MPMD*.

Chaque définition d'un représentant de tâche sera appelée module MeDLey dans ce qui suit. Détaillons rapidement le contenu des quatre parties d'un module MeDLey.

Le préambule

Même si MeDLey ne fournit qu'une description statique d'une application parallèle, il fournit aussi des informations sur la manière dont vont être organisées les instances de tâches à l'exécution. Ces schémas d'interconnexion sont précisés au début de chaque module MeDLey par la clause *connected with*. Plusieurs topologies d'interconnexions sont disponibles dans MeDLey, elles sont précisées par différents mot-clés :

Le premier mode d'interconnexion correspond au modèle *SPMD*. Dans ce cas, la partie connected est optionnelle. En voici un exemple :

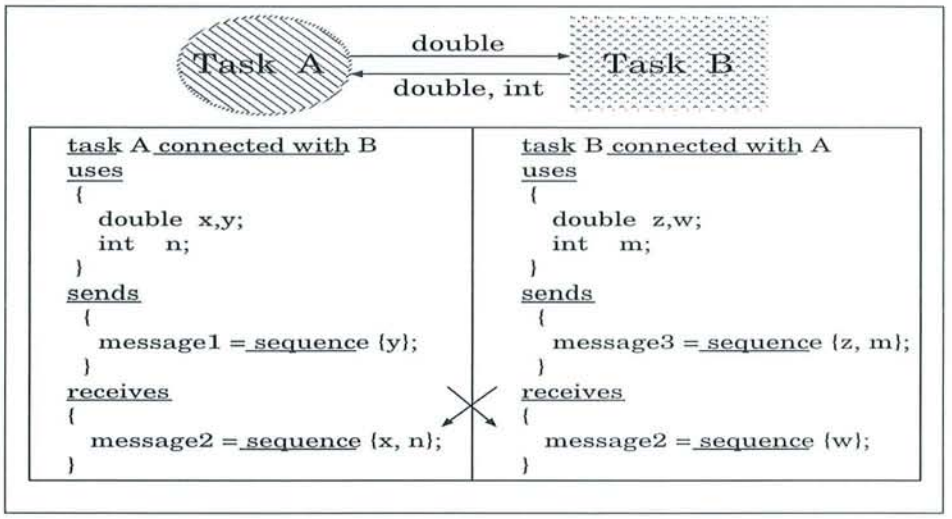

FIG. 5.4 – *Exemple d'une spécification de communications avec MeDLey*

```
Task agent(i) connected with setof(agent) // ou seulement : Task agent(i)
```

Le mot setof désigne que plusieurs instances de la tâche spécifiée (agent dans notre exemple) seront instanciées pendant l'exécution. Chacune de ces instances sera identifiée par son rang et peut envoyer et recevoir des messages des autres instances.

Le deuxième mode d'interconnexion correspond au modèle *M-SPMD*. Dans ce mode, chaque tâche peut être connectée aux autres tâches énumérées dans la partie *connected* comme pour l'exemple suivant :

```
Task agent(i) connected with setof(clients, autres)
```

Par conséquent, chaque instance de la tâche *agent* peut envoyer et recevoir des messages de chaque instance des tâche *clients* et *autres*.

Le troisième mode d'interconnexion correspond au modèle *MPMD*. Chaque tâche sera instanciée une seule fois pendant l'exécution. Un exemple est présenté ci-dessous :

```
Task client connected with agent
```

La clause *uses*

Cette partie d'un module MeDLey doit être utilisée pour déclarer les données locales qui devront être instanciées pour chaque tâche de ce type à l'exécution. Ces données servent de support à la définition de tous les échanges avec d'autres tâches. Toutes les variables déclarées ici seront disponibles à l'exécution dans le langage cible choisi. De plus, c'est le contenu de ces variables qui sera effectivement envoyé vers les variables des autres tâches selon les définitions des différents échanges. Bien entendu, à l'exécution l'utilisateur pourra utiliser ces variables comme n'importe quelles autres variables de son programme, mais seules celles-ci participeront aux échanges avec les autres tâches.

Les clauses *sends* et *receives*

Ces deux clauses permettent de définir respectivement les échanges de données vers d'autres tâches et en provenance d'autres tâches. Dans les deux cas, chacune des définitions va donner

lieu à la création d'une fonction associée par l'outil de génération automatique. La syntaxe de définition des échanges est présentée dans le paragraphe suivant.

5.3.2 La définition des échanges entre tâches MeDLey

La définition des échanges entre tâches MeDLey correspond au corps d'un module MeDLey. Le principe de définition d'un échange est simple. Il peut être un envoi, i.e. le contenu de variables locales sera envoyé vers les variables locales d'une autre tâche, ou inversement une réception, i.e. où certaines variables locales seront modifiées par l'arrivée de données.

Ensuite chaque définition est basée sur l'utilisation des variables déclarées dans la partie *uses* du module MeDLey. Classiquement, lorsqu'on utilise une bibliothèque d'échange de messages, le contenu des messages échangés doit être construit en copiant le contenu de variables dans un buffer d'envoi avant l'appel à la primitive d'envoi à proprement parler. Ici, l'idée est que cette étape de construction doit rester cachée pour l'utilisateur : tout ce dont l'utilisateur doit se soucier est que le contenu de telle ou telle variable locale à une tâche sera envoyée vers telle autre variable d'une autre tâche. En conséquence, tous les échanges ne se feront qu'à partir des variables déclarées dans la partie *uses*. Présentons rapidement la syntaxe de déclaration de ces variables dans la partie *uses* avant d'envisager la définition des échanges.

Les données dans MeDLey: Toutes les variables utilisées dans un module MeDLey doivent être déclarées dans la partie *uses* de ce module. MeDLey fournit schématiquement deux familles de types de données :

1. les types de base (int, double, float) : ils sont fournis par le langage lui-même et n'ont pas besoin d'être déclarés. Ils peuvent être utilisés à l'intérieur de la section *task*, dans la section *uses* que nous allons voir plus tard ;
2. les types construits : ils sont subdivisés en deux catégories. Il y a les matrices et les vecteurs d'une part et les structures d'autre part. Les matrices et les vecteurs doivent être utilisés dans la partie *uses*. La syntaxe à utiliser est très simple et suit des règles claires. En voici des exemples :

   ```
   matrix<float>[100,50] : définit  une matrice à  deux dimensions de
   nombres réels en simple précision.

   vector<int>[200] : définit un vecteur de 100 entiers.
   ```

   Ces types de données sont bien entendu implantés sous forme de tableaux dans les langages cibles. Concernant les structures, elles doivent être déclarées avant d'être utilisées. La syntaxe de la déclaration des structures est celle du langage C.

Définition d'échanges avec MeDLey: Tous les échanges avec MeDLey sont définis en termes de messages, soit comme des messages à destination d'autres tâches (partie *sends*), soit comme des messages en provenance d'autres tâches (partie *receives*). Dans les deux cas, chaque définition de message se fait en précisant 3 composantes :

- son nom : il s'agit d'un identificateur pour le message. Cette chaîne de caractères est utilisée ensuite pour la dérivation automatique de la fonction associée à ce message ;

- sa destination (resp. sa provenance). Cette composante permet non seulement de préciser le destinataire ou l'émetteur du message, mais aussi le type de message en question, i.e. s'il s'agit d'une communication point à point ou collective (diffusion/multi-diffusion). En effet, la syntaxe est la suivante :

```
m1 to A    = ...
m2 to {A}  = ...
m3 to [A]  = ...
```

  Dans cet exemple, le message m1 définit une communication point à point vers une instance de la tâche A. Dans ce cas, le numéro de l'instance concernée sera précisé à l'exécution. Le message m2 définit une diffusion vers toutes les instances de A, alors que le message m3 définit une multi-diffusion vers un sous-ensemble des instances de A. Ici encore, ce sous-ensemble sera précisé à l'exécution ;
- son contenu. Il sera basé sur les définitions de la partie *uses* du module. Supposons dans tout ce qui suit, que le module que l'on définit possède les déclarations de données suivantes :

```
uses
{
   vector<float>[100]     V;
   matrix<int>[50][60]    M;
   int i,j;
   float f1, f2;
}
```

Toute définition d'échange avec MeDLey est basée sur le même constructeur (sequence), qui permet de construire le contenu d'un message à partir d'un ensemble de variables locales. Par exemple,

```
m1 to A = sequence{i, j, f1};
```

Définit un message contenant 3 valeurs, deux entiers, et un nombre réel. Si cette définition est faite dans la partie *sends* du module, elle signifie que le message m1 devra être envoyé à une instance de A, et devra contenir les valeurs des trois variables précisées. Par contre, si cette définition est faite dans la partie *receives* du module, elle signifie que le message m1 sera en provenance d'une instance de A et que son contenu devra être stocké, dans l'ordre, dans les trois variables locales précisées.

Enfin, il existe des constructions un peu différentes dans le cas de types MeDLey construits comme les vecteurs ou matrices. Par exemple, il est possible d'envoyer non seulement le vecteur ou la matrice complète, mais aussi des sous-blocs de ces derniers, comme dans l'exemple suivant :

```
m1 to A = sequence{M, V};
m2 to A = sequence{M[4..10][30..35]};
m3 to A = sequence{V[..]};
```

Dans cet exemple, le message m1 contient la matrice M et le vecteur V, le message m2 contient un bloc fixe de la matrice M, enfin le message m3 extrait un bloc du vecteur V dont les bornes seront précisées à l'exécution. Bien entendu, toutes les combinaisons de ces notations sont possibles.

Enfin, on peut encapsuler les déclarations des messages dans des blocs *async* ou *sync* qui permettent de définir si un message doit être envoyé ou reçu de façon synchrone ou asynchrone. Par défaut, toutes les définitions d'envois (partie *sends*) donneront lieu à des fonctions asynchrones, alors que les définitions de réceptions (partie *receives*) donneront lieu à des fonctions synchrones.

5.4 Dérivation automatique de bibliothèque de communications

Le formalisme MeDLey permet à l'utilisateur de décrire les échanges ayant lieu au sein de son application parallèle en vue de dériver de façon automatique une implantation de ces échanges. Cette implantation doit s'orienter selon deux directions :

L'implantation des tâches : En effet, tout le formalisme est basé sur l'utilisation de représentants et d'instances de tâches. Plusieurs façon d'instancier ces tâches sont possibles a priori selon l'architecture de l'environnement d'exécution disponible. Il faut d'ailleurs noter que ces définitions ont été réorganisées de manière à apparaître selon les opérations d'envoi des messages.

L'implantation des communications : à partir de support logiciel de communication disponible il faut dériver une implantation efficace des échanges entre ces tâches.

5.4.1 Implantation des tâches

Un module MeDLey définit un représentant pour une tâche. Pour donner lieu à une exécution, ce représentant doit être instancié une ou plusieurs fois.

En conséquence, il convient de choisir une forme pour ces instances, qui seront effectivement manipulées par le programmeur. Nous nous plaçons dans le cadre d'un environnement UNIX.

Deux possibilités peuvent être envisagées pour l'implantation d'instances de tâche MeDLey :

1. comme un processus UNIX, i.e. avec son propre contexte ;
2. comme un processus léger UNIX, i.e. où plusieurs flux de contrôle partagent le même contexte.

D'après l'approche proposée par MeDLey, c'est-à-dire basé sur une mémoire distribuée privée à chaque instance de tâche, la première solution est la seule indiquée. Ce qui signifie que pour chaque instance de module MeDLey, on aura un seul flux de contrôle, dans lequel seront déclarées toutes les variables de la partie *uses* du module. En conséquence, chaque instance disposera de son propre contexte et donc de ses propres variables privées.

Puisqu'il s'agit de proposer à l'utilisateur un moyen de programmer son application parallèle, abordons enfin les points généraux concernant les tâches avec MeDLey comme dans toute bibliothèque d'échange de messages :

- **création des tâches :** aucune partie dynamique n'est actuellement prévue dans le formalisme MeDLey. Dans le prototype réalisé, le démarrage des tâches se fait de façon complètement statique. Ceci signifie qu'au démarrage de l'application, l'utilisateur doit préciser combien d'instances de chaque tâche doivent être lancées, après quoi tout sera fixé. Enfin, toutes les instances d'une même famille exécuteront le même code. Bien entendu, ce choix est complètement arbitraire et interdit beaucoup de champs d'application, ainsi que d'autres domaines tels que la tolérance aux fautes par exemple. Cependant, puisque MeDLey n'offre qu'une vision statique d'une application, il est tout à fait envisageable de fournir une interface dynamique pour la création des tâches MeDLey. Nous considérerons que cet aspect ne fait pas partie de MeDLey, mais de l'implantation de MeDLey réalisée ;

- **identification des tâches :** elle se fait à l'exécution par des couples (famille, numéro d'instance dans la famille). De plus, comme chaque définition de message précise en particulier la famille de tâches destinatrices, l'utilisateur n'aura qu'à utiliser les numéros d'instances dans ces familles ;
- **groupement des tâches :** le groupement des tâches avec MeDLey est complètement implicite, et transparent pour l'utilisateur puisque les tâches sont regroupées par familles possédant le même représentant dans la description MeDLey ;

5.4.2 Implantation des échanges

L'implantation des échanges dérivée de la description MeDLey d'une application parallèle se doit être efficace et optimale. En conséquence, il faut pouvoir tirer parti au maximum de l'architecture matérielle et logicielle de communication disponible. Le premier mécanisme utilisé pour cette implantation est celui d'échange explicite de messages en utilisant les bibliothèques d'échange de messages (voir chapitre 2). Mais tout autre mécanisme de communication peut être utilisé pour implanter les échanges.

Dans ce modèle d'implantation, l'identification des tâches pourra se faire directement en utilisant les fonctionnalités de la bibliothèque utilisée, tant au niveau de la numérotation des instances que des mécanismes de groupement des tâches. Concernant les fonctions de communications, chaque échange de message MeDLey pourra être implanté directement par les fonctions d'envoi ou de réception fournies par la bibliothèque.

La première version du prototype de générateur automatique MeDLey-0 à été réalisée en implantant les communications avec MPI. Bien entendu, cette version du prototype inclut également les phases d'analyse syntaxique et sémantique à partir de modules MeDLey. La première version du prototype de générateur automatique MeDLey a été développée de façon à valider l'approche proposée. Il s'agissait donc de générer une implantation automatique des fonctions de communications à partir d'une description réalisée dans le formalisme MeDLey. Cette implantation peut être schématisée selon trois modules :

- un module d'analyse syntaxique du langage MeDLey ;
- un module d'analyse sémantique, effectuant plusieurs contrôles sur les descriptions avec le langage MeDLey ;
- un module de génération de code avec *MPI*.

En plus des contrôles classiques effectués lors de l'analyse syntaxique et sémantique, le prototype MeDLey ajoute un contrôle testant si les messages envoyés peuvent être reçus, i.e. le générateur vérifie que pour chaque définition de message à envoyer, il existe une définition de message permettant la réception de ce message.

Ensuite, chaque définition de message donnera lieu à la définition d'une fonction d'envoi ou de réception. Pour chacune de ces fonctions, l'implantation sera basée sur l'utilisation de primitives MPI. En particulier, de manière à optimiser les performances de communication pour chaque fonction, la construction de chaque message est basée sur la construction de types dérivés MPI. Enfin, en plus de chaque fonction de communication, plusieurs fonctions de "service" sont générées. Ces fonctions incluent des possibilités de synchronisation, d'identification des tâches, et deux fonctions d'initialisation et de terminaison. Le profil de toutes ces fonctions est directement inspiré du profil des fonctions MPI. Ce prototype a été utilisé pour développer les premiers exemples avec MeDLey. Nous présentons dans la suite l'un d'entre eux.

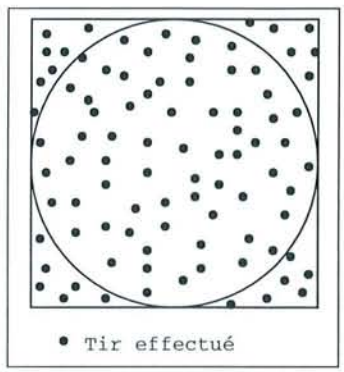

FIG. 5.5 – *Principe de la méthode de* Monte Carlo

5.5 Exemple d'application avec MeDL*ey*

Afin d'améliorer la compréhension de ce qui précède, nous proposons de décrire un petit exemple de programme. L'application va calculer une approximation de π par la méthode connue sous le nom de *Monte Carlo* (voir la figure 5.5). Elle est basée sur le fait que la surface d'un cercle de rayon r est égale à $\pi * r * r$. On tire au hasard des points dans le carré circonscrit (de surface $4 * r * r$). En mesurant le rapport entre le nombre de tirs dans le cercle par rapport au nombre de tirs total, il est possible d'obtenir une approximation de π, en considérant que :

$$lim_{TotalDeTirs \to +\infty} \frac{TirsDansLeCercle}{TotalDeTirs} = \frac{SurfaceCercle}{SurfaceCarr} = \frac{\pi * r * r}{4 * r * r} = \frac{\pi}{4}$$

L'algorithme permettant d'implanter cette méthode est donc très simple. Il suffit de déterminer une tâche chargée de comptabiliser les tirs, et d'utiliser un ensemble de tâches chargées d'effectuer, en parallèle, un nombre suffisant de tirs pour considérer que ce nombre est proche de $+\infty$.

En utilisant MeDL*ey*, cette application se traduit par la définition de deux représentants de tâches :

1. un représentant nommé *father* (ou superviseur), instancié une seule fois, chargé de distribuer aux tâches le nombre de tirs qu'elles doivent effectuer, et de comptabiliser les résultats ;
2. un représentant nommé *son* (ou travailleur), qui sera instancié plusieurs fois, pour effectuer les tirs en parallèle.

Le contenu des deux modules correspondant à la définition de chaque représentant est précisé dans le reste de cette section.

```
Task  father  connected with setof (son)
uses
        {
             double          box, coef;
             int             number;
             int             in;
             int             who;    // le rang de l'instance son
        }
```

```
sends
        {
            shootMessage to {son} = sequence{number};
        }
receives
        {
            startMessage from son = sequence{box,coef};
            inMessage    from son = sequence{in,who};
        }
```

Nous pouvons noter que le module de définition de la tâche superviseur définit la topologie d'interconnexion avec les tâches travailleuses (*son*) chargées d'effectuer les tirs. Dans cet exemple, on remarque qu'il y aura à l'exécution une seule tâche *father* pour un ensemble de tâches *son* (mot clé setof).

Dans la tâche *father* un certain nombre de variables sont déclarées. Elles seront directement utilisées depuis le programme final. Enfin, les définitions de messages reflètent l'algorithme évoqué :

- un premier message (shotMessage) émis aux tâches *son* (diffusion) précisant le nombre de tirs que chacune des tâches doit effectuer ;
- un message (startMessage) précisant que les tâches *son* commencent le calcul ;
- un message qui sera reçu de la part de toutes les tâches *son* précisant le nombre de tirs dans le cercle, et l'instance de la tâche qui a effectué ces tirs (de manière à comptabiliser quelle tâche a tiré, pour une statistique de répartition de charge).

Le module de définition des tâches *son* est présenté ci-dessous.

```
Task  son(i)  connected with father
uses
       {
             double          box,coef;
             int             in;
             int             shootNumber;
             int             who;        //  le rang de l'instance travailleuse
       }
sends
       {
            oneShoot to father = sequence{in,who};
            startMsg to father = sequence{box,coef};
       }
receives
       {
            numberOFShoot from father = sequence{shootNumber};
       }
```

Dans ce module, nous retrouvons les déclarations de variables locales, qui seront instanciées dans chaque instance de la tâche *son*, puis les différents messages qui seront échangés avec la tâche *father*.

Nous pouvons remarquer en particulier, que chaque définition de message en émission du module *father* correspond à une définition de message en réception dans le module *son*.

A partir de ces deux définitions de modules MeDLey, le générateur automatique de fonctions va créer plusieurs fichiers selon le langage cible choisi. Envisageons le cas où le langage cible choisi est le langage C++. Dans ce cas, à chaque module MeDLey, une classe sera créée. A l'exécution, chaque tâche aura donc à instancier l'un de ces deux objets selon qu'elle est une tâche de calcul *son* ou la tâche superviseur *father*.

Chaque définition de message dans un module va se traduire par la création d'une méthode pour l'objet correspondant, permettant de réaliser l'envoi ou la réception décrits en MeDLey.

C'est ainsi que l'objet correspondant au module *father* contiendra les définitions présentées ci-dessous :

```
class father:
{
public:
// déclarations des données
int in;
int who;
double box;
double coeff;
int number;

// méthodes usuelles
int MDL_Init(int* argc, char*** argv); // Fonction d'initialisation
int MDL_MyRank(); // Retourne le rang de l'instance courante
int MDL_FamilySize(); // Le nombre d'instances de la tâche father
int MDL_syncWithFamily(); // Pour synchroniser la famille
int MDL_sonSize(char* name = "son"); // Nombre d'instances de la tâche son
int MDL_End();  // Fonction de terminaison

// Fonctions d'envoi

int MDL_SendTo_son_shotMessage();
int MDL_shotMessageTest();

// Fonctions de réception
int MDL_RecvFrom_son_startMessage();
int MDL_RecvFrom_son_inMessage();
private:
...
};
```

Parmi les définitions public de la classe, nous trouvons toutes les déclarations de variables présentes dans le module MeDLey. Ensuite, on trouve un ensemble de fonctions de service permettant l'identification des tâches, ainsi que des fonctions d'initialisation (MDL_Init) et de terminaison (MDL_End).

Enfin, pour chaque définition de message, une méthode associée est générée. Cet exemple présente ici deux cas : un envoi asynchrone et des réceptions synchrones.

Dans le cas des réceptions synchrones, chaque définition correspond à une fonction qui sera bien entendu bloquante. Dans le cas de l'envoi asynchrone, deux fonctions ont été générées :

- MDL_SendTo_son_shotMessage : il s'agit d'une fonction non bloquante permettant de démarrer l'envoi ;
- MDL_shotMessageTest : cette fonction permet de tester si l'envoi a été réalisé et terminé.

De la même manière dans le cas de réception asynchrone, deux fonctions seraient générées, l'une pour tester l'arrivée du message, l'autre pour recevoir effectivement les données.

Toutes les fonctions définies dans cette classe peuvent être utilisées dans le code de l'application finale. Cela se fait par la création d'une instance de la classe *father* dans le code du programme *father* et une instance de la classe *son* dans le programme *son*, puis l'appel aux fonctions de communication sur ces instances. Le code source complet de l'exemple de *Monte Carlo* en C++ se trouve dans l'annexe A.

5.6 Le point par rapport à d'autres approches

Avant de parler d'avantages et d'inconvénients de cette approche, il convient de comparer rapidement la proposition de MeDLey avec d'autres approches destinées à faciliter la construction d'application par tâches communicantes. Dans un premier temps, il convient d'éliminer toutes les approches basées sur tel ou tel langage séquentiel, comme Para++ [Coulaud 99], OOMPI [OOMPI 00], mpC [Lastovetsky 00], etc. En effet, le premier but de MeDLey est d'être abstrait et indépendant de tout langage ou architecture matérielle. De plus, une autre différence avec des outils similaires est que MeDLey ne se préoccupe que de l'implantation et de l'optimisation des communications. La partie contrôle, précisant notamment comment sont ordonnancées les communications, est complètement laissée à la charge du programmeur dans le langage cible qu'il aura choisi. Par exemple, MeDLey n'impose pas de mode de fonctionnement comme BSP [Skillicorn 96] par exemple qui impose un déroulement très synchrone des différentes tâches.

5.7 Conclusion

L'utilisation de MeDLey pour décrire les échanges entre tâches permet d'extraire des informations concernant les échanges qui ne sont généralement pas présentes lorsqu'on utilise une bibliothèque générale d'échange de messages, cela pour en dériver une implantation efficace.

Cependant, l'approche MeDLey reste encore pauvre pour couvrir la plupart des besoins des applications de calcul distribué et parallèle. En effet, la syntaxe proposée par le langage MeDLey présente un choix très restreint au niveaux de types de communications ainsi que les topologies d'interconnexion. Par conséquent, des extensions de ce langage semblent nécessaires pour l'enrichir. Ces extensions font l'objet des deux prochains chapitres de ce rapport. La première extension concerne les communications collectives et la deuxième permet d'utiliser les méthodes de décomposition de domaine.

D'autre part et au niveau implantation, il est évident que la programmation par échange de messages en général, et la bibliothèque *MPI* en particulier sont largement utiliseés au niveau des applications du calcul distribué et parallèle. Cependant, d'autres environnements de programmation pour ce type d'application existent ou ont vu le jour. L'utilisation de MeDLey dans ces environnements semble aussi intéressante pour permettre à l'utilisateur un choix plus vaste et une certaine évolutivité de ce langage. Dans ce cadre, nous présentons dans la quatrième partie de ce rapport l'approche proposée pour utiliser ce langage dans un environnement *CORBA*.

Troisième partie

Extension de MeDLey

Chapitre 6

Extension de MeDLey pour les communications collectives

6.1 Introduction

Contrairement aux opérations point à point qui permettent à deux processus d'échanger des données, les opérations collectives mettent en œuvre une communication où plus de deux processus peuvent prendre part. Ces opérations peuvent être classées en trois catégories :

- les opérations collectives de transfert de données ;
- les opérations de synchronisation ;
- les calculs collectifs.

Dans les paragraphes suivants, nous allons présenter les fonctionnalités offertes par MeDLey dans chacune de ces catégories. Nous décrivons ensuite les extensions de MeDLey proposées dans ce domaine [Es-sqalli 00a], et nous terminons par la description d'un exemple d'application utilisant les nouvelles fonctions.

6.2 Les opérations collectives de transfert de données

Cette première catégorie regroupe les opérations qui transfèrent des données entre plusieurs processus. Ces opérations sont les suivantes : la diffusion (broadcast ou one to all), la distribution (scattering), le rassemblement (gathering), le commérage (allgather) et la transposition (all to all ou multi-scattering). Le lecteur trouvera plus de détails sur ces opérations dans la suite de ce chapitre ; nous nous contentons actuellement de présenter par des schémas représentatifs le principe de ces opérations (voir figure 6.1).

6.2.1 Les fonctions existantes

MeDLey fournit deux fonctions collectives de transfert de données [Dillon 97a] :

- l'opération de diffusion qui permet de diffuser un message vers l'ensemble des instances d'une tâche. En voici un exemple :

```
m1 to {sun} = ...
```

  Ce message définit une diffusion vers toutes les instances de la tâche *sun* ;

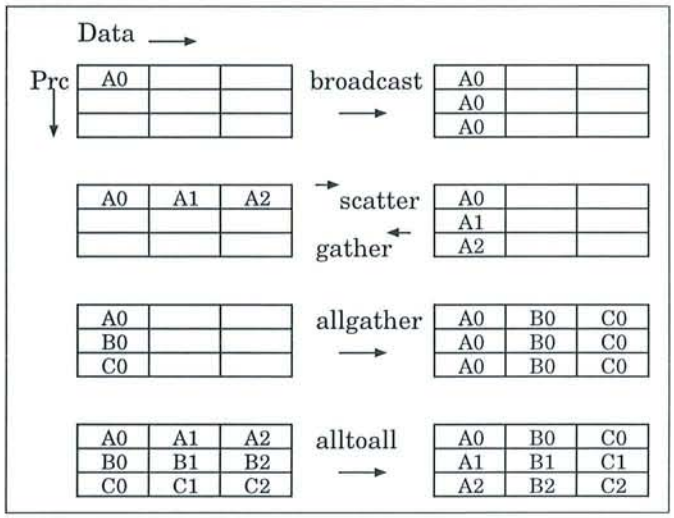

FIG. 6.1 – *Les opérations collectives de transfert de données*

- l'opération de multi-diffusion qui permet de diffuser un message sur un sous-ensemble des instances d'une tâche qui doivent être spécifiées au moment de l'exécution comme pour l'exemple suivant :

```
m1 to [sun] = ...
```

6.2.2 L'extension de MeDLey

Structure de la tâche

Notre extension est basée sur les fonctions de la bibliothèque de communication MPI [Forum 95] dans ce domaine. Le modèle de programmation choisi est un modèle SPMD (Single Program Multiple Data). Les différentes opérations s'effectuent entre les instances de la même tâche.

Pour donner un aspect plus particulier à ces opérations, nous proposons que la déclaration des différents messages leur correspondant se fasse dans un nouveau bloc appelé *collectives*. Chaque ensemble d'opérations de même type doit être déclaré dans un sous-bloc correspondant à ce type :

```
Task  ..... // nom de la tâche
uses
{
  ..... // déclaration des données à utiliser pendant
        // la communication et le calcul
}
sends
{
  ..... // déclaration des messages à envoyer
}
receives
{
  ..... // déclaration des messages à recevoir
}
```

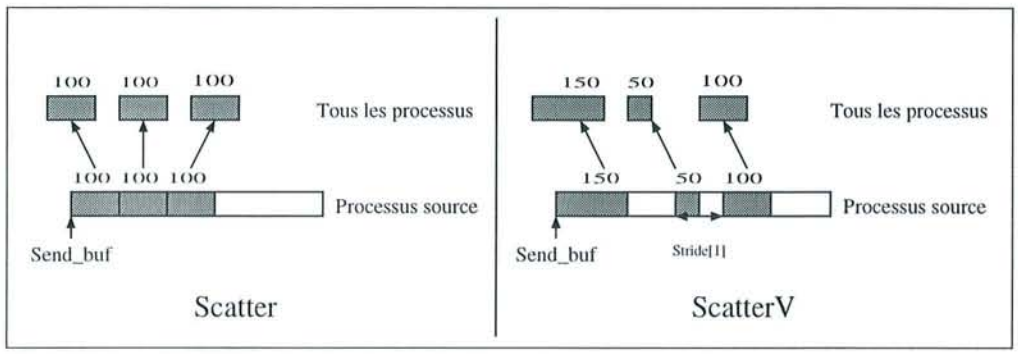

FIG. 6.2 – *Différence entre Scatter et ScatterV*

```
collectives // déclaration des messages correspondant
            // aux opérations collectives
{
  Bcast // pour l'opération de la diffusion
   {
     ..... // déclaration des messages à diffuser
   }
  ..... // déclarations des autres opérations utilisées~:
        // Scatter, Gather, Alltoall ....
}
```

Après la phase de spécification, MeDL*ey* génère pour chaque message une fonction du nom *MDL* suivi par le type d'opération y correspondant, puis le nom de ce message. Comme exemple, MeDL*ey* génère pour un message du nom *mes* déclaré dans un sous-bloc *Bcast* une fonction du nom *MDL_Bcast_mes()*.

Opérations proposées

Dans ce paragraphe, nous présentons brièvement les différentes opérations proposées pour l'extension de MeDL*ey* en communications collectives, le lecteur trouvera plus de détails sur la syntaxe et la sémantique de ces opérations dans l'annexe B.

Bcast : permet de diffuser un message d'une instance source vers l'ensemble des instances d'une tâche. Une variante de cette opération est *Mcast* qui permet de diffuser un message sur un sous-ensemble des instances d'une tâche.

Scatter : permet de fractionner des données provenant de l'instance source en n segments d'une même taille, n étant le nombre des instances au moment de l'exécution, le ième segment est envoyé à la ième instance. *ScatterV* est une variante de Scatter où les segments à distribuer peuvent avoir des tailles différentes (voir figure 6.2).

Gather : c'est l'opération inverse de *Scatter*, elle permet de rassembler des segments de même taille, provenant chacun d'une instance de la tâche, et les mettre dans le tampon de réception

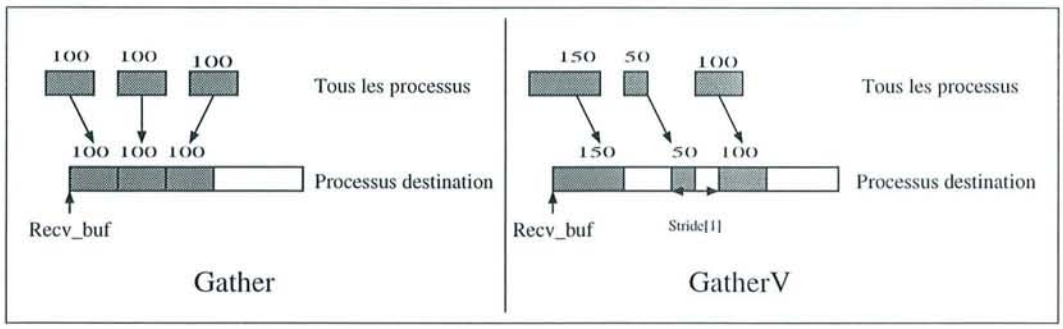

FIG. 6.3 – *Différence entre Gather et GatherV*

de l'instance destination (root). Ces segments seront concaténés en fonction des rangs de leurs instances sources. Une variante de cette opération est *GatherV* qui permet de rassembler des segments de tailles différentes et les mettre dans des déplacements différents par rapport au début du tampon de réception (voir figure 6.3).

Allgather : il s'agit d'une variante de la fonction *Gather* où toutes les instances de la tâche reçoivent le résultat du rassemblement, le segment de données envoyé par la ième instance sera reçu par toutes les instances et sera placé dans le ième bloc dans le tampon de réception. Une extension de cette opération est *Alltoall* ; toute instance envoie des données distinctes à chacune des autres instances. Le jème bloc envoyé par la ième instance sera reçu par l'instance j est sera placé dans le ième bloc.

6.3 Les opérations de synchronisation

Cette seconde catégorie regroupe les opérations qui permettent de synchroniser les processus entre eux. La barrière (barrier), déjà définie dans le langage MeDLey, constitue la fonction de synchronisation de base et permet de bloquer toutes les instances de la même tâche jusqu'à ce que chacune de ces instances ait effectué cette opération.

L'utilisation de cette fonction se fait par l'appel à *MDL_Barrier*. Nous proposons dans cette extension une nouvelle variante de cette fonction, appelée *MDL_Sbarrier*, qui permet de ne bloquer qu'un sous-ensemble des instances d'une tâche. Les rangs de ces instances doivent être fournis en argument pendant l'appel à cette fonction.

Supposons que « noeud » est le représentant d'une tâche définie avec MeDLey. Un exemple d'utilisation de cette nouvelle fonction est le suivant :

```
# include "noeudMDL.h" // Le fichier contenant la classe présentant
                       // la tâche noeud générée par MeDLey
noeud   Un_noeud;      // Une instance de cette tâche
int     *S_ens;        // contient les rangs des instances
                       // qui doivent être bloquées
.......
Un_noeud.MDL_Sbarrier(S_ens); // par exemple, si on travaille dans
  // un contexte maîtres/esclave et qu'on veut synchroniser les
  // instances esclaves, on doit spécifier leur rang dans S_ens
```

6.4 Les calculs collectifs

Cette dernière catégorie comprend les fonctions destinées à effectuer des calculs sur des données fournies par tous les processus pour obtenir un résultat unique.

Chaque processus exécute cette fonction de réduction (reduce) en fournissant une valeur. La réduction applique ensuite un opérateur sur cet ensemble de valeurs pour obtenir une seule valeur résultat qui est destinée à un ou à l'ensemble des processus sources. En général, la fonction de réduction peut traiter directement des vecteurs où la réduction se fait par éléments du même rang.

6.4.1 L'extension de MeDLey

Le langage MeDLey ne fournit aucune fonction de calcul collectif. Pour combler cette lacune, nous proposons deux nouvelles opérations *Reduce* et *Allreduce*. Comme pour les opérations collectives de transfert de données, chaque ensemble de messages de réduction utilisant l'une de ces deux opérations doit être déclaré dans un sous-bloc de *collectives*.

La syntaxe et la sémantique de ces deux opérations sont présentées dans l'annexe B. Nous nous contentons dans ce paragraphe de décrire brièvement leur principe :

Reduce : permet de calculer le résultat d'une opération ayant comme arguments des données fournies par toutes les instances d'une tâches. Ce résultat sera retourné à une seule instance destination appelée root. Cette opération permet d'utiliser plusieurs opérations parmi lesquelles nous pouvons citer maximum (MAX), minimum (MIN) ou somme (SUM).

Allreduce : il s'agit d'une extension de l'opération Reduce où le résultat de calcul est retourné à toutes les instances.

6.5 Exemple d'application : le problème de *N-corps*

6.5.1 Généralités

De nombreux problèmes de simulation numérique nécessitent le calcul de la force d'interaction entre un grand nombre de particules ou objets. Si la force entre les particules est complètement décrite par l'addition des forces entre toutes les paires des particules, et la force entre chaque paire réagit le long de la distance entre ses particules ; un tel problème est appelé le problème de *N-corps* (ou *N-body*) [Gropp 94]. Dans les paragraphes suivants, nous supposons que la force d'interaction entre deux particules n'est pas symétrique, cela veut dire que le calcul de la force d'interaction entre deux particules nécessite le calcul de cette force dans les deux sens.

Comme nous allons voir, ce problème peut bien être traité dans un environnement multiprocesseur. Cela repose sur la distribution des particules entre les différents processus. Chacun de ces processus s'occupe par la suite de calculer la force d'interaction entre les particules dont il dispose, puis entre ses particules et celles détenues par les autres processus ; ce qui permet de distribuer efficacement le calcul.

6.5.2 Implantation avec MeDLey

Pour implanter le code *N-corps*, nous avons besoin en premier lieu de savoir, si au début, les particules sont distribuées en même nombre entre les différents processus ou non. Dans la suite, nous allons proposer des solutions pour les deux cas de figure.

Le calcul de la force d'interaction entre l'ensemble des particules se fait en deux étapes. Pendant la première, chaque processus calcule la force d'interaction entre les particules dont il dispose (calcul 1). La deuxième étape consiste, pour chaque processus, à calculer la force d'interaction entre les particules dont il dispose et celles des autres processus (calcul 2). Cela nécessite, après la première étape, d'échanger les particules entre les différents processus. Après la deuxième étape de calcul, les différents processus échangent les résultats de leurs calculs afin de pouvoir calculer la force d'interaction globale. De ce fait, la résolution du problème de *N-corps* peut se faire en quatre phases :

- Phase 1 : calcul1 ;
- Phase 2 : communication (échange des particules entre les processus) ;
- Phase 3 : calcul2 ;
- Phase 4 : communication (calcul de la somme des forces d'interaction).

Rappelons que notre but en utilisant le problème de *N-corps* n'est pas de détailler les calculs de la force d'interaction entre les différentes particules, mais de spécifier les messages de communication nécessaires pendant la résolution de ce problème tout en utilisant les nouvelles fonctions de MeDLey en communication collective. Pour cette raison, nous allons nous intéresser notamment à la deuxième et à la quatrième phase.

Tout d'abord, il faut déclarer la structure d'une particule. La déclaration d'une telle structure avec MeDLey se fait avec la syntaxe du langage C. Dans la suite, nous supposons que la structure d'une particule, nommée *Particule*, est déjà définie. Chaque processus utilise un vecteur de *Particule* de taille MAX_PARTICULES, qu'il faut inclure dans le bloc *uses* de la tâche *N_corps* :

```
task N_corps(i)
uses
{
    Particule   particules[MAX_PARTICULES];
}
```

Après le premier calcul (phase 1), les différentes instances échangent les particules entre elles. Cela consiste à rassembler toutes les particules dans le tampon *particules* de l'instance 0 (root). Puis diffuser l'ensemble de ces particules sur toutes les instances. L'étape du rassemblement peut se faire de deux façons selon que les différentes instances disposent du même nombre de particules ou non. Si le nombre des particules est le même chez toutes les instances, on utilise l'opération du rassemblement (*Gather*), on commence par déclarer le message y correspondant dans le bloc *collectives* :

```
task N_corps(i)
uses
{
    ..........
    int  count; // le nombre de particules par instance
}
```

```
collectives
{
  Gather(0) // on peut écrire seulement Gather puisque 0 est la valeur par défaut
    {
        m1 = sequence {particules, particules, count};
        // particules (1) contient les données envoyées par chaque instance
        // particules (2) sera le résultat du rassemblement chez l'instance root
    }
}
```

Ce qui va nous permettre par la suite d'appeler la fonction *MDL_ Gather* :

```
# include "noeudMDL.h" // Le fichier contenant la classe présentant
                       // la tâche noeud générée par MeDLey
N_corps  Un_noeud; // une instance de la tâche N_corps
.......
Un_noeud.MDL_Gather_m1();
........
```

En revanche, si le nombre des particules n'est pas le même chez toutes les instances, on utilise la variante *GatherV* de l'opération du rassemblement. Cette opérations requiert l'utilisation de quelques paramètres que nous devons déclarer dans le bloc *uses* :

```
uses
{
   ......
   int  *counts;   // table contenant le nombre de particules par instance
   int  *displs;   // table des déplacements relatifs au début du tampon
                   // de l'instance root.
}
.......
collectives
{
   GatherV(0) // on peut écrire GatherV puisque 0 est la valeur par défaut
     {
        m2 = sequence{particules, particules, counts, displs};
        // particules (1) contient les données envoyées par chaque instance
        // particules (2) sera le résultat du rassemblement chez l'instance root
     }
}
```

L'utilisation de ce message se fait comme suit :

```
# include "noeudMDL.h" // Le fichier contenant la classe présentant
                       // la tâche noeud générée par MeDLey
N_corps  Un_noeud; // une instance de la tâche N_corps
........
int size; // nombre des instances de N_corps
size = Un_noeud.MDL_Sise();
```

```
Un_noeud.counts = (int *) malloc(size * sizeof(int));
Un_noeud.displs  = (int *)malloc(size * sizeof(int);
........
if (Un_noeud.MDL_Myrang() == 0) // L'instance root
 {
   Un_noeud.displs[0]=0;
   for (i=1; i<size;i++)
     Un_noeud.displs[i] = Un_noeud.displs[i-1] + Un_noeud.counts[i-1];
 }
Un_noeud.MDL_GatherV_m2();
........
```

L'instance du rang 0 dispose donc de toutes les particules, il reste à diffuser ces particules aux autres instances, cela peut se faire en utilisant l'opération de diffusion (Bcast). Nous commençons d'abord par spécifier les messages correspondant à cette opération dans le bloc *collectives* :

```
uses
{
    ..........
    int  total_count; // le nombre total des particules
}
......
collectives
{
  ..........
  Bcast(0)
  {
    m3 = sequence{particules, particules, total_count};
     // particules (1) : tampon d'envoi (significatif pour le root)
     // particules (2) : tampon de réception
}
```

Ce qui permet par la suite d'appeler la fonction *MDL_ Bcast* :

```
......

......
N_corps  Un_noeud; // une instance de la tâche N_corps
.....
Un_noeud.MDL_Bcast_m3();
```

Remarquons qu'une alternative consisterait à utiliser la fonction *MDL_ Allgather* au lieu de la fonction *MDL_ Gather* suivie de *MDL_ Bcast.* La fonction *MDL_ AllgatherV* sera utilisée dans le cas où le nombre de particules au départ n'est pas le même pour toutes les instances.

Après la première phase de communication, chaque instance dispose de toutes les particules. Ce qui permet par la suite de commencer la deuxième étape de calcul (phase 3). À l'issue de cette étape, on calcule la somme des forces d'interaction calculées par chacune des instances. Pour ce faire, on utilise l'opération de réduction (reduce) tout en commençant par spécifier le message correspondant :

```
uses
{
  ......
  double  force; // la force d'interaction calculée par chaque instance
  double  force_totale; // la somme des forces d'interaction
}
.......
collectives
{
  ...
  reduce(0)
   {
      m5 = sequence{force, force_totale,1,MDL_SUM};
   }
}
```

L'utilisation de ce message se fait de la façon suivante :

```
.......
N_corps  Un_noeud; // une instance de la tâche N_corps
.....
Un_noeud.MDL_Reduce_m5();
..........
```

6.6 Conclusion

Les différentes fonctions proposées dans ce chapitre permettent de couvrir les besoins pour réaliser une communication collective dans un modèle *SPMD*, où les différentes opérations s'effectuent entre les instances de la même tâche. Cependant, il reste à définir les fonctions permettant à différentes instances de plusieurs tâches de réaliser ce type de communication.

Au niveau performances, cela dépend de la plate-forme sous-jacente de communication utilisée ainsi que de l'organisation physique de la mémoire. En outre, il faut noter que la réalisation de ce type de communication est très coûteuse au niveau des machines à mémoire distribuée et par conséquent le recours à ce type de communication doit être soigneusement pris en compte dans ce cas. En revanche, la plupart des architectures à mémoire partagée sont dotées de mécanismes physiques pour réaliser ce type de communication, ce qui leur permet de réaliser de performances considérablement meilleures qu'en cas de machines à mémoire distribuée.

Chapitre 7

Extension de MeDLey pour les méthodes de décomposition de domaine

7.1 Introduction

La parallélisation de codes à l'aide de méthodes de décomposition de domaine suscite un très grand intérêt. Après un bref rappel du principe de ces méthodes, nous allons présenter l'extension [Es-sqalli 99a] de MeDLey proposée et la façon abordée ici pour la parallélisation sur des architectures parallèles à mémoire distribuée. On termine par un exemple de programme utilisant ces nouvelles fonctionnalités.

7.2 Les méthodes de décomposition de domaine

Ces méthodes se prêtent très bien à des architectures parallèles à mémoire distribuée. On décompose un domaine en plusieurs sous-domaines, autant de sous-domaines que de processus, et dans chacun de ces sous-domaines on effectue les calculs au niveau local [Brugeas 96]. Les données sur les frontières sont échangées via des communications par messages (voir figure 7.1). La taille des frontières est beaucoup plus petite que la taille du domaine global.

Une version parallèle de l'algorithme général, basée sur le principe de la décomposition de domaine, est la suivante :

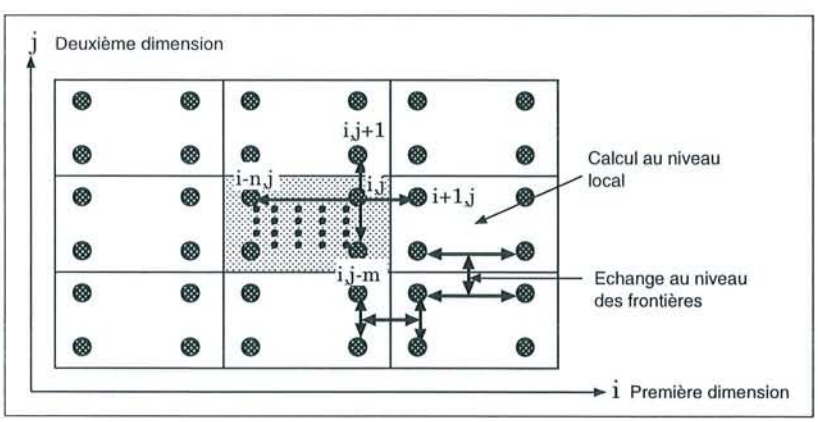

FIG. 7.1 – *Les méthodes de décomposition de domaine*

- décomposer le domaine en autant de sous-domaines qu'il y a de processus au moment de l'exécution ;
- itérer :
  - échanger les messages aux frontières ;
  - calculer.

7.3 L'extension de MeDL*ey*

L'utilisation de MeDL*ey* pour les méthodes de décomposition de domaine est très intéressante. Il existe en effet plusieurs bibliothèques de communication offrant des fonctions prédéfinies qui permettent notamment de créer une grille virtuelle de processus, de déterminer les processus voisins, etc. Notre extension est basée sur les fonctions de *MPI* [Forum 95] concernant ces grilles. La syntaxe et la sémantique de cette extension [Es-sqalli 99a] ont été définies avec le souci d'être complètes sans être complexes afin de permettre à des personnes de disciplines différentes d'utiliser cette notation unifiée.

7.3.1 Structure d'une spécification MeDL*ey*

Le modèle de programmation choisi est le modèle *SPMD* (Single Program Multiple Data). Un seul code s'exécute sur chacun des sous-domaines. Il y a autant de processus que de sous-domaines. Pour chaque processus, on a besoin de connaître ses voisins. Dans la boucle (voir l'itération ci-dessus), on échange les données avec les voisins sur les frontières puis on calcule à l'intérieur de chaque sous-domaine. D'autre part, dans la plupart des applications reposant sur le principe de la décomposition de domaine, l'échange des frontières se fait avec les sous-domaines voisins adjacents. Dans une telle situation, l'utilisateur doit au préalable déterminer les voisins de chaque sous-domaine avant de lancer le processus de communication. Avec notre extension, c'est MeDL*ey* qui prend en charge le calcul des voisins de chaque sous-domaine, il ne reste à l'utilisateur qu'à définir le contenu des messages à échanger. Les voisins adjacents de chaque sous-domaine peuvent être référencés en utilisant le mot clef *Neighbor* précédé par une combinaison des mots clefs suivants séparés par des soulignés :

- *West* ou *East* : pour déterminer les voisins adjacents selon la première dimension ;
- *North* ou *South* : pour déterminer les voisins adjacents selon la deuxième dimension ;
- *Down* ou *Up* : pour déterminer les voisins adjacents selon la troisième dimension.

Cette technique ne permet de spécifier que les voisins adjacents. Cependant, certaines applications nécessitent des échanges avec des voisins éloignés, d'où le besoin de pouvoir déterminer ces voisins. Pour ce faire, nous proposons deux possibilités :

- soit en utilisant le mot clef *Neighbor* suivi par des indices présentant le déplacement en nombre de pas dans chaque dimension (voir figure 7.2). Les voisins adjacents peuvent être définis par cette méthode. Par exemple, *West_North_Neighbor* est équivalante à *Neighbor[-1][1]* ;
- soit en utilisant la nouvelle fonction *MDL_Get_Neighbors* (voir section 7.3.2).

La structure d'une spécification MeDL*ey* proposée pour les méthodes de décomposition de domaine est la suivante :

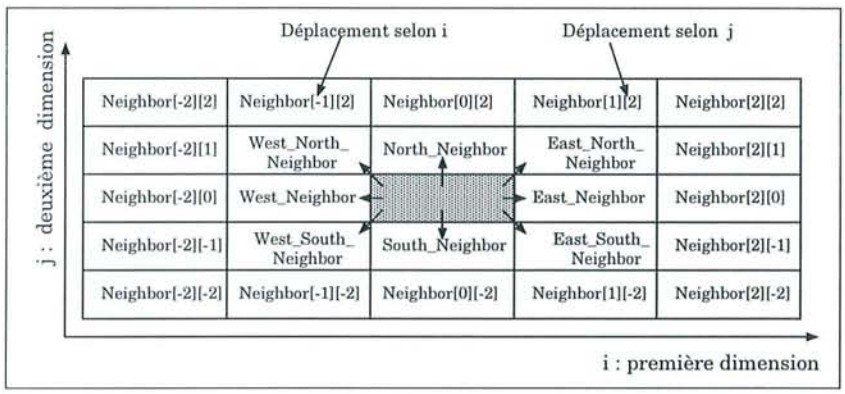

FIG. 7.2 – *Spécification des voisins dans un domaine bidimensionnel*

```
Task  A  connected  with grid(Ndim) [of  A]
  // Ndim = Nombre de dimensions, par défaut = 2
  // of A est optionnelle
uses
{
  ..... // déclaration des données à utiliser pendant
        // la communication et le calcul
}
sends
{
  ..... // déclaration des messages à envoyer en utilisant
        // les mots clefs : to, sequence, Neighbor,
        // East, West, North, South, Up, Down
}
receives
{
  ..... // déclaration des messages à recevoir en utilisant
        // les mots clefs: from, sequence, Neighbor,
        // East, West, North, South, Up, Down
}
```

7.3.2 Les fonctions proposées

Dans ce paragraphe, nous présentons brièvement les différentes fonctions proposées pour l'extension de MeDLey pour les méthodes de décomposition de domaine. Le lecteur trouvera plus de détails sur la syntaxe et la sémantique de ces fonctions dans l'annexe C.

- *MDL_Grid_Create* : permet de construire une grille de processus ainsi que de déterminer le nombre de processus dans chaque dimension (voir aussi MDL_Grid_Screate dans l'annexe C) ;
- *MDL_Grid_Rank* : renvoie le rang du processus associé aux coordonnées locales ;
- *MDL_Get_Neighbors* : permet de déterminer les voisins d'un processus ;
- *MDL_Grid_Coords* : renvoie les coordonnées du processus dont le rang est spécifié dans l'appel ;

- *MDL_ Grid_ Mycoords* : il s'agit d'une variante de *MDL_ Grid_ Mycoords* qui permet de renvoyer les coordonnées du processus qui fait l'appel.

7.4 Exemple d'application

7.4.1 Le problème de *Poisson*

Le problème de *Poisson* est une équation différentielle exprimée par les deux équations suivantes [Gropp 94] :

$$\begin{cases} \nabla^2_U & = f(x,y) \text{ à l'intérieur} \\ U(x,y) & = g(x,y) \text{ dans les bornes (marges)} \end{cases}$$

Le but de ce problème est de trouver une approximation de U étant donné les deux fonctions f et g. Pour simplifier notre discussion, nous choisissons comme domaine (U) un carré de rang N+2. Pour cette raison, on définit une grille constituée par l'ensemble des points (Xi, Yj) tel que :

$$Xi = \frac{i}{N+1} \quad pour \quad \text{i=0,1,...,N+1.} \quad Et \quad Yj = \frac{j}{N+1} \quad pour \quad \text{j=0,1,.....,N+1.}$$

On utilise U(i,j) comme abréviation de U(Xi,Xj). La valeur 1/N+1 est souvent utilisée : on la notera h. On commence par initialiser U. Ensuite, on peut donner une approximation de U dans chaque point de la grille on itérant le calcul suivant (k est le numéro de l'étape) :

$$U^{k+1}_{i,j} = \frac{1}{4} * (U^k_{i-1,j} + U^k_{i,j+1} + U^k_{i,j-1} + U^k_{i+1,j} + h * h * {f_{i,j}}^k)$$

Ce processus, nommé itération de *Jacobi*, est répété jusqu'à l'arrivée à une solution ou bien le dépassement d'un nombre maximum d'itérations sans atteindre une solution.

7.4.2 Résolution avec MeDLey

La résolution de ce problème avec MeDLey consiste à diviser le domaine (U) en autant de sous-domaines que de processus (tâches). Le travail de chaque tâche sera d'une part le calcul des valeurs de Ui,j dans le sous-domaine dans lequel effectue ses calculs, et d'autre part l'échange des frontières avec les tâches voisines.

Structure de la tâche noeud

Pour notre exemple, on prend 2 comme nombre de dimensions. La structure de la tâche noeud (présentant un sous-domaine) est alors la suivante :

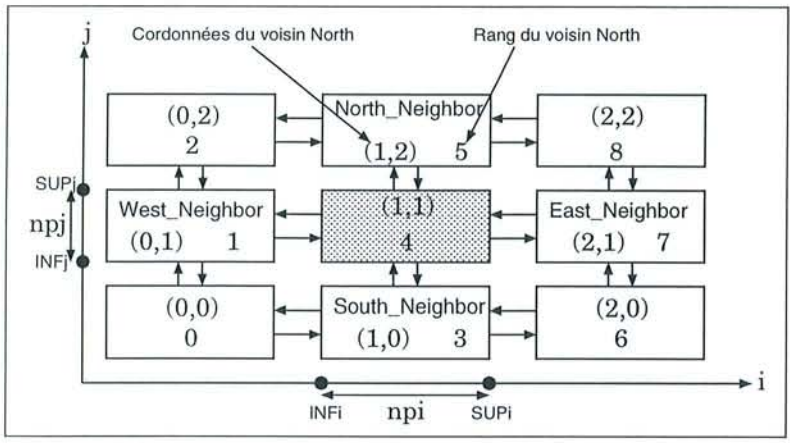

FIG. 7.3 – *Echange des frontières avec les processus voisins*

```
Task  noeud(i) connected with grid(2) [of noeud] // (2) est optionnelle
uses
{
   matrix<double>[N+2,N+2]  U;       // U est le domaine
   int INFi, SUPi, INFj,  SUPj;      // les bornes de chaque sous-domaine
}
sends
{
   m1 to North_Neighbor =  sequence{U[INFi,SUPi][SUPj]} ;
   m2 to South_Neighbor =  sequence{U[INFi,SUPi][INFj]} ;
   m3 to East_Neighbor  =  sequence{U[SUPi][INFj,SUPj]} ;
   m4 to West_Neighbor  =  sequence{U[INFi][INFj,SUPj]} ;
}
receives
{
   m5 from North_Neighbor  = sequence{U[INFi,SUPi][SUPj+1]} ;
   m6 from South_Neighbor  = sequence{U[INFi,SUPi][INFj-1]} ;
   m7 from East_Neighbor   = sequence{U[SUPi+1][INFj,SUPj]} ;
   m8 from West_Neighbor   = sequence{U[INFi-1][INFj,SUPj]} ;
}
```

Notations utilisées

Le domaine est découpé dans la direction de i et de j. On place les derniers points dans les derniers sous-domaines. Chacun des processus est repéré par ses coordonnées locales et son rang dans la grille. L'échange des frontières se fait avec les processus voisins (voir figure 7.3).

Les notations employées sont les suivantes :

- *int Ntask* : le nombre des tâches (intuitivement des instances de la tâche) ;
- *int dims[2]* : le nombre des sous-domaines dans chaque dimension ;
- *int npi,npj* : le nombre de points d'un sous-domaine selon les axes de i et de j ;
- *int Max_nbr_iteration* : le nombre maximum des itérations ;
- *double Unew[N+2][N+2]* : contient les nouvelles valeurs de U après chaque itération ;

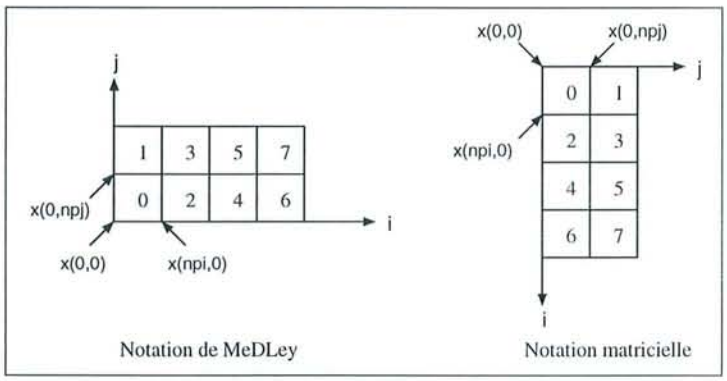

FIG. 7.4 – *Différence entre la notation matricielle et celle utilisée en MeDLey*

- *double DIF, SUM_DIF* : contiennent respectivement la différence entre U et Unew au niveau d'un sous-domaine et au niveau du domaine entier après chaque itération ;
- *double MIN_DIF* : contient la valeur de la différence minimale entre U et Unew pour laquelle il y a une convérgence.

En langage *C*, les tableaux bidimensionnels sont stockés de façon contiguë en mémoire suivant les lignes. Avec MeDLey, la même technique sera utilisée à l'exception qu'il faut faire une rotation du repère par rapport à celle utilisée en *C* (voir figure 7.4).

Programme principal

Le schéma du programme principal utilisé pour la résolution du problème de poisson en utilisant les nouvelles fonctionnalités de MeDLey de décomposition de domaine est le suivant :

```
# include "noeudMDL.h" // Le fichier contenant la classe présentant
                       // la tâche noeud générée par MeDLey

noeud   Un_noeud;      // Une instance de cette tâche

Initialisation;
Creation_grille;
Calcul_taille_Sdomaine;
Do
 {
  Un_noeud.U = Unew;
  Itération_Jacobi;
  Calcul_différence;
  Echange_ frontière;
 }
While (++nbr_iteration < Max_nbr_iteration) && (Un_noeud.SUM_DIF > Min_DIF )
      // Test de convergence selon la valeur de Un_noeud.SUM_DIF
```

Description des procédures

Nous présentons dans ce paragraphe les différentes procédures que nous avons utilisées dans le schéma du programme principal :

Procédure Initialisation : permet d'initialiser des variables utilisées tel que le domaine U.

Procédure Creation_grille : permet la création de la grille, elle se fait par l'appel à la fonction suivante : *MDL_Grid_Create(dims)* ;

Procédure Calcul_taille_Sdomaine : permet de déterminer les bornes de chaque sous-domaine. Nous présentons dans la suite la façon de calculer les bornes selon la première dimension. Le calcul des bornes selon la deuxième dimension utilise la même technique :

```
Un_noeud.MDL_Grid_Mycoords(coords);
if (East_Neighbor != MDL_PROC_NULL)
//  si je ne suis pas le premier sous-domaine selon la première dimension
{
  npi   =  N/dims[0];
  Un_noeud.INFi  = coords[0]*npi + 1;
  Un_noeud.SUPi  = Un_noeud.INFi + npi - 1;
}
else
{
   npi  = N - ( dim[0] - 1 ) * ( N/dim[0] );
   Un_noeud.INFi = N - npi + 1;
   Un_noeud.SUPi = N;
}
```

Procédure Itération_Jacobi : elle se compose de la boucle suivante :

```
for (int i = Un_noeud.INFi ; i <= Un_noeud.SUPi ; i++)
    for(int j= Un_noeud.INFj; j<= Un_noeud.SUPj ; j++)
       Un_noeud.Unew[i][j] = 0.25 * (Un_noeud.U[i-1][j] + Un_noeud.U[i][j+1] +
                   Un_noeud.U[i][j-1] + Un_noeud.U[i+1][j] - h*h*f[i][j]);
```

Procédure Calcul_différence : elle permet de calculer la somme des différences entre U est Unew de l'ensemble des sous-domaines. Suite à l'extension proposée au chapitre précèdent, on doit commencer par déclarer un message de réduction dans le bloc *collectives*. On déclare d'abord les variables utilisées pendant cette réduction :

```
uses
{
 ....
 double   DIF,  SUM_DIF;
}
```

Puis le bloc *collectives* suivant :

```
collectives
{
  Allreduce // le résultat de calcul sera retourné à toutes les instances
    {
       m9 = sequence{DIF,SUM_DIF, 1,MDL_SUM};
    }
}
```

Ce qui permet d'utiliser la fonction suivante :

```
MDL_Allreduce_m9();
```

Procédure Echange_frontière :

Pendant chaque itération, chaque processus calcul les nouvelles valeurs de U[i][j] pour $i \in [INFi,SUPi]$ et $j \in [INFj,SUPj]$. Pendant ces calculs, chaque tâche utilise des valeurs n'appartenant pas à son propre sous-domaine telles que U[INFi-1,SUPi], U[INFi,SUPi+1]. D'où la nécessité après chaque itération de communiquer ces valeurs par échange aux frontières entre les tâches voisines. On donne l'exemple de communication avec le voisin « North ». Le principe est le même pour communiquer avec les voisins « South, West et East ». Ces échanges ne diffèrent qu'au niveau du contenu des messages (les frontières) à échanger.

```
if (North_Neighbor != MDL_PROC_NULL)
// si je ne suis pas le dernier sous-domaine selon la deuxième dimension
{
  Un_noeud.MDL_SendTO_noeud_m1();
  Un_noeud.MDL_RecvFrom_noeud_m5();
}
```

7.5 Conclusion

La méthode de programmation par décomposition de domaine conduit à une parallélisation très naturelle et a l'avantage de bien se prêter à l'utilisation de la mémoire locale. L'exemple présenté dans ce chapitre est un cas à deux dimensions mais les nouvelles fonctions de MeDLey peuvent s'utiliser dans le cas d'un problème à trois dimensions

En outre, l'utilisation des voisins et des fonctions prédéfinis permet de faciliter la programmation. L'utilisateur ne doit se soucier que de définir le contenu des messages à échanger.

Quatrième partie

Utilisation de MeDLey dans un environnement CORBA

Chapitre 8

CORBA et parallélisme

8.1 Introduction

Dans un système d'objets distribués, les messages entre les objets peuvent traverser les frontières d'une application, circuler sur un réseau reliant des machines hétérogènes, et trouver sans erreur leur destinataire.

Dans ces systèmes, on distingue la couche des services de la couche de transport. C'est dans la couche des services que l'on trouve les objets propres aux applications. La couche de transport, quant à elle, est responsable de l'administration des objets et de l'acheminement des messages entre les applications.

Concernant la norme *CORBA*, les services sont décrits dans le langage *IDL* [OMG 98a], qui est indépendant des langages de programmation choisis pour implanter les comportements de ces objets. L'infrastructure de communication, ou *ORB*, est chargée de l'acheminement des requêtes entre les programmes clients et serveur. Cet isolement entre client et serveur, en termes à la fois du langage de programmation mis en œuvre, du protocole réseau, de l'infrastructure matérielle et de mécanismes de transport de données, fait la grande force de la norme *CORBA* et permet de couvrir un champ d'application plus vaste et concrétiser les bénéfices attendus de l'adoption des systèmes d'objets distribués.

Les applications parallèles font partie de celles qui peuvent en tirer profit et les recherches dans cette direction sont d'actualité. Plusieurs d'entre elles visent à évaluer les performances de certaines implantations de *CORBA* [Levine 97] [Gokhale 97] [Kuhns 99] et tester leur utilisation pour faire du calcul parallèle [Antoine 98]. D'autres développent des implantations de *CORBA* orientées vers le calcul à hautes performances et les systèmes temps réels [Schmidt 98] [Pyarali 99]. Enfin, une troisième catégorie dont le but est de fournir des environnements pour le développement des applications parallèles en utilisant *CORBA* [Priol 98] [Beaugendre 98].

Mais d'une manière générale, deux cadres d'utilisation de *CORBA* pour le développement des applications parallèles semblent intéressants :

1. **en terme de services** : le modèle d'objet client serveur que propose *CORBA* propose une coopération à grande échelle ce qui permet de faire du couplage de codes entre différents sites de calcul. En effet, les serveurs de calcul parallèle peuvent exporter leurs fonctionnalités (services) sous la forme d'objets *CORBA*. Ces services seront accessibles, partout dans le monde, par toute application adoptant cette norme ;
2. **en terme de support de communication** : le mécanisme d'invocation de méthode sur des objets que propose *CORBA* offre plusieurs fonctionnalités et peut être utilisé comme un

modèle de communication pour le développement des applications parallèles, notamment celles reposant sur le principe de tâches communicantes. En effet, chaque tâche parallèle peut être modélisée par un objet *CORBA*. Les communications entre ces tâches se font via des invocations de méthodes sur les objets présentant ces tâches. L'*ORB* sera par conséquent le support logiciel de communication à utiliser.

Nous développons ces deux aspects dans le reste de ce chapitre.

8.2 *CORBA* comme bus de services des objets parallèles

8.2.1 Le besoin d'interopérabilité pour les applications parallèles

L'enjeu de la communauté du parallélisme et de rendre les différents outils du calcul parallèle accessibles aux demandes des utilisateurs dans le cadre d'un environnement distribué à grande échelle. Pour atteindre ce but, un bon support d'interopérabilité et d'encapsulation est nécessaire. En effet, les serveurs de calcul parallèle devraient devenir accessibles à distance et intégrés dans un cadre d'application distribuée. L'utilisateur devrait par conséquent pouvoir combiner dans son code parallèle des appels à des méthodes parallèles d'algèbre linéaire, à un module de visualisation disponible, ou à un système orienté objet de base de données, avec la même facilité que dans le mode séquentiel.

Une architecture d'objets distribués telle que *CORBA* permet à des composants interopérables de coopérer. En outre, elle favorise l'utilisation d'un modèle de programmation orienté objet, avec les avantages accrus d'un développement et d'un entretien plus faciles de code, en raison de l'encapsulation, de la transmission et de la réutilisation des composants.

Les outils de programmation parallèle ne sont pas intégrés avec ce système. Une approche fondamentale pour réaliser une telle intégration est de contenir le parallélisme dans des objets. De tels objets, appelés objets *CORBA* parallèles, peuvent être mis en application en utilisant différents modèles de programmation parallèle : HPF [Koelbel 94], Message Passing, ou les langages C++ parallèles tels que pC++ [Gannon 93], CC++ [Chandy 93], Mentat [Mentat 94], ABC++ [Arjomandi 95] ou Charm++ [Kale 93]. Ou en utilisant des bibliothèques numériques parallèles développées avec des langages orientés objets. Des exemples sont fournis par la bibliothèque *A++/p++* [Parsons 94] pour des exécutions d'alignement et des calculs finis de différence, la bibliothèque de *ScaLAPACK++* [Dongarra 93] pour l'algèbre linéaire dense, et la bibliothèque d'IML [Dongarra 94] pour les méthodes itératives. L'information sur la distribution de données est encapsulée. Cependant, l'interface de ces objets, décrite en utilisant le langage *IDL*, cache la nature de l'implantation qui peut être séquentielle ou parallèle sans affecter la conception ou la mise en place des autres objets dans un système.

8.2.2 Mise en œuvre

L'enjeu actuel est d'étendre un cadre tel que *CORBA* afin de préserver l'interopérabilité avec des implémentations séquentielles et supporter efficacement les objets parallèles. En particulier, il faut éviter les goulots d'étranglement séquentiels dans les situations où un objet parallèle appelle un autre objet parallèle : contrôler l'organisation des données distribuées et contrôler les ressources de calcul des deux côtés de l'interface. Ainsi, le langage *IDL* doit être étendu pour décrire la distribution de données et supporter une variété de méthodes et différents modèles de passage de paramètres.

8.3 *CORBA* comme modèle de communication pour les applications parallèles

8.3.1 Communication par invocation de méthode

Le transfert d'information est réalisé via les paramètres de la méthode. Les informations sont encodées de façon à pouvoir être échangées entre des machines d'architectures différentes. Par le biais des paramètres de la méthode, le transfert d'information est particulièrement explicite en terme de type et de taille.

Pour répondre aux requêtes instantanées et multiples sur le même objet (ou serveur d'objets), l'invocation d'une méthode crée souvent un nouveau flot d'exécution pour réaliser le traitement qui lui est associé.

8.3.2 Performance des communications

L'architecture *CORBA*, avec toutes les opportunités qu'elle présente, parait comme un nouveau paradigme de la programmation parallèle. Cependant, *CORBA* a toujours été critiqué au niveau des performances même s'il existe des implantations de *CORBA* présentant des performances intéressantes. En tenant compte de tous ces éléments, une évaluation de performances s'avère nécessaire. Dans ce cadre, plusieurs projets de recherche s'intéressent à évaluer les performances des implantations de *CORBA* [Plasil 98c] [Plasil 98b] [Plasil 98a] [Henning 98] [Levine 99] [Shanely 98], mais le plus connu est "CORBA Comparison Project" [Group 99] qui articule ses tests autour des bus CORBA : ORBacus, Orbix et omniORB.

Nous nous sommes évidemment aussi intéressés à cet aspect de recherche [Es-sqalli 99c]. Nous avons réalisé des tests qui visent essentiellement à mesurer les performances de quelques implantations de CORBA (ORBacus [OOC 97] et TAO [Schmidt 98], de quelques implantations de la bibliothèque *MPI* (MPICH [Laboratory 99] et LAM [Team 00]) et de PVM [Team 94] sur un réseau de stations afin de faire une comparaison et situer les performances de CORBA par rapport à celles des deux bibliothèques, vues comme les standards actuels de communication par échange de messages, *MPI* et *PVM*.

Le choix des deux bus de *CORBA* ORBacus et TAO est justifié par le fait qu'ils sont parmi les implantations les plus performantes du domaine public. En outre, ils fournissent une liaison avec le langage C++ avec lequel nous avons fait nos tests.

Dans le reste de cette section, nous présentons les différents types de tests réalisés. Nous passerons en revue ensuite les résultats obtenus.

8.3.3 Types de tests développés

Test du type ping-pong classique

Le premier type de test réalisé (le plus simple et le plus connu) est un « ping-pong » classique entre une entité cliente et une autre serveur. Comme son nom l'indique et la figure 8.1 le montre, ce type de test consiste en un processus d'envoi et de réception des données entre les deux entités. Le but est de mesurer le temps de communication pour différentes tailles de données. Son principe consiste en une application cliente qui va envoyer des données à l'application serveur (exécutée sur une autre machine) qui à son tour lui renvoie les données reçues. L'application cliente va mesurer le temps séparant l'envoi et la réception des données. Pour éviter les perturbations des systèmes et du réseau, nous répétons l'échange des données plusieurs fois. Ensuite le temps d'une communication (envoi ou réception) sera déduit du temps total divisé par le double du nombre

de répétitions. Ainsi les résultats obtenus seront fonction de la taille de données échangées. Les entités clientes et serveurs seront implantées comme des objets selon l'architecture CORBA et comme des tâches pour le modèle de communication par échange de messages. Ce type de tests est à la base des autres types de tests développés.

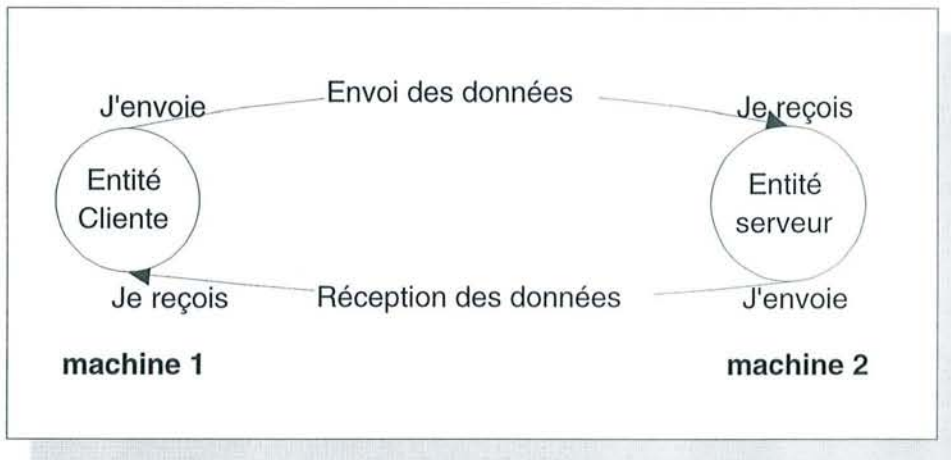

FIG. 8.1 – *Ping-pong client/serveur*

Afin de bien concrétiser l'aspect du parallélisme dans nos tests, nous avons opté pour le développement de deux autres types d'application qui se présentent comme une généralisation de celui-ci. La première application consiste en la création de plusieurs objets client interagissant en même temps avec un seul serveur tandis que la deuxième application consiste à la création de plusieurs couples client/serveur en même temps.

Application de type un seul serveur et plusieurs clients

Le premier type d'application consiste en plusieurs ping-pong entre plusieurs clients (situés sur des machines différentes) et un seul serveur. L'interaction client/serveur est matérialisée par un échange de données (envoi/réception) de tailles variables. Les résultats ainsi obtenus à l'issue de cette application seront fonction du nombre de clients et de la taille de données échangées. Ce type de test vise essentiellement à modéliser le comportement du serveur quand il a à gérer plusieurs demandes en même temps. La figure 8.2 illustre un tel type d'application.

Application de type plusieurs couples client/serveur

Le deuxième type d'application se présente aussi comme plusieurs ping-pong. Seulement ces ping-pong sont indépendants dans le sens où ils sont effectués entre plusieurs clients et plusieurs serveurs et que l'ensemble des clients est exécuté sur une machine et celui des serveurs sur une autre. Autrement dit, l'application consiste à créer plusieurs interactions entre des couples client/serveur en même temps et non entre plusieurs clients et un seul serveur comme dans la première application. Les résultats issus de ce type de test seront fonction du nombre des couples client/serveur interagissant en même temps et de la taille de données échangées. La figure 8.3 illustre un tel type de test.

Ce type de test vise à modéliser le comportement de l'ORB quand il a à gérer plusieurs

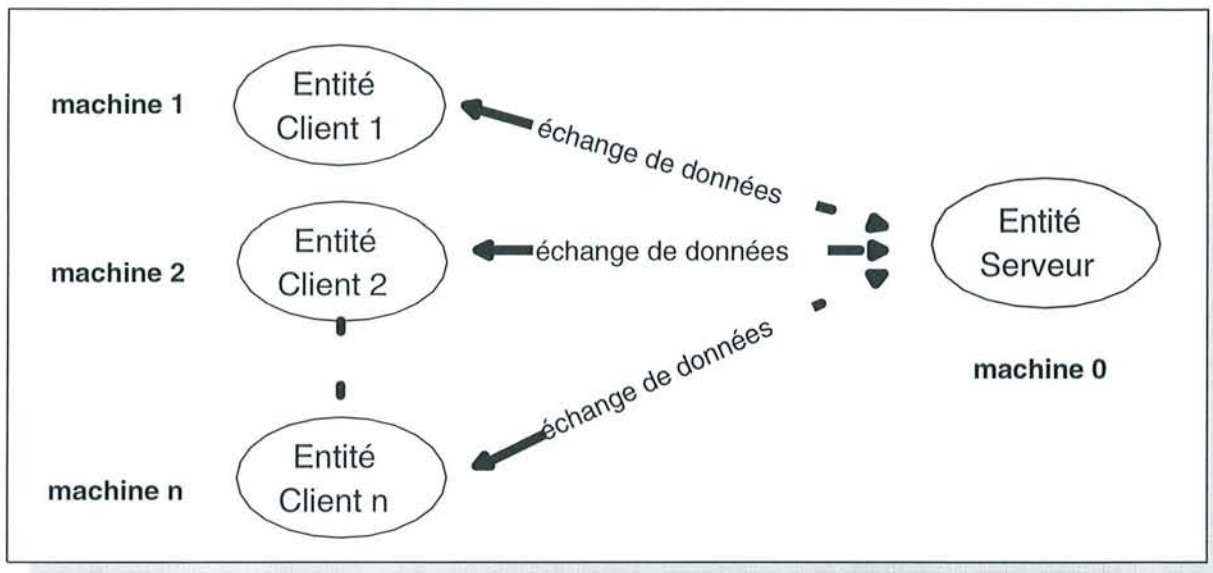

FIG. 8.2 – *Application de type un seul serveur et plusieurs clients*

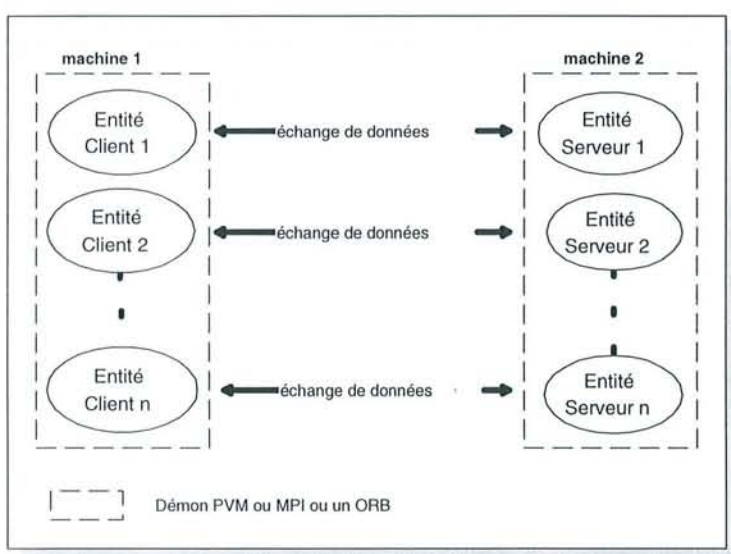

FIG. 8.3 – *Application de type plusieurs couples client/serveur*

communications entres plusieurs objets en même temps et de comparer ce comportement à celui des démons de *MPI* et de *PVM*.

8.3.4 Implantation des tests

Dans cette partie nous essayerons de présenter, d'une façon générale et générique, les différentes approches utilisées pour mettre en œuvre les tests.

Nous mettrons l'accent sur les tests de type « plusieurs clients/un seul serveur » et ceux de type « plusieurs couples client/serveur ». Les tests de type ping-pong classique sont simples à réaliser et consistent en deux entités communicantes, ces entités sont modélisées sous formes d'objets au niveau de *CORBA* et de tâches au niveau des bibliothèques d'échange de messages.

Approche en utilisant *CORBA*

La première étape à effectuer pour construire une application orientée objet distribuée selon le modèle *CORBA* consiste à déterminer les différents objets, acteurs de l'application et interagissant ensemble. Dans nos tests on a deux types d'application : l'application cliente et l'application serveur. D'après la description des tests à effectuer, la notion de l'objet client et de l'objet serveur n'est pas assez concrète dans le sens où l'objet serveur ne présente pas des services que l'objet client ne possède pas. En effet, les deux entités présentent les mêmes services d'envoi et de réception de données. Seulement, on a voulu désigner par objet serveur celui qui aura à gérer plusieurs interactions en même temps avec d'autres objets. En d'autres termes, la notion client/serveur n'est pas réalisée au niveau des objets, mais au niveau des applications. L'application cliente et serveur implantent le même type d'objet, la seule différence, c'est que l'application serveur rend publique la(es) référence(s) de(s) l'objet(s) qu'elle crée pour que les objets créés par les applications clientes puissent y accéder. En conclusion, c'est le même type d'objet qui sera implanté de part et d'autre des deux types d'application.

L'étape suivante vise à exprimer les services présentés par cet objet à l'aide du langage OMG-IDL sous forme d'un contrat-IDL entre les différentes parties. La figure 8.4 montre le contrat établi entre les différentes parties en langage OMG IDL.

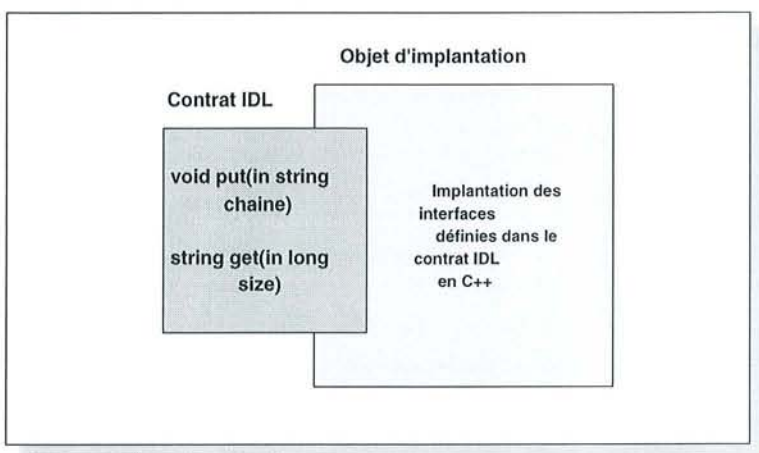

FIG. 8.4 – *Contrat IDL entre le client et le serveur*

Ce contrat contient deux méthodes (services présentés par l'objet) à savoir:

- La méthode put correspondant à l'envoi d'une chaîne de caractères passée en paramètre.
- La méthode get correspondant à la réception d'une chaîne de caractère d'une taille « size » passée comme paramètre de la méthode.

Comme on l'a déjà mentionné, l'aspect client/serveur se manifeste au niveau des applications regroupant les objets. Pour tous les types de tests, on développera deux types de programmes. Le premier est le programme (serveur) incluant l'objet jouant le rôle du "serveur", le deuxième est celui incluant l'objet jouant le rôle du "client".

Programme serveur Un programme serveur est celui regroupant l'objet ou les objets serveur(s). La seule différence entre un programme serveur pour une application de type plusieurs client/un seul serveur et le programme serveur d'une application gérant plusieurs couples client/serveur c'est que ce dernier doit créer plusieurs objets gérés par le même ORB. Le problème qui s'ajoute dans ce cas par rapport au précédent c'est la gestion des références des objets créés. En effet, dans ce deuxième type d'application on aura à gérer des couples client/serveur. Donc à un seul serveur est associé un seul client qui doit posséder la bonne référence pour qu'il puisse communiquer avec le serveur correspondant approprié. Ce programme doit donc assurer, avec le programme client, en plus de la diffusion des références des objets créés, la bonne gestion des correspondances client/serveur.

Programme client Le programme client regroupe les différents objets clients. Pour une application de type plusieurs clients/un seul serveur l'application cliente crée un objet client qui cherche la référence de l'objet serveur puis entame un ping-pong avec cet objet.

Pour le type d'application plusieurs couples client/serveur, le programme client crée plusieurs objets clients et crée autant de processus fils. Chaque processus fils se charge de gérer la communication entre un objet client avec son correspondant objet serveur. Il faut noter que pour ce type de test tous les objets clients sont gérés par un seul ORB.

Approche en utilisant les bibliothèques de communication par échange de messages

Dans un environnement de programmation par échange de messages, une application est un ensemble de tâches, coopérantes et concurrentes, exécutant leurs propres codes. L'interaction entre ces tâches est matérialisée par des communications par échange de messages. Les entités clientes et serveurs seront implantées comme des tâches par analogie aux objets dans l'architecture CORBA. Le passage de messages sera l'analogue des invocations de méthodes sur des objets à distance. Nous présentons dans ce qui suit les approches adoptées pour implanter nos tests.

Pour le type de test plusieurs clients/un seul serveur, la tâche de rang 0 pour MPI ou maître pour PVM, jouera le rôle du serveur et les autres tâches lancées joueront le rôle des clients interagissant simultanément avec la tâche serveur. Le processus d'échange de données est effectué pour des données de différentes tailles et répété un certain nombre de fois pour une même taille.

Bien que le principe du deuxième type de test (plusieurs couples client/serveur) soit le même pour MPI et PVM, la mise en œuvre diffère selon le modèle de programmation utilisé. L'idée consiste à créer deux groupes de tâches. Le premier groupe rassemble les entités clientes et le deuxième, les entités serveurs. Les deux groupes seront lancés sur deux machines différentes. Ainsi, on assistera à la création des deux groupes regroupant les tâches clientes et serveurs. Chaque tâche déterminera la tâche partenaire avec laquelle elle échangera les données.

8.3.5 Résultats et analyses

Rappelons d'abord que tous les tests ont été réalisés en utilisant des stations SUN Ultra 5 interconnectées par un réseaux Ethernet 10 Mbit/s.

Résultats du test ping-pong classique

Rappelons que ce type de test consiste en un échange de données de tailles variables entre deux entités. Les résultats seront donc fonction de la taille de données échangées. Vu que cette taille varie de 1 octet à 4MO et que les résultats obtenus présentent plusieurs phases d'inversement de performances, nous avons opté pour présenter les courbes par parties au fur et à mesure. Pour ce type de test, nous désignerons par, données de petites tailles, celles allant de 1 octet à 1 KO, données de tailles moyennes celles couvrant la plage de 1KO à 200KO et enfin données de grandes tailles celles inclues dans l'intervalle 200KO à 4MO. La courbe présentée dans la figure 8.5 montre les performances des différents composants testés pour des données de petites tailles.

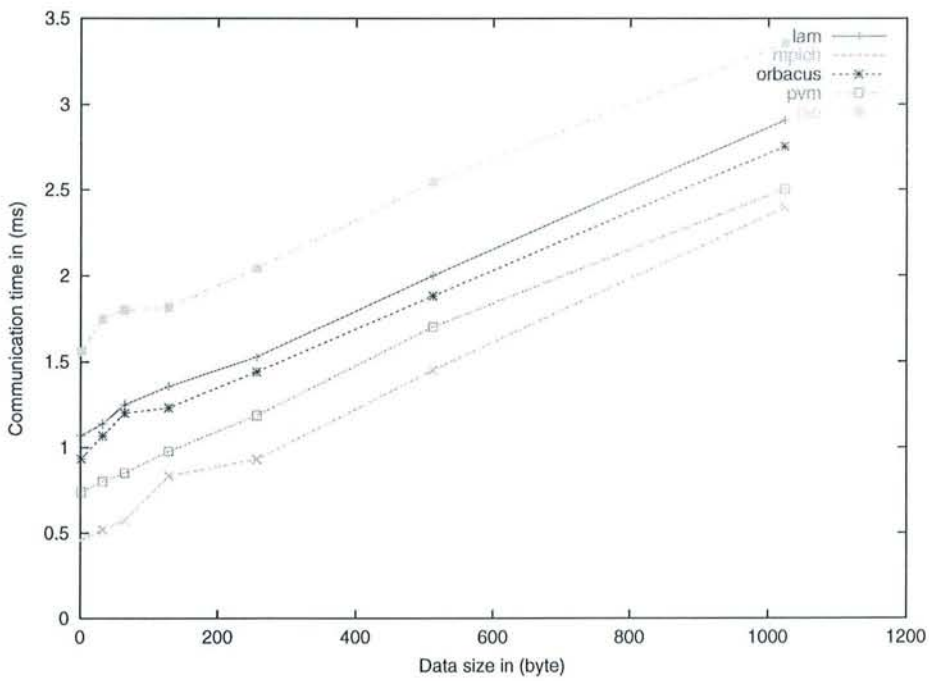

FIG. 8.5 – *Temps de communication pour une application de type ping-pong pour des données de petites tailles*

On remarque que pour ces tailles de données, MPICH suivie de PVM présentent des temps de communication assez faibles par rapport à ceux d'ORBacus qui se trouve en troisième place, puis LAM et TAO qui possèdent les plus grands délais. Cependant la plage de données entre 2KO et 16KO présentent plusieurs points seuils à partir desquels les rangs changent et les performances s'inversent. En effet, les valeurs du tableau 8.1 montre qu'à partir de 2KO, MPICH, qui était la plus performante, commence à se dégrader pour devenir la dernière à partir de 4KO et finir en dernier rang pour des données de grandes tailles.

Le même scénario est répété pour PVM qui commence à se ralentir à partir de la taille 3KO pour dépasser TAO, ORBacus et LAM et entrer en compétition avec MPICH pour éviter la dernière place.

Data size(Byte)	1024	2048	4096	8192	16384	32768	65536	131072	262144
lam	2.90	4,07	6,09	9,53	19,41	33,06	63,74	127,27	254,06
mpich	2.39	3,99	7,68	13,18	21,61	36,04	95,50	209,43	446,17
orbacus	2.75	4,06	5,82	9,86	18,56	36,49	72,40	142,34	284,45
pvm	2.5	3,79	6,20	11,54	22,56	44,1963	88,57	172,63	346,67
tao	3.35	4,64	6,36	10,05	17,54	31,98	63,24	125,63	243,99

TAB. 8.1 – *Temps de communication en (ms) d'une application ping-pong pour des données de moyennes tailles*

Contrairement au comportement de MPICH et PVM, LAM, ORBacus et surtout TAO deviennent de plus en plus performants avec l'augmentation de la taille de données. ORBacus maintient les meilleures performances avec TAO jusqu'à la taille 32KO à partir de laquelle LAM devient plus efficace par rapport à ORBacus. Quant à TAO, à partir de 10KO, il devient le plus performant et termine en premier rang avec LAM. La courbe de la figure 8.6 montre les différentes performances pour des données de grandes tailles.

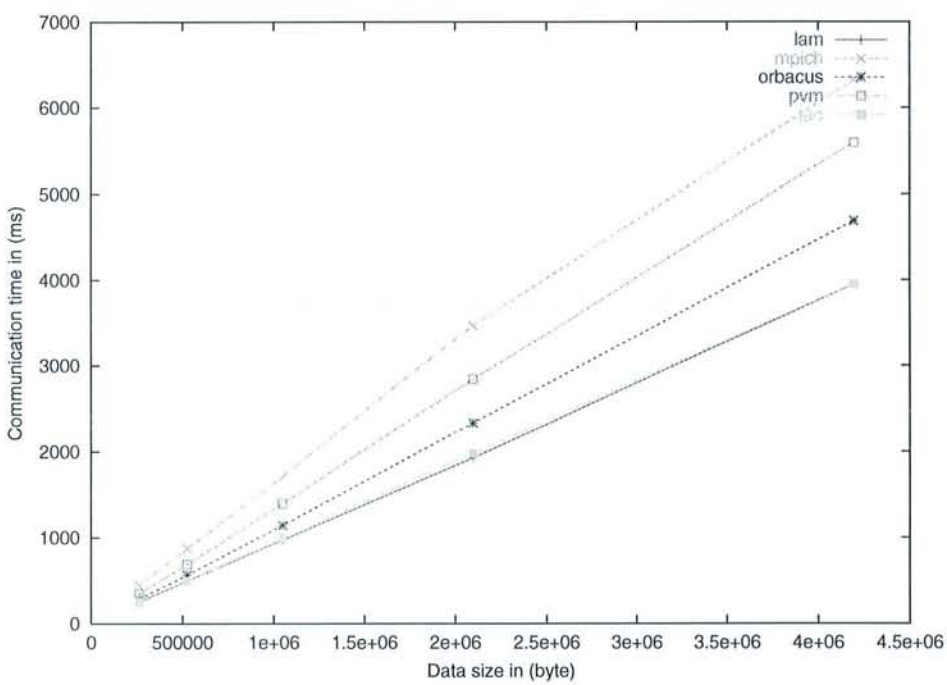

FIG. 8.6 – *Temps de communication pour une application ping-pong pour des données de grandes tailles*

En conclusion pour ce type de tests, nous pouvons dire que bien que MPICH suivi de PVM présentent les meilleures performances pour les petites tailles, elles commencent à se ralentir rapidement et perdent facilement leurs places au profit d'ORBacus qui maintient une bonne performance avec TAO jusqu'à 32KO. A partir de cette valeur LAM devient plus performant qu'ORBacus. Pour les grandes tailles, LAM et TAO présentent les meilleures performances.

Résultats des tests de type plusieurs clients/un seul serveur

Ce type de test n'est rien d'autre qu'une généralisation d'un ping-pong classique. En effet, ce dernier est un cas particulier de ce type de test correspondant à un nombre de clients égal à 1. Les courbes obtenues seront en trois dimensions puisque les résultats sont fonction de la taille de données échangées et le nombre de clients. Cependant, et pour des raisons de clarté, nous avons opté à les présenter sur deux dimensions et en fixant le nombre de clients ou la taille de données échangées.

La courbe de la figure 8.7 et le tableau 8.2 présentent les temps de communication pour une application 25 clients/un serveur pour des données allant de 1 Octet à 1 Mo.

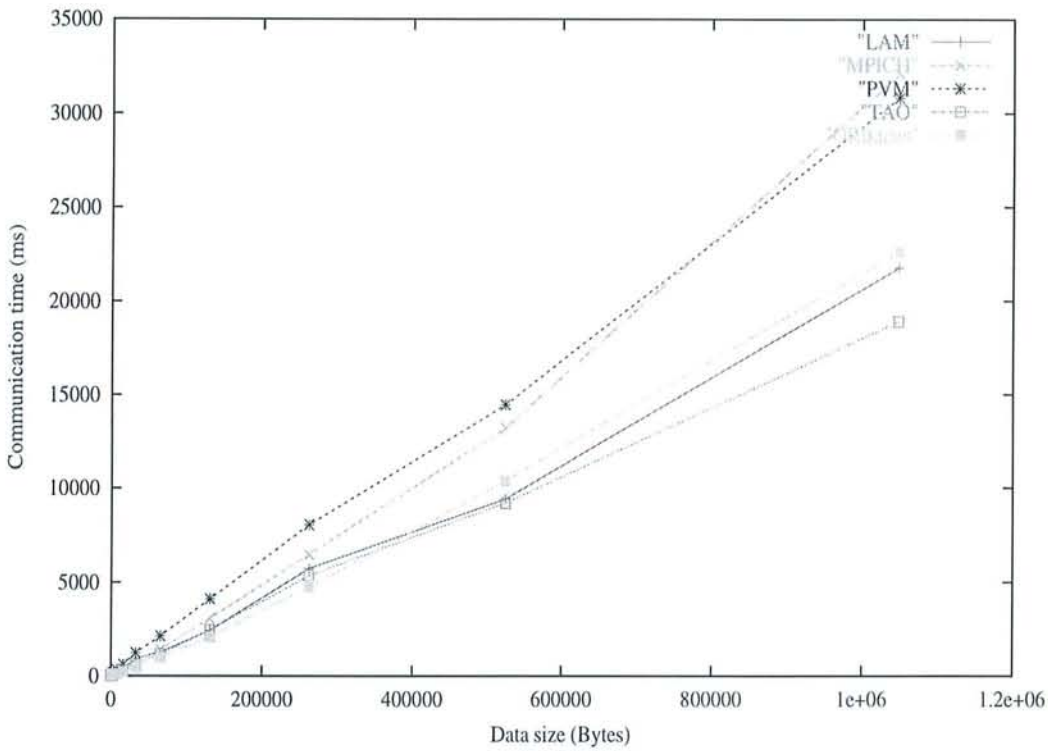

FIG. 8.7 – *Temps de communication pour une application 25 clients/un serveur*

Data size(Byte)	1	512	4096	16384	65536	262144	524288	1048576
lam	18.2	66.28	128.40	374.53	1254.86	5731.84	9404.42	21762.03
mpich	7.38	34.99	135.82	356.98	1394.32	6483.09	13198.18	32111.14
orbacus	26.14	83.72	104.26	234.69	983.09	4737.10	10384.02	22623.77
pvm	14.06	69.48	182.67	604.36	2106.6	8038.91	14454.18	30832.38
tao	17.15	30,91	91,32	331,92	1151,8	5351,04	9197,84	18893,1

TAB. 8.2 – *Temps de communication en (ms) d'une application 25 clients/un serveur*

En conclusion, puisque ce type de test est une généralisation du premier, on remarque la similarité de ces courbes avec celles présentées pour un seul ping-pong. Dans une première phase, MPICH suivie par PVM présentent des temps de communication assez faibles. Puis dans une

deuxième phase leurs performances commencent à se dégrader pour devenir les moins performantes et laisser leur place à TAO, LAM et ORBacus. Ce dernier réalise les meilleures performances pour des données de moyennes tailles (16 KO à 300 KO). Enfin, pour les données de grandes tailles, on trouve TAO nettement plus performant que le reste suivi de LAM et ORBacus qui réalisent des temps de communication similaires, puis en dernier rang MPICH et PVM.

En comparant ces courbes à celles d'un ping-pong correspondant à un seul client, on remarque la robustesse d'un serveur avec TAO par rapport à un serveur selon les autres approches. En effet, lors de l'application ping-pong classique, on a constaté que LAM et TAO deviennent de plus en plus performants en fonction de la taille des données et finissent ensemble en premier rang. En revanche, pour ce type d'application, 25 clients/un serveur, TAO devient plus performant que LAM et présente des performances nettement meilleures. En outre, ORBacus, qui était moins performant que LAM dans un ping-pong classique réalise les mêmes performances avec ce dernier pour une application de type 25 client/un serveur. Enfin, on remarque la dégradation des performances de MPICH et PVM pour ce type d'application. Toutes ces constatations montrent bien les performances d'un serveur en CORBA par rapport à celui d'une bibliothèque d'échange de messages quand on augmente le nombre de clients.

Afin de bien éclaircir cette idée et mieux voir le comportement des serveurs en fonction du nombre de clients, nous avons opté pour présenter quelques résultats d'une autre façon en fixant la taille de données et en faisant varier le nombre de clients. La courbe de la figure 8.8 montre les temps de communication pour une taille égale à 1MO et un nombre de clients variant de 1 à 25. Cette courbe illustre les remarques évoquées auparavant quand on augmente le nombre de clients : d'abord, l'efficacité de TAO par rapport aux autres, ensuite les performances similaires d'ORBacus et de LAM et enfin la dégradations des performances de MPICH et de PVM.

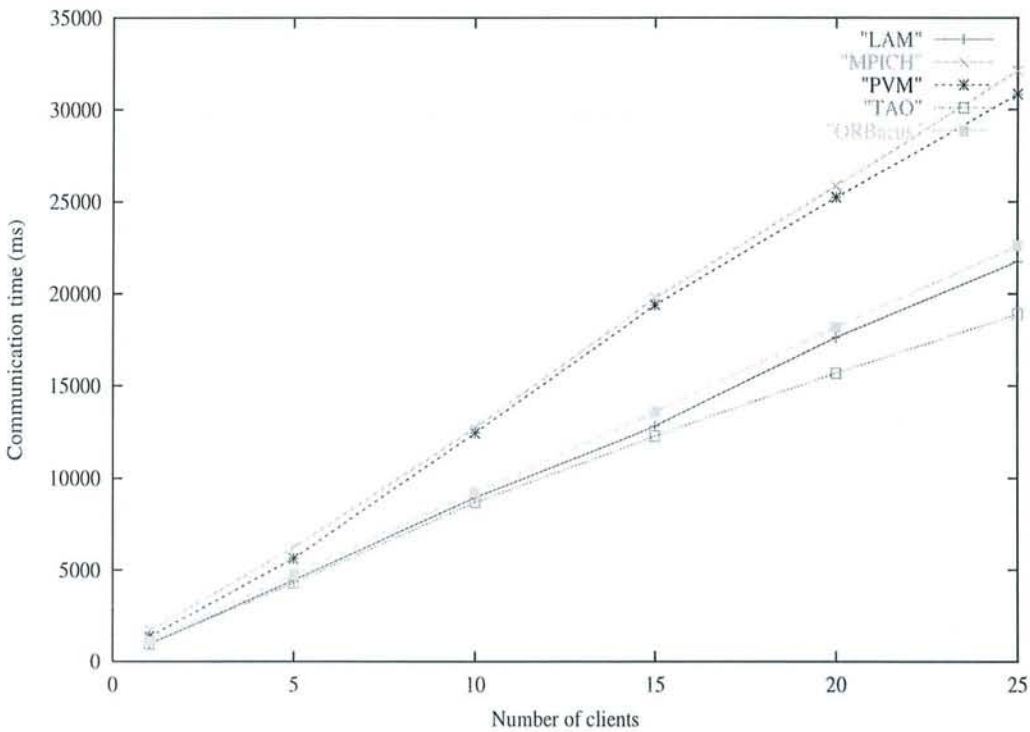

FIG. 8.8 – *Temps de communication pour une taille de données de 1MO pour une application plusieurs clients/un seul serveur en fonction du nombre de clients*

Résultats des tests de type plusieurs couples client/serveur

Ce type de test peut être aussi vu comme une généralisation du premier. Seulement pour ce type on aura plusieurs ping-pong indépendants entre plusieurs clients et plusieurs serveurs et non entre plusieurs clients et un seul serveur.

Les courbes obtenues sont en trois dimensions puisque les résultats obtenus sont fonction de la taille de données échangées et le nombre de couples communiquant simultanément. Seulement pour les mêmes raisons de clarté que précédemment nous les présentons en deux dimensions en fixant le nombre de couples ou la taille de données échangée.

Le tableau 8.3 montre les temps de communication pour une application de type 10 couples client/serveur pour des données de tailles de 1 Octet à 1 Mo. La figure 8.9 illustre ces temps de communications.

Data size(Byte)	1	512	4096	16384	65536	262144	524288	1048576
lam	6.85	10.01	42.29	150.87	589.09	2357.87	4692.04	9145.44
orbacus	5.6	8.7	36,6	147.10	578,7	2305,7	4748,5	9286,5
tao	6.2	7.8	35,8	141,3	548.8	2150,2	4244,9	8477,7
pvm	5.77	16.57	72.11	224.60	888.11	3560.72	6976.33	14016.55
mpich	3.36	13.2	38.11	126.75	642.14	3006.05	5704.10	12382.35

TAB. 8.3 – *Temps de communication en (ms) d'une application 10 couples client/serveur*

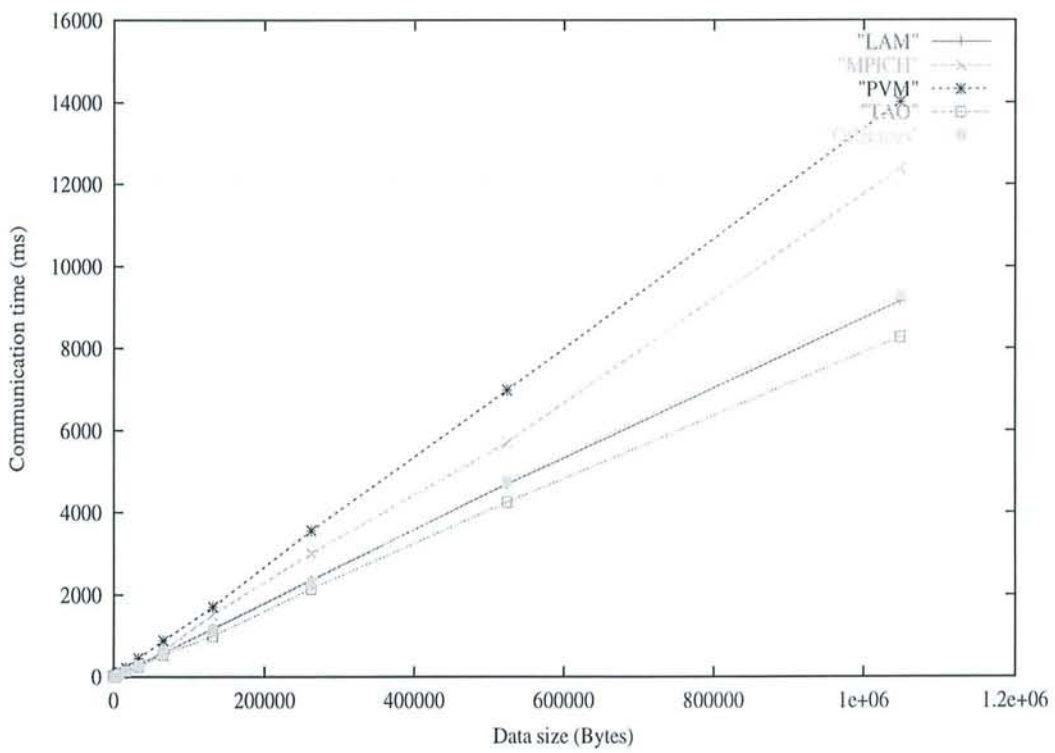

FIG. 8.9 – *Temps de communication pour une application 10 couples client/serveur*

On remarque la différence de performances entre le premier ensemble regroupant TAO, ORBacus et LAM et le second ensemble contenant PVM et MPICH. Cette différence va s'accentuer

avec l'augmentation de la taille de données.

La courbe de la figure 8.9 montre la différence nette des performances des deux ensembles. On trouve TAO nettement plus performant que les autres, suivi de LAM, ORBacus, MPICH et enfin PVM.

Ces résultats sont compatibles avec celles de types précédent dans le sens où ils préservent le même classement au sein de chaque ensemble, avec la seule différence que TAO, et pour la première fois, devient le plus performant au niveau des données de petites tailles. Cette différence de performances par rapport aux autres va s'accentuer avec l'augmentation de la taille de données échangées. D'autre part, on constate les performances, presque similaires, d'ORBacus et de LAM et enfin la dégradation nette des performances de MPICH et de LAM.

Afin de comprendre cette dégradation de performances de MPICH et de PVM, nous avons pensé que la construction de groupe pour PVM et des communicateurs et intercommunicateurs pour MPICH influençaient les temps de communication. Pour cela nous avons effectué ce même type de test sans utilisation de groupes. Ces résultats ont donné presque les mêmes performances qu'avec construction de groupes. La courbe de la figure 8.10 illustre les résultats obtenus pour MPICH avec et sans création de groupes pour 5 couples.

Vu ces résultats, on constate, que la création de groupe n'est pas le facteur responsable des mauvaises performances des bibliothèques MPICH et PVM pour ce type de tests.

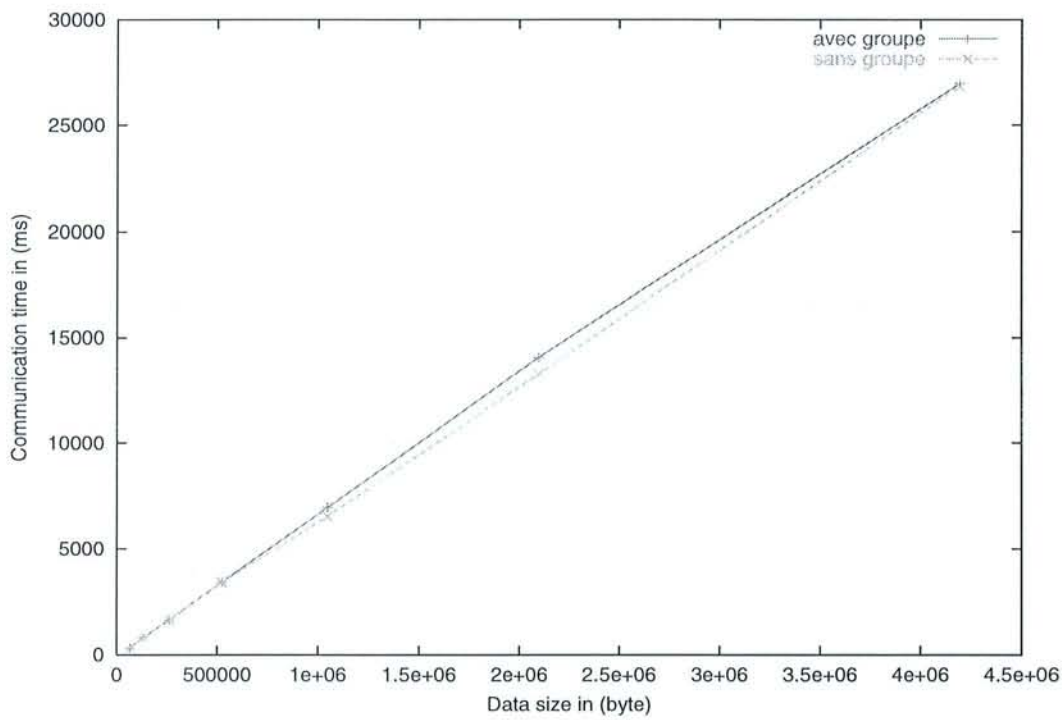

FIG. 8.10 – *Comparaison des performances de MPICH pour 5 couples avec et sans construction de groupes*

Afin de répondre à l'objectif initial de ces tests, nous présentons quelques résultats d'une façon différente, en fixant la taille de données et en faisant varier le nombre de couple de 1 à 10. Rappelons tout d'abord, que le but de ce type de test est de modéliser et comparer les comportements de l'ORB, des démons de *PVM* et de *MPI* lorsqu'ils ont à gérer plusieurs communications simultanément. C'est pour cette raison, on a opté pour présenter ces résultats d'une telle façon

pour voir le comportement de ces entités en fonction du nombre de couples. La figure 8.11 illustre le temps de communication entre différents couples pour une taille de données de 1 Mo.

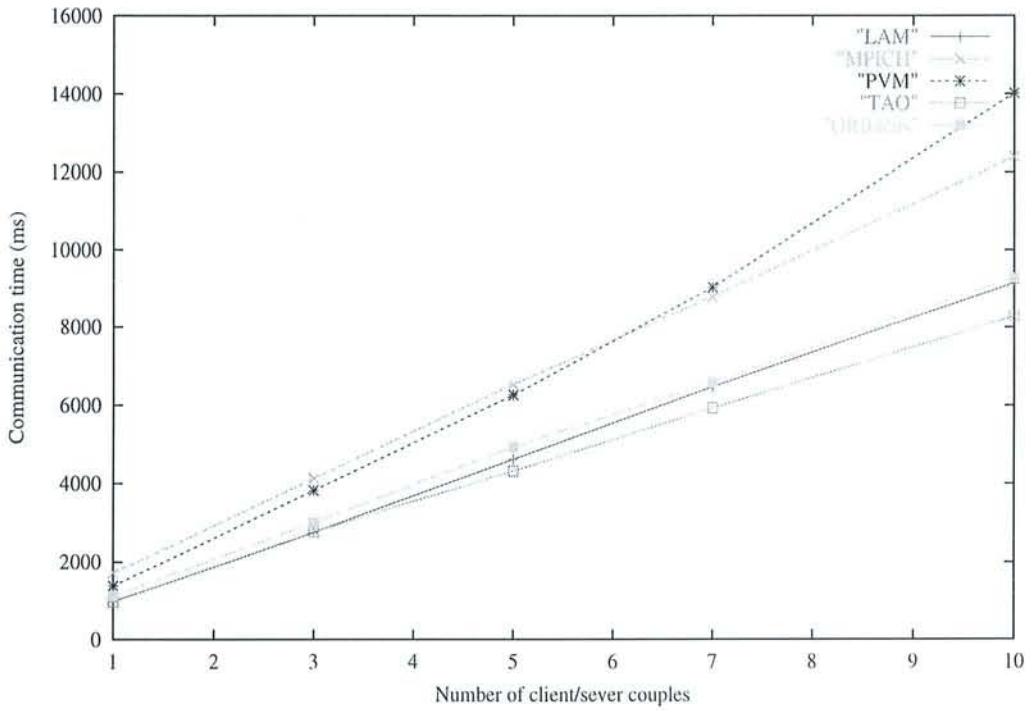

FIG. 8.11 – *Temps de communication pour une taille de données de 1 Mo en fonction du nombre de couples client/serveur*

On remarque la différence nette entre TAO, ORBacus et LAM d'une part et entre PVM et MPICH d'autre part. Cette courbe confirme les performances de la courbe précédente. On peut conclure que l'ORB, notamment celui de TAO, est plus robuste que les démons des bibliothèques de communication par échange de messages dans le sens où il peut supporter plusieurs communications simultanément d'une façon plus performante.

A noter que les performances de TAO, pendant presque tous les tests, sont obtenues grâce à son architecture. En effet, TAO est une plate-forme temps-réel conforme au modèle *CORBA* et qui permet aux développeurs d'applications distribuées de bénéficier d'une qualité de service portant sur le temps-réel. Pour ce faire, TAO utilise une plate-forme de communication appelée ACE (Adaptive Communication Environment). Cette plate-forme fournit un ensemble de composants logiciels réutilisables sur plusieurs architectures et ouverts vers plusieurs interfaces de communication : sockets, mémoire partagée, pipe, etc. TAO bénéficie de tous les avantages de ACE et par conséquent peut être utilisé efficacement sur différentes architectures.

8.4 Conclusion

Dans ce chapitre, nous avons montré l'intérêt de l'utilisation de *CORBA* pour le développement des applications parallèles. Si l'avantage de cette architecture en terme d'interopérabilité semble évident et incontestable, la question se posait sur ses performances de communication. Pour cette raison, nous avons développé un ensemble de tests de performances des deux bus *CORBA* ORBacus et TAO et nous les avons comparés à ceux des bibliothèques d'échange de

messages *MPI* (MPICH et LAM) et *PVM*.

Nous avons développé trois types de test dont le premier représente la base des deux autres. Il consiste en un processus d'envoi/réception de données de tailles variables. Les résultats, ont montré que pour des données de petites tailles, MPICH et PVM présentent des temps de communication plus faibles que ceux d'ORBacus, de LAM et de TAO. A partir d'une certaine valeur, MPICH et PVM perdent leur efficacité au profit d'ORBacus et TAO. ORBacus maintient une bonne performance par rapport à LAM jusqu'à une certaine valeur à partir de laquelle cette dernière devient plus performante. Pour les données de grandes tailles, on trouve TAO et LAM en tête, suivis d'ORBacus et en dernier rang PVM et MPICH.

Dans le second type de test, on a voulu modéliser le comportement du serveur quand il communique simultanément avec plusieurs clients. Les résultats, ont montré que TAO présente les meilleurs performances pour les grandes tailles de données, surtout quand le nombre de clients augmente. Pour les données de petites tailles, PVM et MPICH présentent les plus petits délais contrairement à TAO et LAM. Cette bonne performance de MPICH et PVM disparaît rapidement. ORBacus sera, pour une deuxième phase, le plus performant. Pour les données de grandes tailles, on remarque, surtout pour des nombres de clients élevés, la performance nette de TAO. On trouve ensuite LAM et ORBacus en deuxième position et finalement PVM et MPICH.

Le troisième type de test avait comme objectif la modélisation des comportements de l'ORB et les démons de *PVM* et de *MPI* quand ils gèrent plusieurs communications en même temps. Pour ce type de test, on distingue deux ensembles. Le premier regroupant TAO, ORBacus et LAM qui présentent les meilleurs performances avec un avantage en faveur de TAO. Le deuxième ensemble regroupant PVM et MPICH qui présentent des performances assez modestes par rapport à celles du premier ensemble.

Comme conclusion pour l'ensemble de ces tests, on remarque que CORBA présente des implantations compétitives, voir même meilleures, au point de vue performance que les bibliothèques de communication par échange de messages et que l'ORB est plus robuste que les démons de *PVM* et *MPI*. L'ORB peut gérer plusieurs communications simultanément d'une façon performante, contrairement aux démons de *PVM* et de *MPI*.

Par conséquent, l'utilisation de *CORBA* comme nouveu paradigme pour la programmation parallèle semble intéressante et peut être envisagée en ajoutant une couche d'abstraction au dessus de *CORBA* implantant divers services et primitives du traitement parallèle. Cela semble très bénéfique puisqu'on assure en plus de l'interopérabilité et la portabilité, de bonnes performances. C'est dans ce cadre que se situe notre travail sur l'utilisation de `MeDLey` dans un environnement *CORBA* que nous allons présenter dans le chapitre suivant.

Chapitre 9

Utilisation de MeDLey dans un environnement CORBA

9.1 Introduction

Comme nous avons vu dans le chapitre précédent, l'architecture *CORBA* présente plusieurs avantages d'ordre qualitatif et quantitatif. Les applications du calcul distribué et parallèle, en particulier celles reposant sur le principe des tâches communicantes peuvent tirer profit de ces avantages. En effet, chaque tâche peut être représentée par un objet *CORBA*. Les communications entre ces tâches se font via des invocations de méthodes à distance sur les objets représentant ces tâches. L'*ORB* sera par conséquent le support logiciel de communication à utiliser.

Toutefois, et vue la diversité des protocoles et les modes de communication utilisés par les applications parallèles, qui s'ajoute à la difficulté intrinsèque de l'utilisation de l'approche de conception orientée objet et les mécanismes de base des composants de *CORBA* (initialisation de l'*ORB*, de l'OA, recherche des références d'objets, etc.), cette tâche semble ardue. Elle nécessite par ailleurs une expertise, de l'utilisateur, dans ces différents domaines.

Dans ce chapitre, nous allons voir comment remédier à cette difficulté en utilisant le langage MeDLey. Nous présentons donc l'approche proposée pour spécifier les communications dans un environnement *CORBA* [Es-sqalli 99d]. Nous distinguons les différentes étapes de construction d'une applications reposant sur cette approche et nous terminons par une discussion autour d'elle.

9.2 Présentation de l'approche

Comme nous avons vu dans le cinquième chapitre, l'approche originale proposée par MeDLey est de décrire tout ce qui concerne les échanges d'une application parallèle, afin d'en dériver une implantation efficace automatique. Le but donc, est de fournir à l'utilisateur une vision plus abstraite de son application en termes de tâches et d'échanges entre ces tâches indépendamment de l'architecture matérielle et des moyens de communication utilisés.

A l'inverse de l'approche proposée pour la programmation par échange de messages qui s'appuie sur une implantation des communications en utilisant les primitives de communication des bibliothèques d'échange de messages (MPI, PVM), celle utilisée dans un environnement *CORBA* est basée sur une implantation des communications par invocation de méthodes sur les objets représentant les tâches communicantes. Son principe repose sur la modélisation de chaque tâche MeDLey par un objet *CORBA*. Les communications entre ces tâches se font via

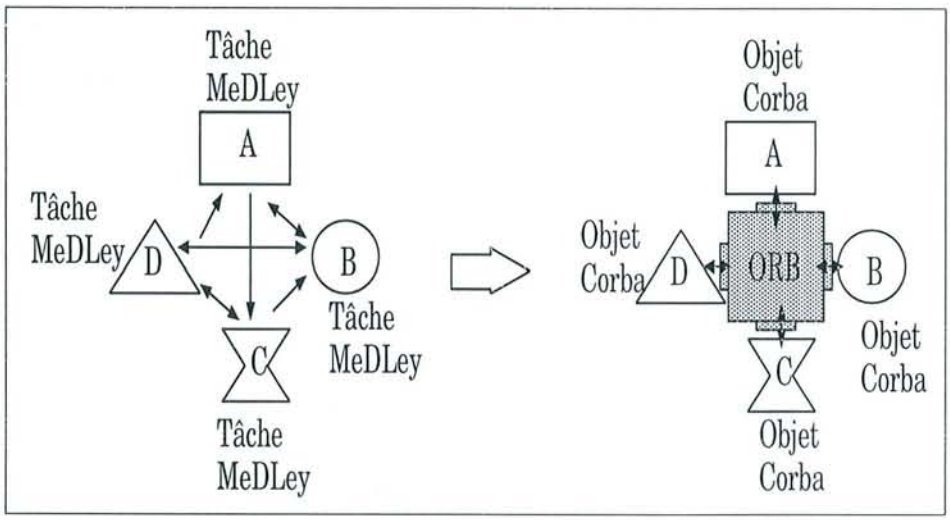

FIG. 9.1 – *Modélisation des tâches MeDLey par des objets* CORBA

des invocations de méthodes à distance sur les objets représentant ces tâches. L'*ORB* sera par conséquent le support logiciel de communication à utiliser (voir figure 9.1).

La mise en œuvre de cette approche est simple et se compose comme c'est indiqué dans la figure 9.2 de six étapes [Es-sqalli 99c] :

1. *Structuration de l'application en tâches communicantes ;*
2. *description des échanges entre ces tâches dans un fichier de spécification MeDLey ;*
3. **projection de la spécification MeDLey en contrat *IDL* ;**
4. **projection du contrat *IDL* en souches et squelettes ;**
5. **implantation des échanges entre tâches et des méthodes d'initialisation et de destruction des objets représentant ces tâches ;**
6. *écriture des programmes correspondant aux différentes tâches composant l'application en utilisant les fonctions de communications générées.*

L'utilisateur d'une telle approche ne doit se soucier que des étapes 1, 2 et 6. En revanche, l'outil de génération automatique prendra en charge les étapes 3, 5 et fait appel au compilateur du langage *IDL* pour réaliser la quatrième étape. Dans les paragraphes suivants, nous allons détailler chacune des cinq premières étapes.

9.2.1 Structuration de l'application

Dès que le choix du programmeur s'est porté sur l'utilisation du principe de tâches communicantes, qui caractérise le parallélisme de contrôle, se pose le problème de la découpe de l'application en tâches parallèles. Actuellement, cette étape de décomposition reste manuelle et s'appuie sur une analyse et une transformation des schémas des échanges en graphe de communications. A partir de ce graphe, on peut donc extraire facilement les entités (i e. tâches communicantes) ainsi que les échanges entre ces tâches.

9.2.2 Description des échanges avec MeDLey

Une fois la structuration de l'application en tâches parallèles faite, l'utilisateur peut alors décrire le fichier de spécification MeDLey (voir chapitre 5). Une telle description doit contenir

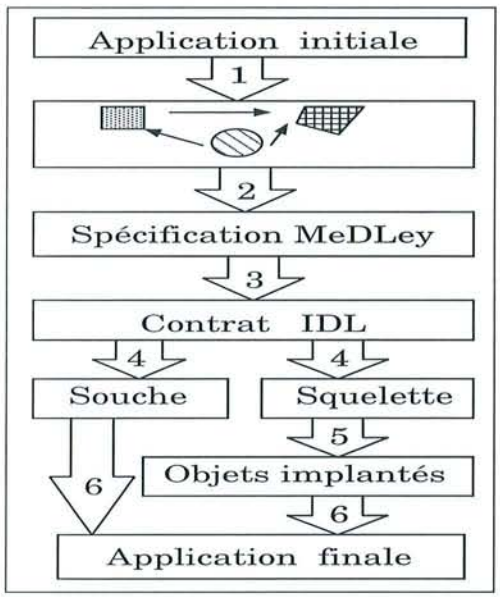

FIG. 9.2 – *Les étapes de construction d'une application* MeDLey *dans* CORBA

la spécification des échanges entre les représentants des tâches en terme de messages et de leurs contenus. Chaque représentant de tâche MeDLey est définie par son nom, et contient les parties suivantes :

- un préambule, permettant de définir le représentant lui-même ainsi que les interactions avec les autres représentants de l'application ;
- le bloc *uses*, qui permet de définir les données locales à chaque instance de tâche ;
- le bloc *sends*, qui permet de définir les envois de données vers d'autres instances de tâches ;
- le bloc *receives*, qui permet de définir les réceptions de données provenant d'autres instances de tâches ;
- le bloc *collectives*, permet de définir les opérations de communications collectives.

9.2.3 La projection MeDLey vers l'*IDL*

Une fois que la description MeDLey est faite, le compilateur du langage, après avoir rempli la table de symboles, génère le contrat *IDL* correspondant. Une telle génération repose en première étape sur la projection des types utilisés dans la description MeDLey vers leur type correspondant en langage *IDL* ; puis dans une deuxième étape, la génération des signatures de méthodes de communication correspondant aux messages déclarés dans la spécification MeDLey.

Les règles de projection MeDLey vers l'*IDL*

Les règles de projection d'une description MeDLey vers un contrat *IDL* sont les suivantes :

- Une description MeDLey est projetée vers un module en *IDL* du nom de fichier contenant cette description ;
- une tâche est traduite vers une interface *IDL* ;
- les types simples ne subissent pas un grand changement du fait que les deux langages reposent sur la syntaxe du langage C, et sont traduits dans les types natifs du langage

MeDLey	*CORBA*/IDL	Valeurs
Int	Long Entiers	32 bits signés
Float	Float	Nombre flottant 32 bits
Double	Double	Nombre flottant 64 bits

TAB. 9.1 – *la projection des types de données élémentaires de MeDLey vers l'*IDL

IDL. Le tableau 9.1 présente la projection des types de données élémentaires du langage MeDLey vers le langage *IDL* ;

- le type Vector est transformé en type anonyme d'un tableau d'une dimension et de la même taille ;
- le type Matrix devient un type anonyme d'un tableau de la même dimension et de la même taille ;
- les structures sont simplement traduites en structures de données en *IDL* ;
- les identificateurs de variables sont traduits en attributs en langages *IDL* ;
- les messages sont projetés en méthodes en *IDL*.

9.2.4 Projection du contrat *IDL* en souches et squelettes

Une fois la spécification décrite en *IDL*, il est nécessaire de la projeter vers un langage de programmation pour pouvoir implanter les objets, représentant les instances des tâches communicantes, et réaliser les application utilisant ces objets.

A partir d'une description *IDL*, deux types de projection vers un langage de programmation sont réalisés : la projection pour implanter les objets, appelée squelette *IDL*, et celle pour les applications utilisant les objets, appelée souche *IDL*. Rappelons que les souches assurent la liaison entre l'application cliente et l'*ORB*, tandis que les squelettes implémentent la liaison entre l'*ORB* et le serveur pour l'objet considéré. Ces deux éléments permettent de séparer les applications du substrat de communication, c'est à dire le bus *CORBA*.

Cette étape de projection est automatisée par les compilateurs *IDL*. Dans notre approche, après avoir projeté une description MeDLey vers un contrat *IDL*, l'outil de génération automatique de MeDLey fait appel au compilateur *IDL* pour générer les souches et les squelettes du contrat *IDL*.

9.2.5 Implantation des échanges

L'implémentation d'un objet fournit la sémantique de l'objet, en définissant les données pour l'instance d'un objet et le code pour les méthodes de l'objet.

Rappelons que dans notre approche, chaque tâche MeDLey sera projetée en une interface *IDL*, qui à son tour sera traduite en une classe d'objet par le compilateur *IDL*. Reste donc à implanter les méthodes de cette classe qui ne sont que les échanges à réaliser par cette tâche.

Une telle implantation nécessite d'abord la définition d'un protocole de communication qui correspond aux différents modes d'envoi/réception utilisés dans MeDLey.

Dans ce contexte, nous avons développé un premier protocole pour les communications point à point mais pendant sa mise en pratique nous avons du constater que notre approche était lourde à utiliser. En effet, lors de chaque changement au niveau d'une spécification de communication avec MeDLey il faut tout recommencer : générer la nouvelle interface *IDL*, générer ensuite les souches et les squelettes, les compiler, etc. En outre, *CORBA* n'était pas complètement caché

à l'utilisateur du fait qu'il devait lui même prendre en charge la compilation des souches et squelettes.

Nous avons conclu que l'erreur commise était de générer une interface *IDL* à partir d'une spécification MeDLey sachant que seul le protocole de communication dépendait de *CORBA*, et qu'il fallait par conséquent séparer la spécification de communication de MeDLey de l'interface *IDL*; C'est à dire développer un protocole de communication avec *CORBA* puis l'utiliser directement par MeDLey.

Pour cette raison, nous avons développé une bibliothèque de communication par échange de messages au-dessus de *CORBA* et qui implante notre protocole. Cette bibliothèque sera utilisée par MeDLey de la même façon que dans le cas de l'implantation de MeDLey avec MPI. Cela veut dire qu'à partir d'une spécification de communication, MeDLey génère directement des primitives de communication à partir de celles qui existent dans cette bibliothèque sans passer par une projecton de cette spécification en langage *IDL*. Cela assure parfaitement l'indépendance entre MeDLey et le langage *IDL*.

9.3 Conclusion

Dans ce chapitre, nous avons présenté une première approche pour utiliser le langage MeDLey dans un environnement *CORBA*. Si cette approche reste faisable, nous avons pu remarquer lors de sa mise en pratique qu'elle était lourde et difficile à utiliser. Pour remédier à ce problème, nous avons alors développé une bibliothèque de communication par échange de messages au dessus de *CORBA*. La description de cette bibliothèque fait l'objet du prochain chapitre.

Chapitre 10

MPC : Message Passing in *CORBA* environment

10.1 Introduction

Dans ce chapitre, nous allons présenter la bibliothèque de communication par échange de messages dans un environnement *CORBA* appelée *MPC* (Message Passing in *CORBA* environment) [Es-sqalli 00c].

Comme nous l'avons signalé au chapitre précèdent, la motivation initiale qui nous a conduit à développer cette bibliothèque est l'implantation du langage `MeDLey` dans un environnement *CORBA*. Cependant, nous avons également développé une interface de communication à cette bibliothèque pour qu'elle puisse être utilisée, comme toute autre bibliothèque de communication par échange de messages, indépendamment du langage `MeDLey`. Cette bibliothèque propose donc un environnement complet pour la programmation par échange de messages dans un environnement *CORBA*.

10.2 *MPC*: concepts de base

MPC est une bibliothèque d'échange de messages en C++. Cette bibliothèque repose sur l'utilisation du mécanisme d'invocation de méthode sur des objets distants du système *CORBA*. Les buts que nous avions fixés à l'avance sont les suivants :

- une interface de communication habituelle : l'interface de *MPC* doit être semblable, au maximum, à celles des bibliothèques d'échange de messages existantes, notamment *MPI* et *PVM* ;
- diverses primitives de communication : la bibliothèque *MPC* doit offrir une variété de primitives de communications utilisée au niveau du calcul parallèle. Ces primitives doivent couvrir en particulier les communications point à point et celles collectives. En outre, elles doivent offrir divers modes de communication ;
- performance des communications : il est évident que ces performances seront liées à l'implantation de CORBA utilisée. Cependant, cette bibliothèque doit utiliser de la façon la plus efficace les mécanismes développés par *CORBA*. En outre, le surcoût de cette bibliothèque doit être faible ;
- masquer *CORBA*: aucune connaissance du système *CORBA* n'est imposée aux utilisateurs ;

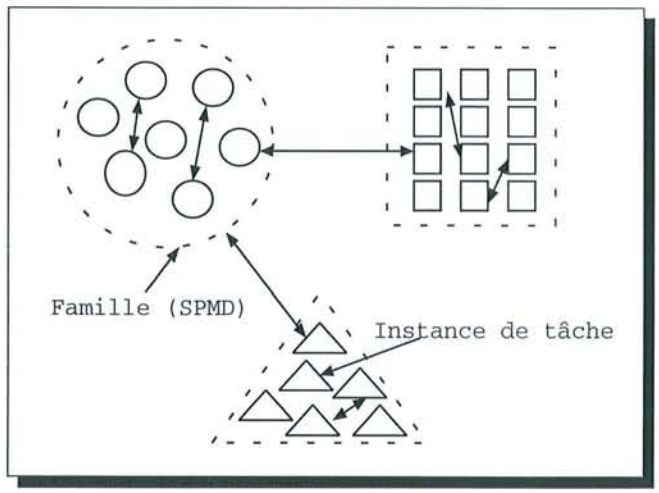

FIG. 10.1 – *La structuration de l'application avec* MPC

- un bon support d'interopérabilité et de couplage de codes: *MPC* doit permettre à des composants interopérables de coopérer. En outre, les primitives de *MPC* doivent cohabiter avec celles des autres bibliothèques de communication. Cela permet par conséquent à des programmes de diverses implantations de différentes bibliothèques d'échange de messages de communiquer via les primitives de *MPC*.

Par ailleurs le choix du langage C++ est justifié par le confort de la programmation du à l'adoption de l'approche orientée objet avec tous les mécanismes qu'elle offre : héritage, généricité, etc. En plus, la plupart des bibliothèques de communication existantes offrent une interface en C++ ou au moins en C. Enfin, la plupart des compilateurs *IDL* fournis avec les implantations de *CORBA* qui existent à l'heure actuelle génèrent des interfaces en C++ ou en Java, ces interfaces seront à la base de la bibliothèque *MPC*. Cependant, proposer une interface de *MPC* en langage Fortran reste un de nos objectifs vu le poids de ce langage dans le calcul haute performance.

Avant d'entrer en détail dans la façon dont la bibliothèque *MPC* est implantée, nous présentons d'abord la structuration d'une application utilisant cette bibliothèque puis l'interface de communication proposée.

10.3 La structuration de l'application avec *MPC*

MPC propose un modèle de programmation en tâches concurrentes de type M-SPMD (Multiple - Single Program Multiple Data) (voir chapitre 5 section 2). Dans ce modèle, différents types de tâches peuvent s'exécuter simultanément, avec un nombre de tâches variable par type (voir figure 10.1).

Le modèle M-SPMD permet donc de décrire la structure d'une application allant du SPMD (une seule famille) au MPMD (plusieurs familles réduites à un membre unique). Le nombre de tâches (instances) au sein d'une famille n'est pas obligatoirement une constante de l'application ; ces nombres peuvent être donnés comme paramètres initiaux lors du lancement de l'application. Notons que l'opération de structuration de l'application en tâches de types différents, ainsi que leur nombre, restent à la charge du programmeur : *MPC* ne propose aucun formalisme pour aider l'utilisateur dans cette étape du développement.

Nous présenterons donc *MPC* selon deux points de vue complémentaires :

- le modèle de tâches introduit ;
- l'interface de communication proposée.

10.3.1 Le modèle de tâches

La programmation avec *MPC* est basée sur l'utilisation des objets *CORBA*. Le premier objet à considérer est celui qui représente les communications relatives à une instance de tâche, on l'appelle dans l'implantation de *MPC* l'objet *MPCTaskInstance*. Toutes les primitives de communication portent implicitement sur cet objet. Présentons d'abord un exemple de squelette d'une tâche *MPC*:

```
#include ''mpc.h''

main(int argc, char ** argv)
{
  ...
  MPC_Init(argc, argv); // phase d'initialisation

  ...
  // Calcul et communications
  ...

  MPC_End(); // phase de terminaison
}
```

Le squelette présenté ci-dessus respecte les trois phases successives classiques de la programmation SPMD. On peut y distinguer trois phases :

1. une phase d'initialisation, *MPC_Init(..)* avec comme paramètres ceux de la ligne de commande ; cette fonction permet d'initialiser l'environnement *MPC* ;
2. une phase de calcul et de communication ;
3. une phase de terminaison, *MPC_End()*.

En outre, et comme nous allons voir dans le reste de ce chapitre, toute primitive de la bibliothèque *MPC* commence par le préfixe "MPC_" suivi du nom de cette primitive et ses paramètres.

Identification des tâches

Pour envoyer ou recevoir un message, il faut disposer d'un moyen pour préciser où va le message et éventuellement d'où il vient. Plusieurs approches sont possibles pour identifier une tâche. Le choix de ces identificateurs pourrait naturellement être guidé par la façon dont le programmeur utilise les tâches. On peut dégager essentiellement deux méthodes :

1. la décomposition fonctionnelle, où le problème est divisé en différentes unités fonctionnelles concurrentes. Cette décomposition correspond au parallélisme de contrôle ;
2. la décomposition par les données, ou toutes les tâches exécutent les mêmes actions sur des données différentes. Ce qui correspond à un parallélisme de données.

Le premier cas s'adapte très bien à un nommage (ou identification) symbolique alors que le second s'adapte plutôt à un nommage numérique.

Le nommage symbolique L'utilisation d'identificateurs de tâches symboliques est naturelle et très pratique pour le programmeur. En effet, s'il utilise une tâche pour l'échange graphique des résultats d'un calcul, il pourra nommer sa tâche Echange_graphique, plutôt que de se souvenir qu'il s'agit de la tâche 7 parmi les 20 tâches de son application. En fait, les noms symboliques permettent de faire une distinction fonctionnelle des tâches.

Le nommage numérique Le nommage numérique des tâches permet par contre de calculer les destinataires d'un message à l'aide d'une expression arithmétique. Par exemple, prenons le cas où des tâches sont organisées suivant une forme d'anneau. Si l'on veut faire tourner j une valeur dans cet anneau, il suffira d'envoyer cette valeur à la tâche i + 1 modulo la taille de l'anneau. Cette opération serait relevée très lourde dans le cas d'un nommage symbolique.

Dans le cas de la bibliothèque *MPC*, il est évident qu'une combinaison de ces deux approches permet facilement d'identifier les tâches. On appelle ce type de nommage hybride. Dans ce cas, un nommage symbolique est utilisé pour distinguer les groupes ou familles de tâches ; et un nommage numérique est utilisé pour identifier une instance d'une tâche ou sein d'une famille. En conséquence, un identificateur de tâche est considéré comme un couple : (symbolique, numérique).

Le lancement des tâches

Le lancement d'une application *MPC* se fait en utilisant la commande *mpcrun*. Cette commande utilise un fichier de configuration appelé *mpchosts* qui contient la liste des machines à utiliser pendant l'exécution.

Par exemple, pour lancer une application composée de 3 instances d'une tâche A et 5 autres d'une tâche B, il faut utiliser la ligne de commande suivante :

```
mpcrun -t 3 A -t 5 B
```

10.3.2 L'interface de communication

Les communications point à point

MPC offre 4 primitives de communication point à point :

- *MPC_Asend* : c'est la primitive d'envoi asynchrone ; l'appel à cette primitive est non bloquant ;
- *MPC_Ssend* : c'est la primitive d'envoi synchrone ; l'appel à cette fonction bloque l'émetteur jusqu'à ce que la primitive de réception correspondante soit appelée par le récepteur ;
- *MPC_Arecv* : c'est la fonction de réception asynchrone ; comme dans le cas de l'envoi asynchrone, cette primitive est non bloquante. La réception effective se fait après l'envoi du message correspondant par l'émetteur ;
- *MPC_Srecv* : C'est la fonction de réception synchrone. Cette primitive bloque le récepteur jusqu'à l'arrivée du message attendu.

Nous présentons ci-dessous un exemple simple de leur utilisation :

Tâche 'A' :

```
#include ''mpc.h''
main(int argc, char **argv)
{
```

```
    double  x[3];
    int     c = 15 ;
    int  myrank ;
    MPC_Init(argc, argv); // phase d'initialisation
    myrank = MPC_Myrank() ;

    if (myrank == 0)
        MPC_Asend('B', 1, 10, c, 1, MPC_INT) ;
    else if (myrank == 1)
        MPC_Srecv('B', 0, 100, x, 3,  MPC_DOUBLE) ;

    MPC_End();
}
```

Tâche 'B' :

```
#include ''mpc.h''
main(int argc, char **argv)
{
    double  y[3]={3.2, 2.1, 0.23};
    int     d ;
    int  myrank ;
    MPC_Init(argc, argv); // phase d'initialisation
    myrank = MPC_Myrank( ) ; // retourne le rang de la tâche courante

    if (myrank == 0)
        MPC_Ssend('A', 1, 100, y, 3, MPC_DOUBLE) ;
    else if (myrank == 1)
        MPC_Arecv('A', 0, 10, &d, 1,  MPC_INT) ;

    MPC_End();
}
```

Dans cet exemple, l'instance 0 (MPC_Myrank()) de la tâche 'A' envoi un message asynchrone à l'instance 1 de la tâche 'B'. Ce message a comme tag (ou étiquette) la valeur 10 et comme contenu un entier dont la valeur se trouve dans la variable *c*. De son coté, l'instance 1 de la tâche 'A' reçoit un message synchrone de l'instance 0 de la tâche B. Ce message a comme étiquette la valeur 100 ; les données à recevoir sont 3 nombres de type double qui seront mis dans le tableau x.

Les communications collectives

Contrairement aux opérations point à point qui permettent à deux tâches d'échanger des données, les opérations collectives mettent en œuvre une communication entre un groupe de tâches. Dans la bibliothèque mpc, la notion du groupe est implicite, chaque famille (i.e. SPMD) est considérée comme un groupe. Par conséquent, chaque instance d'une tâche fait implicitement partie du groupe de sa famille lors du démarrage de l'application. Pour se désabonner de ce groupe, *MPC* offre la fonction *MPC_Lfamily()* ; et pour se réabonner la fonction *MPC_Jfamily()*.

MPC fournit quatre classes d'opérations collectives :

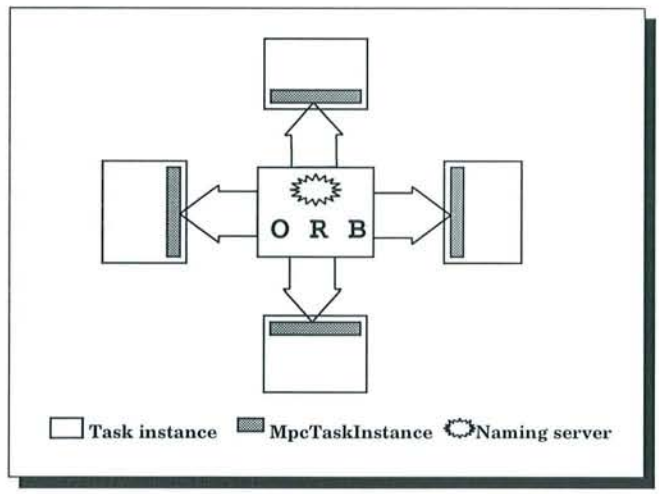

FIG. 10.2 – *L'environnement* MPC

- *MPC_ Lbarrier* et *MPC_ Gbarrier*: la première fonction permet une synchronisation locale des membres d'un groupe (ou une famille). En revanche, la deuxième fonction permet une synchronisation globale de l'ensemble des tâches d'une application.
- *MPC_ Lbcast* et *MPC_ Gbcast* : permettent respectivement la diffusion locale et globale de données (voir section 3.6) ;
- *MPC_ Gather*: pour la rassemblement de données ;
- *MPC_ Scatter*: permet la distribution de données ;

10.4 Implantation de la bibliothèque *MPC*

10.4.1 Les communications point à point

Comme nous l'avons signalé dans la section 10.3.1, la programmation avec *MPC* est basée sur l'utilisation des objets *CORBA*. Le premier objet à considérer s'appelle $MPCTaskInstance$ et représente l'entité, ou l'interface de communication de chaque instance d'une tâche (voir figure 10.2). Toutes les primitives de communication portent implicitement sur cet objet. Une primitive de communication est traduite sous forme d'une ou de plusieurs invocations de méthodes sur les objets participant à cette communication. La communication entre ces objets se fait via l'ORB.

En outre, un serveur de nommage est exécuté au début de chaque application *MPC*. Pendant la phase d'initialisation (*MPC_ Init*), chaque objet $MPCTaskInstance$ s'abonne à ce service puis cherche les références (*IOR*) des autres $MPCTaskInstance$ et les rangent dans un tableau en associant à chaque couple <nom de tâche, numéro d'instance> l'IOR de son $MPCTaskInstance$.

Chaque instance d'une tâche *MPC*, peut invoquer, via son $MPCTaskInstance$, des méthodes sur les autres $MPCTaskInstance$ des autres instances de tâches. De la même façon, ces dernières peuvent aussi invoquer des méthodes sur ce $MPCTaskInstance$. Par conséquence, chaque instance d'une tâche *MPC* est considérée comme une application cliente et serveur à la fois. Ce que l'on appelle également une application mixte.

D'autre part, et comme nous pouvons le remarquer à travers les exemples présentés précédemment, aucune phase de déclaration de l'objet *CORBA*, $MPCTaskInstance$, n'est nécessaire. Cette déclaration se fait implicitement lors de l'inclusion du fichiers "mpc.h" comme indiqué dans la partie du code suivante :

```
/******  fichier mpc.h    *****/
static mpc MPCTaskInstance = new mpc;
```

L'attribut static est d'une importance primordiale. En effet, supposant que le programme principal d'une tâche fait appel à deux modules séparés et dont chacun inclut le fichier "mpc.h". En l'absence de l'attribut static, une erreur sera déclenchée lors de l'édition de liens faute de duplication de la déclaration de l'objet *MPCTaskInstance*. Pour remédier à cette erreur, une solution possible consiste à déclarer l'objet *MPCTaskInstance* au sein de la fonction main() du programme principal puis passer cet objet comme paramètre lors de chaque appel à une fonction l'utilisant et développée dans un module séparé. Dans ce cas, tout appel à la bibliothèque *MPC* doit être explicitement fait sur cet objet.

Cette solution semble lourde à utiliser notamment pour les utilisateurs non habitués à la programmation orientée objet. La déclaration de l'objet *MPCTaskInstance* dans le fichier "mpc.h" en utilisant l'attribut static permet donc de résoudre ce problème. En effet, il permet dans le cas d'une programmation modulaire séparée, l'unicité de l'objet *MPCTaskInstance* déclaré et qui est donc considéré comme un objet global partagé par l'ensemble des modules.

Dans ce cas, tout appel à la bibliothèque *MPC* est traduit implicitement par l'appel à une fonction du même nom et des mêmes paramètres sur cet objet. Cela permet, d'une part, d'avoir une interface semblable aux bibliothèques de communication habituelles ou aucune déclaration préalable n'est demandée et, d'autre part, de cacher l'approche orientée objet. Par conséquent, un utilisateur habitué à l'approche structurée proposée par le langage C peut facilement utiliser cette bibliothèque sans aucune connaissance préalable du langage C++ et de l'approche orientée objet en général, à condition bien sûr de disposer d'un compilateur C++.

L'interface *MPCTaskInstance*

Le concepteur d'applications séquentielles se doit de structurer son application en, d'une part, les données qu'il doit utiliser et, d'autre part, les traitements qui manipulent ces données. L'approche objet, maintenant bien connue l'aide dans ce travail en lui fournissant une entité unique de structuration (l'objet) qui regroupe logiquement ces deux aspects de la programmation. Le concepteur d'applications réparties est devant un double problème de conception. Il doit représenter les données manipulées par son application (ainsi que les traitements qui les manipulent), mais il doit aussi gérer l'accès concurrent à ces données. Nous détaillerons ces concepts dans les paragraphes suivants.

Le modèle de données

La mise en pratique des modes de communication proposés nécessite l'utilisation au niveau de chaque *MPCTaskInstance* de deux structures dynamiques de zones mémoire temporaires qu'on appelle des buffers :

- Un buffer de réception (recvBuffer): il contient l'ensemble des messages envoyés par les autres tâches et dont la primitive de réception n'est pas encore exécutée. La structure d'un message *MPC* est présentée dans la figure 10.3 ;
- un buffer de descripteurs des messages à recevoir (recvDescBuffer) : il contient la description des différents messages attendus, il s'agit essentiellement de ceux dont la primitive de réception est déjà appelée, en revanche leur envoi n'a pas encore eu lieu. Le descripteur d'un message est une structure de données dont les composants sont indiqués dans la figure 10.3.

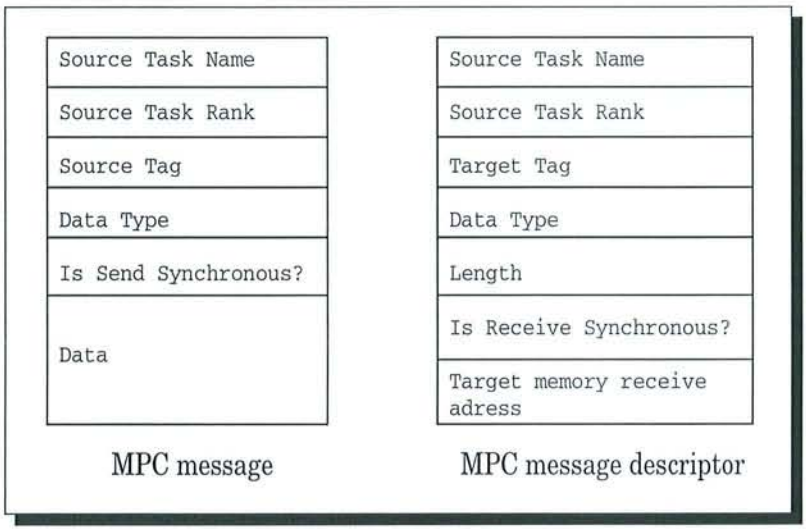

FIG. 10.3 – *Différence dans* MPC *entre un message et une descripteur de message*

Le retrait des messages et des descripteurs de messages des deux buffers suit l'ordre de leur arrivée (l'ordre FIFO: First In First Out).

En plus de ces deux buffers, une variable de type booléen est utilisée pour assurer la synchronisation dans certains modes de communication. On l'appelle dans le reste de ce chapitre la variable de synchronisation.

Rappelons enfin que la gestion de formats de données hétérogènes est réglée implicitement par l'utilisation du protocole CDR (Common Data Representation) définit dans GIOP (General Inter ORB Protocol).

Le modèle de traitements

Ce modèle comporte l'ensemble des traitements relatifs à la manipulation des structures de données présentées auparavant. Mais avant de détailler ce modèle, donnons d'abord l'interface *IDL*, appelé MpcTask, d'un objet *MPCTaskInstance* :

```
interface  MpcTask
{
    void addRecvBuffer(in Message msg);
    void synchronize();
};
```

Cette interface se compose de deux méthodes :

- la méthode addRecvBuffer : comme son nom l'indique, l'invocation de cette méthode sur un objet de type MpcTask permet d'ajouter le contenu du message donné en paramètre dans le buffer de réception de cet objet ;
- la méthode synchronize : elle permet de mettre à jour le contenu de la variable de synchronisation de l'objet appelé à la valeur vraie.

Les appels distants portent sur ces deux méthodes, c'est à dire que ce sont les seuls appels qui engendrent une communication entre les objets MpcTaskInstance. Pour répondre aux appels multiples sur un même objet, l'appel à chacune de ces deux méthodes génère un flux d'exécution sous forme d'un processus léger (ou thread) du coté de l'objet appelé. L'exécution de ce processus est maintenue jusqu'à la fin du traitement associé à cette méthode.

Par ailleurs, l'implantation de cette interface implante ces deux méthodes et l'enrichit en ajoutant des nouvelles fonctionnalités parmi lesquelles on trouve :

- La méthode addRecvDescBuffer : permet d'ajouter la description d'un message de réception dans le buffer des descripteurs des messages à recevoir ;
- la méthode waitSynchronize : cette méthode bloque l'objet appelant jusqu'à ce que la variable de synchronisation ait la valeur vraie, puis remet sa valeur à fausse.

A l'inverse des deux méthodes de l'interface, l'appel à ces deux méthodes n'engendre pas de communication. Ce sont des appels locaux.

Notre implantation comporte plusieurs autres fonctions, mais nous considérons que les méthodes présentées ci-dessus sont suffisantes pour décrire le protocole de communication point à point mis en œuvre.

La gestion de concurrence d'accès

Comme nous l'avons indiqué dans le paragraphe précédent, les appels distant sur un objet $MPCTaskInstance$ reposent sur un modèle à base de processus légers. Par conséquent, il faut introduire un mécanisme pour gérer l'accès multiple et concurrent aux données d'un objet. Pour y parvenir, nous utilisons le mécanisme du moniteur.

Rappelons dans ce cadre qu'un moniteur est une construction syntaxique comportant des variables d'état internes et des procédures (ou points d'entrée) accessibles depuis l'extérieur et qui interdit l'accès concurrent à un ensemble d'instructions. Chaque moniteur est une structure partagée entre plusieurs processus, qui se synchronisent en appelant ses points d'entrée ; les variables d'état ne sont pas directement accessibles. Les procédures du moniteur contiennent des opérations qui permettent de bloquer ou de réveiller les processus qui utilisent le moniteur. Les constructions de synchronisations expriment à l'aide de conditions. Une condition c est déclarée comme une variable, mais ne peut être manipulée qu'au moyen de deux principales primitives : wait : bloque le processus p, et le place en situation d'attente ; notify : réveille un des processus en attente de c.

Le mécanisme des moniteurs vise donc à fournir un mode d'expression de la synchronisation plus structuré que les sémaphores, en vue de faciliter la compréhension et l'écriture des schémas de synchronisation, de donner à ces schémas une forme synthétique, et de permettre la construction de preuves. Dans le cas de MPC, chaque entité de donnée est associée à un moniteur. Par conséquent, nous disposons de trois moniteurs :

- le moniteur du buffer de réception (recvBufferMonitor) ;
- le moniteur du buffer de descripteurs des messages à recevoir (recvDescBufferMonitor) ;
- le moniteur de la variable de synchronisation (synchronizeMonitor).

Dans les sections précédentes, nous avons présenté les modèles de traitements et de données ainsi que le gestion de concurrence d'accès relatifs à l'objet MpcTaskInstance. Nous pouvons maintenant décrire le protocole de communication point à point de la bibliothèque MPC.

L'envoi asynchrone

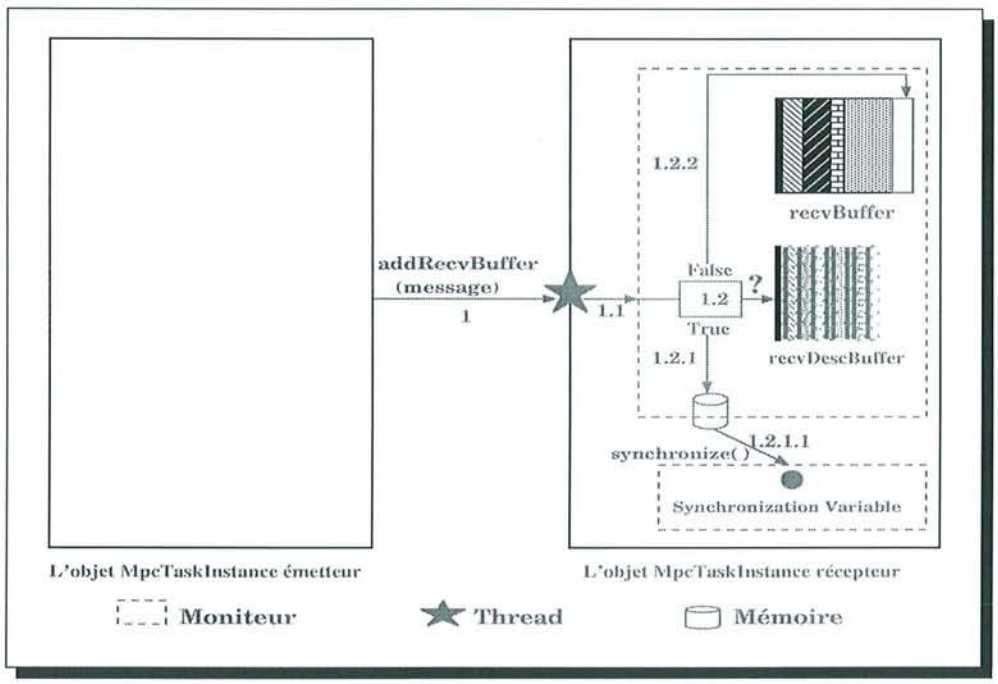

FIG. 10.4 – *L'envoi asynchrone*

Ce type d'envoi se déroule de la façon suivante (les numéros des étapes sont ceux indiqués sur la figure 10.4) :

- 1 L'émetteur invoque la méthode *addRecvBuffer()* avec comme paramètre le message à envoyer sur l'objet MpcTaskInstance récepteur. Cela génère un flux d'exécution coté récepteur ;
  - 1.1 Le flux d'exécution associé à cette méthode verrouille l'accès aux deux buffers de données (buffer de réception et buffer de descripteurs des messages à recevoir) en utilisant leur moniteur.
  - 1.2 Il cherche ensuite si le descripteur de ce message existe dans le buffer de descripteurs des messages à recevoir (i.e. si l'appel de réception de ce message est déjà faite), deux cas de figures se présentent :
    - 1.2.1 Si un tel descripteur existe, le message est copié directement dans la zone mémoire de réception ;
      - 1.2.1.2 Selon le mode de réception qui est indiqué sur le descripteur de ce message, s'il s'agit d'une réception synchrone, la méthode *Synchronize* est appelée localement. Cette méthode verrouille l'accès à la variable de synchronisation puis met sa valeur à vraie, ensuite une notification (notify) est signalée par le moniteur de cette variable pour réveiller le récepteur qui dans ce cas devait être en attente (wait) ;
    - A la fin, le descripteur de ce message est effacé du buffer de descripteurs des messages à recevoir ;
    - 1.2.2 Dans le cas inverse, le message est ajouté dans le buffer de réception.

L'envoi synchrone

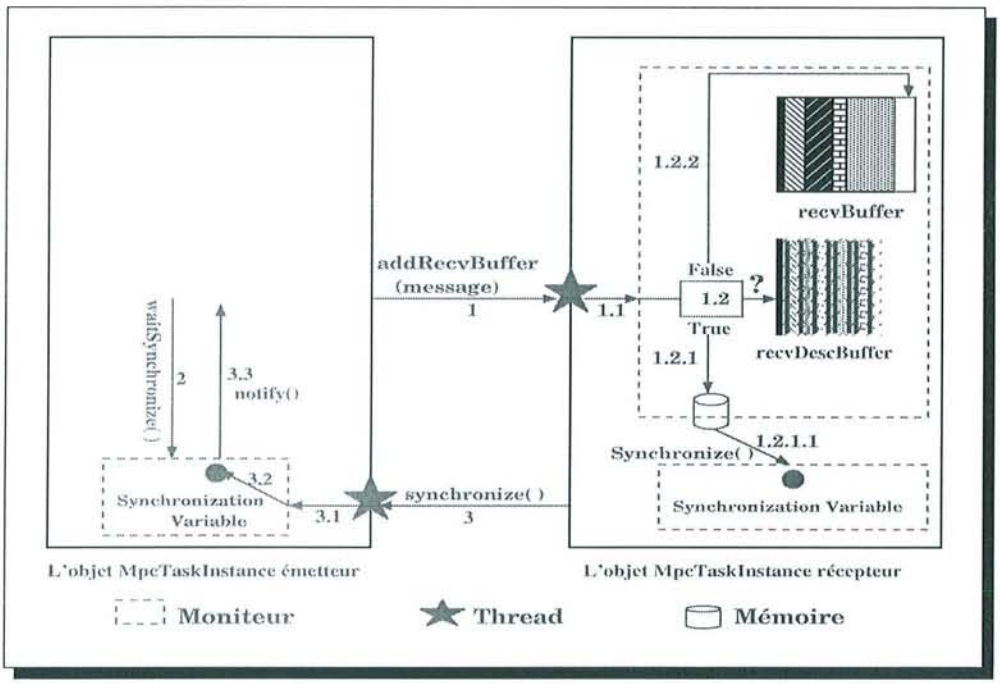

FIG. 10.5 – *L'envoi synchrone*

Ce type d'envoi comporte les étapes suivantes (voir figure 10.5) :

- 1 L'émetteur invoque la méthode addRecvBuffer() avec comme paramètre le message à envoyer sur l'objet récepteur. Cela génère un flux d'exécution coté récepteur qui suit les mêmes traitements que ceux de l'envoi asynchrone.
- 2 Une fois l'invocation de cette méthode est finie, l'émetteur appelle localement la méthode waitSynchronize, cette méthode verrouille l'accès à la variable de synchronisation locale et endormit (wait) l'émetteur tant que la valeur de cette variable est à faux.
- 3 Après l'appel de la primitive de réception coté récepteur, ce dernier invoque la méthode synchronize sur l'objet émetteur. Cette invocation génère un flux d'exécution coté émetteur.
  - 3.1 Ce flux d'exécution qui à son tour verrouille l'accès à la variable de synchronisation.
  - 3.2 La valeur de cette variable est ensuite mise à vraie.
  - 3.3 Une notification (notify) est signalée par le moniteur associée à la variable de synchronisation pour réveiller l'émetteur.

Dans les étapes présentées ci-dessus, nous avons supposé que la demande d'émission est appelée avant celle de la réception. Ce qui explique que l'appel à la fonction synchronize (étape 3) est effectuée après le retour de la fonction addRecvBuffer. Dans le cas inverse, la copie du message se fait au moment de l'appel à la fonction, et par conséquent la fonction synchronize serait invoquée juste après cette copie (i.e. avant le retour de la méthode addRecvBuffer). Ce retour est suivi coté émetteur par l'appel à la fonction waitSynchronize qui à l'inverse du premier cas, ne bloque pas l'émetteur du fait que la variable de synchronisation est déjà positionnée à vraie après l'appel à la fonction synchronize.

La réception asynchrone

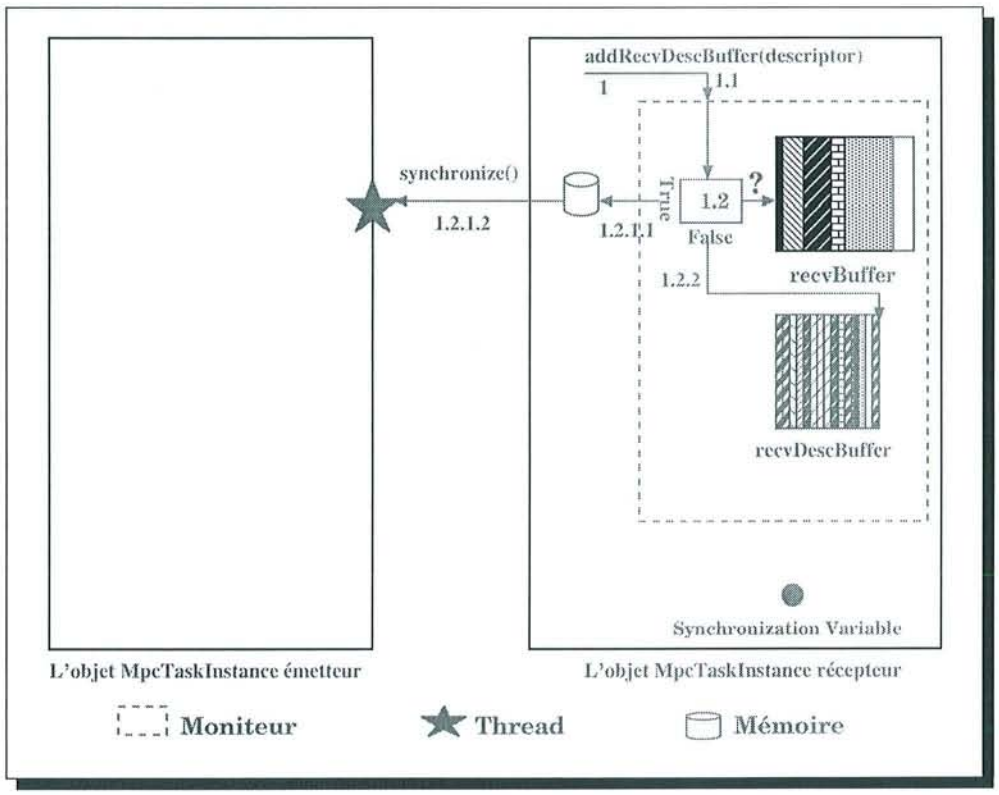

FIG. 10.6 – *La réception asynchrone*

Ce type de réception comporte les étapes suivantes (voir figure 10.6) :

- 1 Le récepteur invoque localement la méthode addRecvDescBuffer().
  - 1.1 L'appel à cette méthode verrouille l'accès aux deux buffers de données (buffer de réception et buffer de descripteurs des messages à recevoir) en utilisant leurs moniteurs.
  - 1.2 Il cherche ensuite si le message dont la description est donné en paramètre existe dans le buffer de réception (i.e. si l'appel d'émission de ce message est déjà fait), deux cas de figures se présentent :
    - 1.2.1 Si un tel message existe :
      - 1.2.1.1 Le message est copié directement dans la zone mémoire de réception.
      - 1.2.1.2 Selon le mode d'émission qui est indiqué sur ce message, s'il s'agit d'une émission synchrone, le récepteur invoque à distant la méthode synchronize sur l'émetteur pour lui indiquer que la synchronisant est déjà établie. A la fin, ce message est effacé du buffer de réception.
    - 1.2.2 Dans le cas inverse, le descripteur du message est ajouté au buffer de descripteurs des messages à recevoir.

La réception synchrone

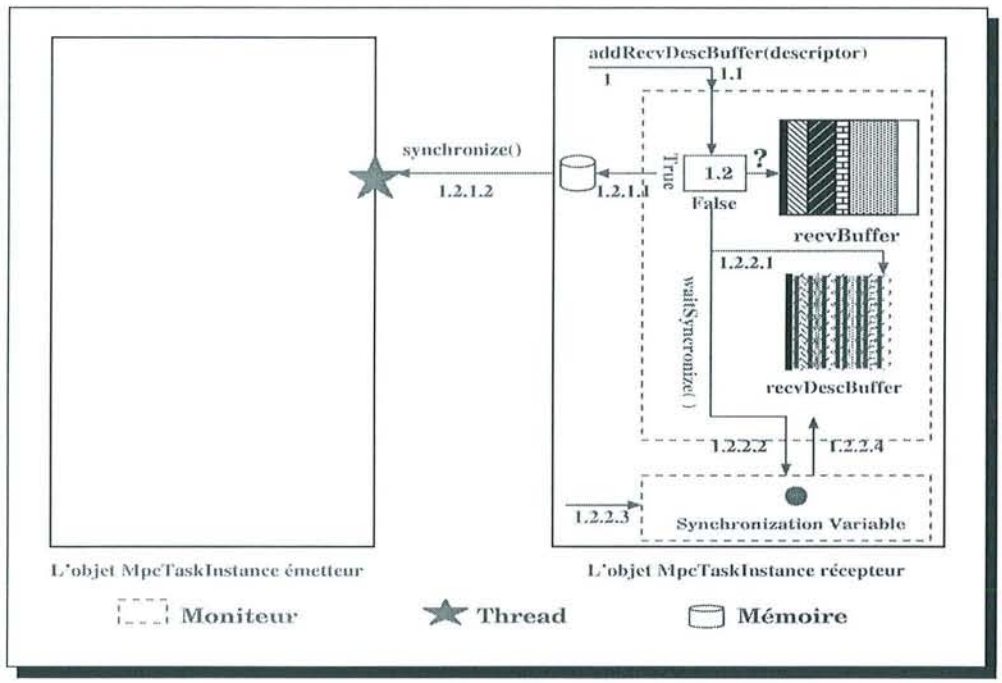

FIG. 10.7 – *La réception synchrone*

Ce type de réception se déroule de la façon suivante (figure 10.7) :

- 1 Le récepteur invoque localement la méthode addRecvDescBuffer().
  - 1.1 L'appel à cette méthode verrouille l'accès aux deux buffers de données (buffer de réception et buffer de descripteurs des messages à recevoir) en utilisant leurs moniteurs.
  - 1.2 Il cherche ensuite si le message dont la description est donné en paramètre existe dans le buffer de réception (i.e. si l'appel d'émission de ce message est déjà fait), deux cas de figure se présentent :
    - 1.2.1 Si un tel message existe. On suit les mêmes traitements que dans le cas de la réception asynchrone.
    - 1.2.2 Dans le cas inverse :
      - 1.2.2.1 Le descripteur du message est ajouté au buffer de descripteurs des messages à recevoir.
      - 1.2.2.2 Le récepteur appelle ensuite localement la méthode waitSynchronize et se met en attente de la modification de la valeur de synchronisation.
      - 1.2.2.3 Lors l'appel de la primitive d'envoi du message attendu, la méthode synchronize serait appelée localement et la valeur de la variable de synchronisation est ensuite mise à vraie ;
      - 1.2.2.4 Une notification (notify) est signalée par le moniteur associée à la variable de synchronisation pour réveiller le récepteur.

Taille de données en octet	MPICH	LAM	PVM	ORBacus	MPC
0	0.46	1.11	0.68	0.88	1.35
64	0.57	1.16	0.78	1.01	1.49
128	0.83	1.33	0.90	1.13	1.61
256	0.93	1.46	1.11	1.34	2.61
512	1.45	1.89	1.59	1.82	2.89
1024	2.40	2.90	2.50	2.75	3.37
2048	4.01	4.07	3.79	4.06	4.64
4096	7.69	5.87	6.20	5.82	6.61
8192	13.18	9.56	11.54	9.86	11.30
16384	21.61	19.21	22.56	18.56	20.35
32768	36.04	33.19	44.20	36.49	39.03
65536	95.50	67.74	88.57	72.40	77.10
131072	209.43	124.74	172.63	142.34	153.16
262144	446.17	245.07	346.67	284.45	304.77
524288	869.30	492.75	690.93	560.49	611.10
1048576	1714.30	978.76	1380.52	1124.45	1214.59

TAB. 10.1 – *Temps d'envoi en ms de messages sur MPICH, LAM, PVM, ORbacus et MPC en fonction de la taille de données*

Autour du protocole proposé

Le protocole que nous venons de décrire présente les caractéristiques suivantes :

- Indépendance vis-à-vis des envois réceptions. En effet, l'émetteur (respectivement le récepteur) n'est pas obligé de connaître le mode de réception (respectivement d'émission) du message à envoyer (respectivement à recevoir) ;
- l'utilisation des modes non bloquants permet de couvrir le temps de communication tant que le contenu de la zone mémoire d'envoi ou de réception ne soit pas utilisé ;
- une interface IDL simple ;
- une seule invocation de méthode à distance (communication) est effectuée pour chaque envoi-réception sauf dans le cas de l'envoi synchrone qui nécessite deux invocations.

Au niveau performances, nous avons développé un premier prototype de la bibliothèque *MPC* en utilisant le bus de *CORBA* ORBacus. Nous avons effectué une comparaison au niveau du temps de communication entre cette implantation de *MPC*, *Orbacus*, et les bibliothèques d'échange de messages *PVM* et *MPI* (MPICH et LAM) en utilisant deux stations SUN Ultra 5 interconnectées par un réseaux Ethernet 10 Mbit/s. Nous avons mesuré le temps nécessaire à l'échange de données entre deux processeurs pour des tailles de données différentes (ping-pong classique). La figure 10.8 et le tableau 10.1 présentent une comparaison entre les différents temps de communication obtenus.

Concernant les résultats de tests obtenus, en calculant la différence au niveau du temps d'envoi d'un message de taille 0 entre la bibliothèque *MPC* et le bus CORBA ORBacus on obtient la latence de cette bibliothèque, qui est dans ces tests d'environ 47ms. D'autre part, pour les messages de moyennes et grandes tailles, le surcoût de la bibliothèque *MPC* est d'environ 10% du temps de communication initial réalisé sur ORBacus. Ce surcoût nous semble raisonnable vu l'utilisation dans la bibliothèque *MPC* des mécanismes supplémentaires outre que l'invocation de méthode. Rappelons dans ce cadre que le temps de communication obtenu avec ORBacus est

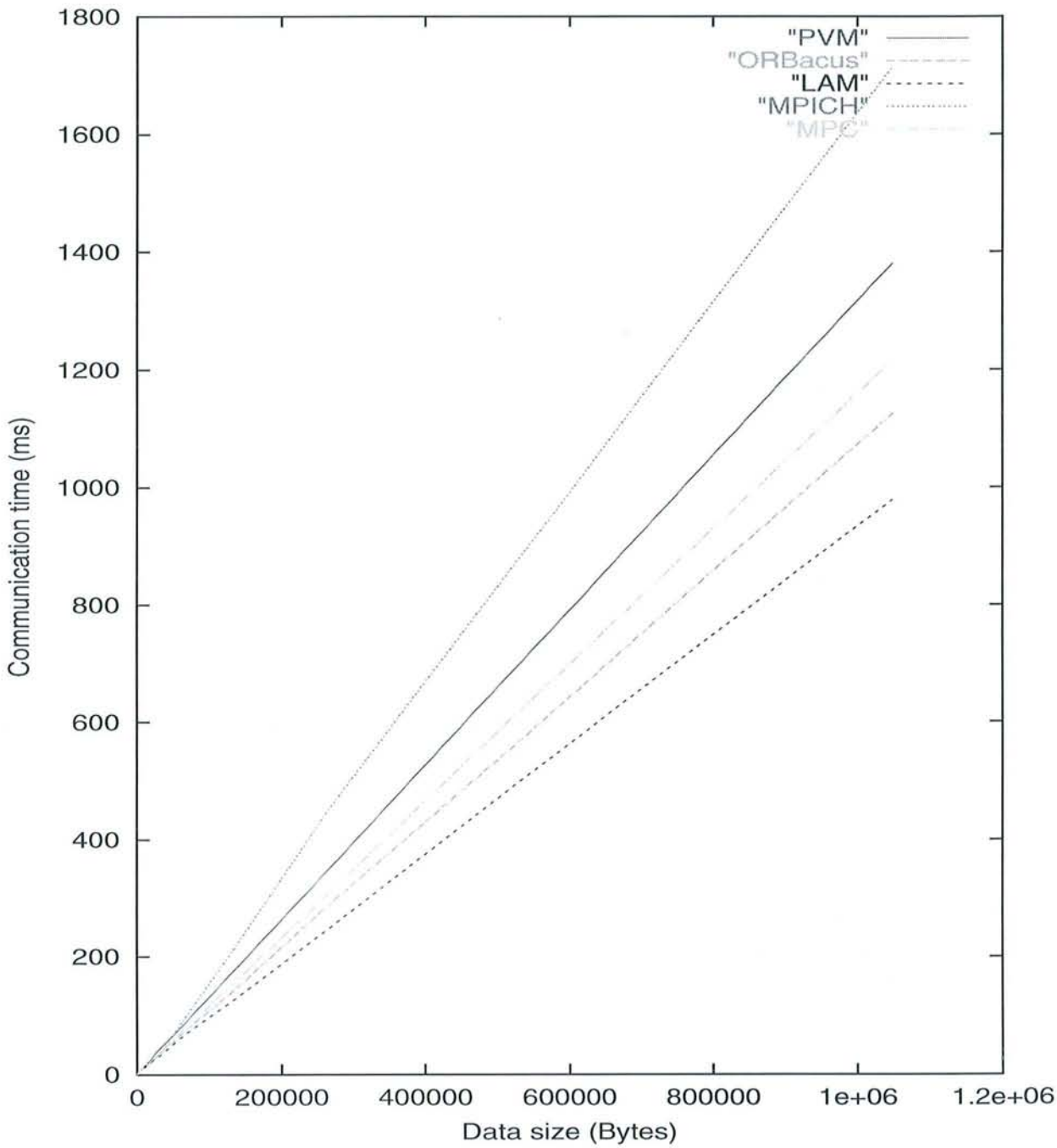

FIG. 10.8 – *Comparaison des performances des communications entre ORbacus, MPC, MPICH, LAM et PVM*

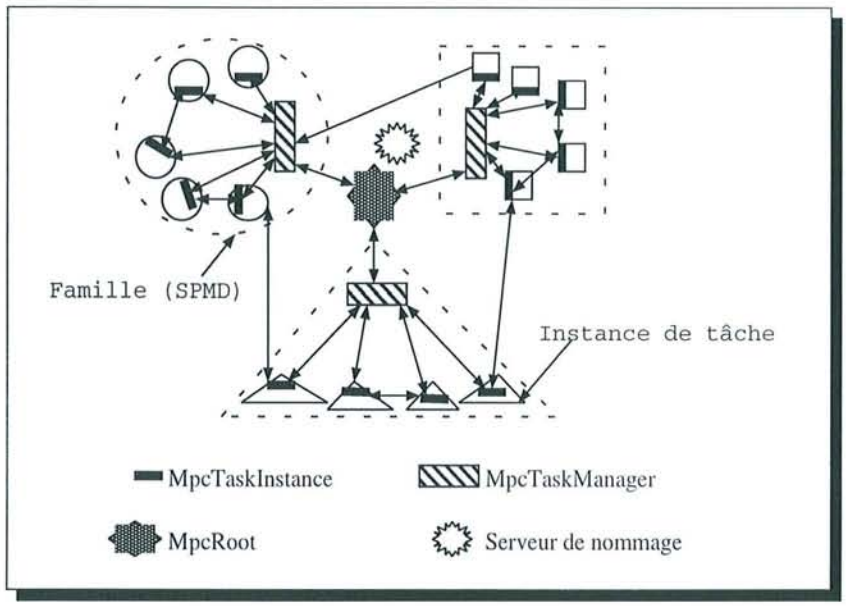

FIG. 10.9 – *L'environnement global de MPC*

celui d'invocation à distance d'une méthode. Cependant, celui de la bibliothèque *MPC* engendre, en plus de l'invocation à distance d'une méthode, l'appel à plusieurs fonctions locales, la gestion de la concurrence d'accès en utilisant les moniteurs et enfin la gestion de la synchronisation. Ce faible surcoût de la bibliothèque *MPC* lui permet de réaliser des temps de communication compétitifs avec ceux réalisés avec les bibliothèques d'échange de messages utilisées : *MPI* et *PVM*.

10.4.2 Les communications collectives

Si les communications point à point entre les différentes instances des tâches se font via leur objet *MPCTaskInstance*, la communication entre un groupe de tâches (les instances d'une même famille) nécessite l'utilisation d'un gestionnaire de groupe. On l'appelle dans *MPC* l'objet *MPCTaskManager*. Cet objet assure la coordination et la synchronisation entre les instances d'une même famille, et entre ces instances et les autres familles (voir figure 10.9).

D'autre part, l'ensemble des *MPCTaskManager* est relié à un objet central qui s'appelle *MPCRoot*. Cet objet assure les communications qui impliquent tous les objets *MPCTaskInstance* d'une application (voir figure 10.9).

Dans le reste de ce chapitre, nous présentons comment utiliser l'ensemble de ces composants pour réaliser l'une des communications collectives la plus utilisée à savoir la diffusion. Notre modèle de diffusion [Es-sqalli 00b] repose sur l'utilisation du service d'événements de CORBA. Toutes les autres communications collectives sont basées sur ce modèle et les primitives de communications point à point présentées auparavant. Mais avant de présenter ce modèle, nous donnons une brève introduction sur ce service.

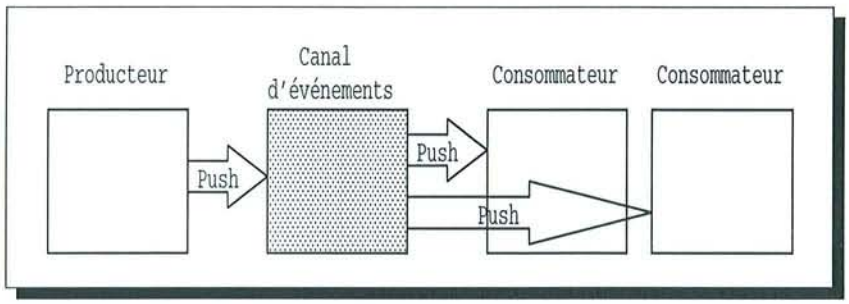

FIG. 10.10 – *Le modèle dépôt*

10.5 Le service d'événements de CORBA

Le service d'événements permet à des objets d'enregistrer ou d'annuler dynamiquement leurs intérêts pour des événements spécifiques. Un événement est une occurrence à l'intérieur d'un objet qui est déclarée comme pouvant présenter un intérêt pour un ou plusieurs objets. Une notification est un message envoyé aux parties intéressées les informant qu'un événement particulier s'est produit. Normalement, l'objet qui a généré l'événement n'a pas à savoir qui sont les parties intéressées. Tout cela est traité par le service d'événements qui crée un canal de communication entre les objets qui savent peu de choses les uns des autres.

10.5.1 Les producteurs et les consommateurs d'événements

Le service d'événements relaie la communication entre objets. Il définit deux rôles pour les objets : producteurs et consommateurs. Les producteurs fournissent les événements, les consommateurs les traitent par des programmes de traitement d'événements. Les producteurs et les consommateurs se communiquent les événements au moyens de requêtes CORBA standards. Il existe deux modèles pour communiquer les données d'un événement : dépôt (push) ou retrait (pull). Dans le modèle dépôt, le producteur d'événements prend l'initiative et déclenche le transfert des données de l'événement aux consommateurs. Dans le modèle retrait, le consommateur prend l'initiative et demande les données d'un événement à un producteur. Un canal d'événements (event channel) est un objet intermédiaire, qui est à la fois producteur et consommateur d'événements. Il permet à des producteurs multiples de communiquer de manière asynchrone avec des consommateurs multiples sans que ceux-ci se connaissent. C'est un objet CORBA standard qui se trouve sur le bus d'objets répartis, et qui relaie les communications entre producteurs et consommateurs.

Le canal d'événements, prend en charge les deux modèles de notification d'événements : dépôt et retrait (figure 10.10 et figure 10.11).

- Le modèle dépôt (push model) : les producteurs prennent l'initiative d'envoyer les événements aux consommateurs. Dans ce cas, le producteur est appelé un producteur actif et le consommateur est appelé un consommateur réactif. En effet, le producteur actif (push supplier) déclenche l'opération de dépôt et le consommateur réactif (push consumer) est notifié par l'avènement d'un événement sur le canal. Les producteurs actifs contrôlent l'émission des données vers le canal, et les consommateurs réactifs sont notifiés par appel du canal d'événements.
- Le modèle retrait (pull model) : dans ce modèle, les événements sont toujours délivrés par les serveurs à destination des clients. Les méthodes d'invocation sont cependant différentes

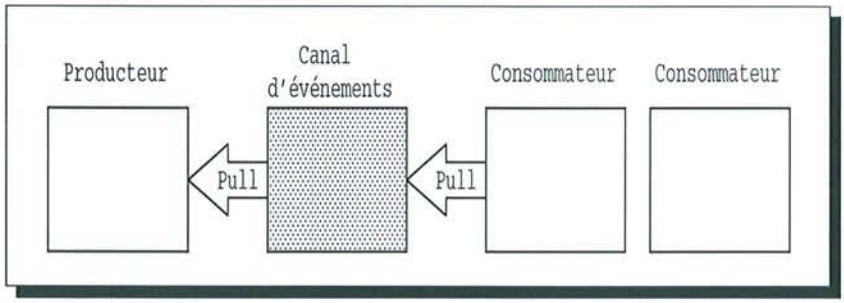

FIG. 10.11 – *Le modèle retrait*

du côté de client et du côté de serveur. En effet, un client actif (pull consumer) décide d'invoquer lui même des méthodes sur le canal d'événements lorsqu'il est prêt à les traiter. De ce fait, le canal maintient une file d'événements pour tous les clients de ce type. Lorsque la file est pleine, le service d'événements remplace la plus ancienne référence par la plus récente. Le consommateur dans ce cas est appelé un consommateur actif et le producteur est un producteur réactif (pull supplier).

L'objet générique canal d'événements ne connaît pas le contenu des données qu'il transmet. Ce sont les producteurs et les consommateurs qui se mettent d'accord sur une sémantique d'événements commune. Cependant, le service d'événements reconnaît également un modèle d'événements typé (Typed event) qui permet aux applications de décrire le contenu d'événements en utilisant l'*IDL*, on parle alors d'un canal d'événements typés.

10.5.2 L'architecture objet d'un canal

Dans la spécification OMG, on introduit une nouvelle notion : les mandataires (proxies). Un mandataire est une représentation d'un producteur ou d'un consommateur au sein du canal. Cette représentation est unique puisque le mandataire ne peut représenter à la fois deux producteurs ou deux consommateurs. Le canal d'événements fournit deux fabriques : le premier pour les producteurs (Supplier Admin) et le second pour les consommateurs (Consumer Admin). Un producteur (respectivement consommateur) est toujours connecté à un mandataire consommateur (resp mandataire producteur). Par exemple un producteur actif (push supplier) est toujours lié à un mandataire consommateur réactif (proxy push consumer).

Pour mieux comprendre le fonctionnement de ce service, nous présentons dans le paragraphe suivant un exemple de son utilisation dans le modèle dépôt (producteur actif et consommateur réactif).

10.5.3 Exemple d'utilisation du service d'événements

Le producteur actif Chaque producteur actif utilise le même scénario pour accéder aux services du canal d'événements, et qui est le suivant (voir figure 10.12) :

- a) Phase de connexion :
  - Obtenir la fabrique des mandataires des producteurs (Supplier Admin) en invoquant la méthode "for_suppliers()" sur le canal d'événements.
  - Obtenir le mandataire des consommateurs réactifs en invoquant la méthode " obtain_push_consumer() " sur la fabrique des mandataires des producteurs.

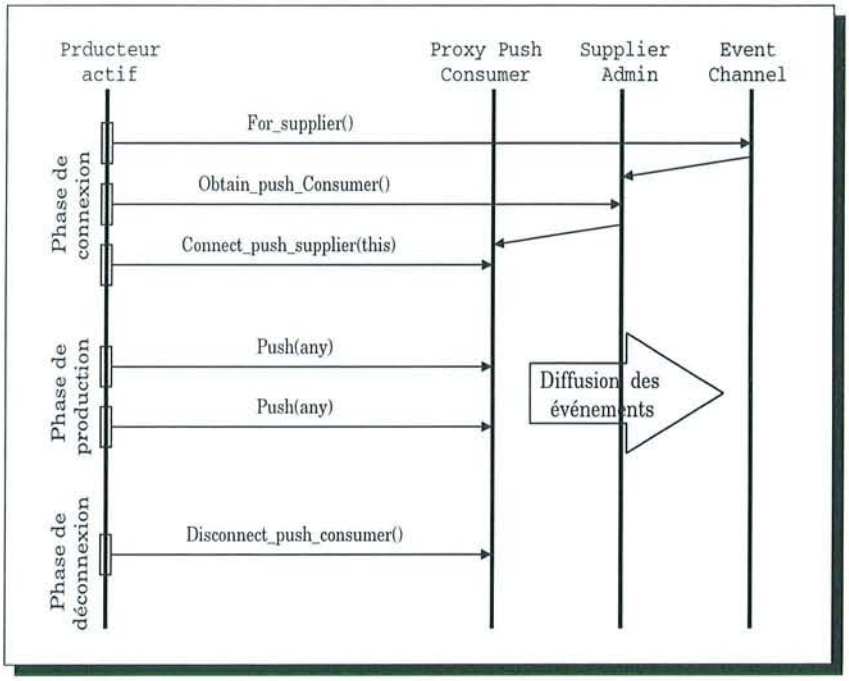

FIG. 10.12 – *Le scénario d'un producteur actif*

- La dernière étape de la phase de connexion est la connexion au mandataire des consommateurs réactifs.

- b) Phase de production : cette étape consiste en la génération des événements par le producteur actif.
- c) Phase de déconnexion.

Le consommateur réactif Chaque consommateur réactif possède le même scénario pour accéder aux services du canal d'événements, il doit donc suivre les phases suivantes (voir figure 10.13) :

- a) Phase de connexion :
  - Obtenir la fabrique des mandataires des consommateurs en invoquant la méthode "for_consumers()" sur le canal d'événements.
  - Obtention du mandataire des producteurs actifs en invoquant, sur la fabrique des mandataires des consommateurs, la méthode "obtain_push_supplier()" .
  - La dernière étape de la phase de connexion est la connexion au mandataire des producteurs actifs.
- b) Phase de consommation : cette étape consiste en la consommation des événements par le consommateur réactif.
- a) Phase de déconnexion.

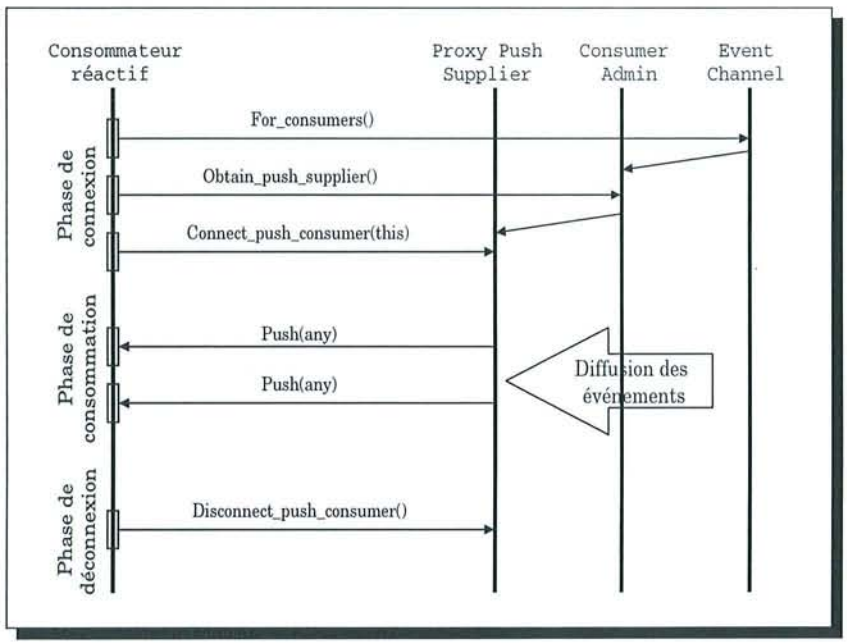

FIG. 10.13 – *Le scénario d'un consommateur réactif*

10.6 Le modèle de diffusion de la bibliothèque *MPC*

Le gestionnaire du groupe

Nous commençons par décrire l'interface *IDL* de l'objet *MPCTaskManager*, nous montrerons que les méthodes relatives à l'opération de diffusion :

```
interface MpcTaskManager // pour faire des opérations globales
{
  void    Lbcast(in any message);  // diffusion locale
  void    Gbcast(in any message) ; // diffusion globale
};
```

Comme nous pouvons remarquer, cette interface se compose de deux méthodes :

- La méthode Lbcast : il s'agit d'une diffusion locale, l'invocation de cette méthode sur un objet MpcTaskManager permet de diffuser le message donné en paramètre à l'ensemble des instances de tâches (la famille) gérées par cet objet. Cette méthode peut être invoquée par un objet *MPCTaskInstance* de la même famille ou appartenant à une autre famille. La diffusion locale concerne notamment les messages de données.
- La méthode Gbcast : à l'inverse de la méthode précédente, cette méthode permet une diffusion globale du message donné en paramètre. Les messages diffusés dans ce cadre sont soit des messages de données ou d'indication d'un événement : lancement (resp. arrêt) d'une nouvelle (resp. ancienne) instance de tâche, ou d'un gestionnaire de groupe, etc.

10.6.1 La diffusion locale

Chaque gestionnaire de groupe dispose d'un canal d'événements, puis un producteur actif connecté à ce canal. De son coté, un objet *MPCTaskInstance* dispose d'un consommateur

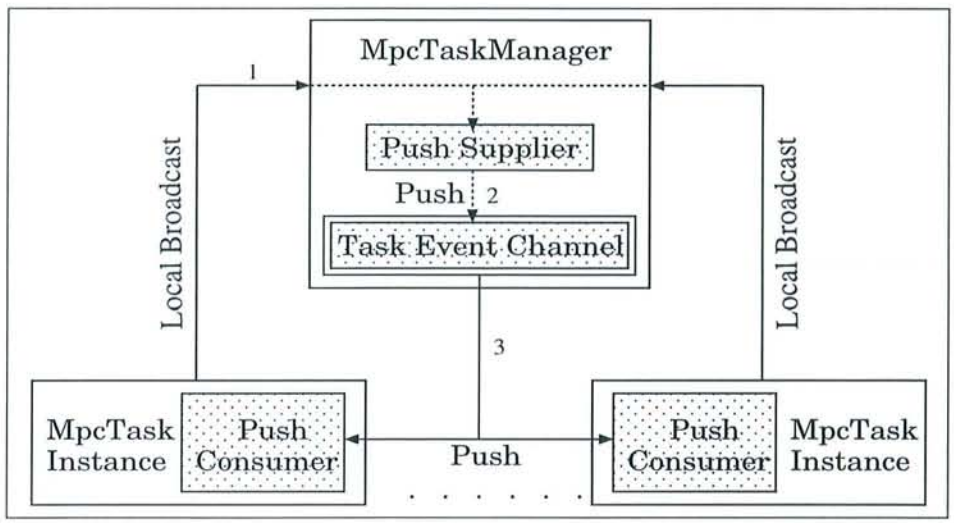

FIG. 10.14 – *La diffusion locale*

réactif connecté au canal d'événement de son gestionnaire de groupe (voir figure 10.14).

La diffusion locale d'un message suit les étapes suivantes :

1. Un objet *MPCTaskInstance* invoque la méthode de diffusion locale (Lbcast) sur son gestionnaire de groupe ;
2. l'invocation de cette méthode est traduite par un appel à la méthode Push sur le producteur actif connecté au canal d'événement de ce gestionnaire ;
3. le canal d'événement diffuse ce message à tous les consommateurs réactifs qui y sont connectés ;
4. chaque *MPCTaskInstance* récupère ce message et le traite.

Rappelons que tous les gestionnaires des groupes sont abonnés au serveur de nommage lancé au début de l'exécution d'une application *MPC*. Par conséquent, un objet *MPCTaskInstance* peut diffuser un message vers un autre groupe à part son groupe d'origine. Il suffit de chercher la référence du gestionnaire de ce groupe et invoquer la méthode Lbcast sur cet objet.

10.6.2 La connexion gestionnaire de groupe et MPCRoot

Comme nous l'avons indiqué auparavant, tous les gestionnaires de groupes sont reliés à un objet central qui s'appelle *MPCRoot*. Cet objet n'est rien d'autre qu'un canal d'événements. Chaque gestionnaire de groupe dispose d'un producteur actif connecté à ce canal. En même temps, chaque canal d'événements d'un gestionnaire de groupe est connecté au canal d'événement du *MPCRoot* comme étant un consommateur réactif. On peut obtenir une telle connexion en connectant le mandataire des consommateurs actifs de la façons suivante (figure 10.15) :

1. obtenir un mandataire de producteur actif (ProxyPushSupplier) du canal d'événements du *MPCRoot* ;
2. obtenir un mandataire de consommateur réactif (ProxyPushConsumer) du canal d'événements du gestionnaire de groupe (the *MPCTaskManager* event channel) ;
3. invoquer la methode connect_push_supplier sur le mandataire de consommateur réactif (ProxyPushConsumer) en lui passant comme paramètre le mandataire de producteur actif, (ProxyPushSupplier) ;

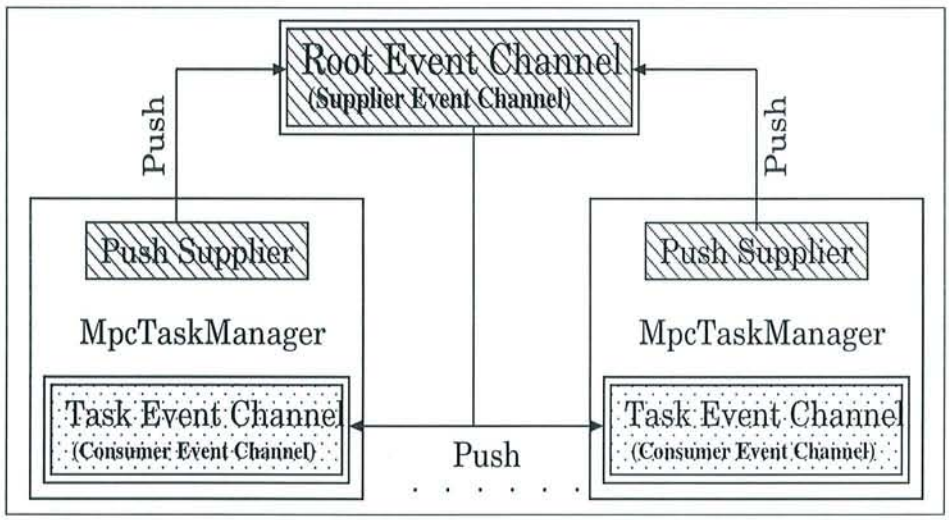

FIG. 10.15 – *La connexion gestionnaire de groupe et MPCRoot*

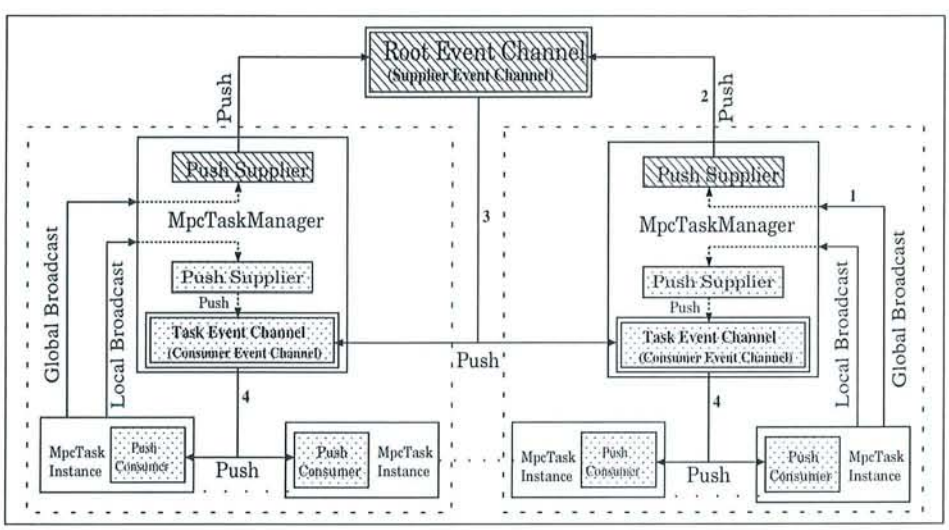

FIG. 10.16 – *Le modèle de la diffusion globale*

4. invoquer la méthode connect_push_consumer sur le mandataire de producteur actif (ProxyPushSupplier) en lui passant comme paramètre le mandataire de consommateur réactif (ProxyPushConsumer).

Après cette connexion, le canal d'événements central sera considéré comme un canal d'événements producteur et ceux des gestionnaires de groupes comme des canaux d'événements consommateurs. Chaque message arrivé au canal d'événements du *MPCRoot* sera diffusé à l'ensemble des canaux d'événements des gestionnaires de groupes.

10.6.3 La diffusion globale

En regroupant les deux modèles précédents, nous obtenons le modèle de la diffusion globale comme indiqué dans la figure 10.16.

Une telle diffusion nécessite les étapes suivantes :

1. Un objet *MPCTaskInstance* invoque la méthode de diffusion globale (Gbcast) sur son gestionnaire de groupe ;
2. l'invocation de cette méthode est traduite par un appel à la méthode Push sur le producteur actif connecté au canal d'événement du *MPCRoot* ;
3. le canal d'événement central diffuse ce message à l'ensemble des consommateurs réactifs qui y sont connectés. Il s'agit dans ce cas des canaux d'événements des gestionnaires de groupes ;
4. le canal d'événements de chaque gestionnaire de groupe diffuse ce message à tous les consommateurs réactifs qui y sont connectés ;
5. chaque *MPCTaskInstance* récupère ce message et le traite.

10.7 Conclusion

Nous avons décrit dans ce chapitre *MPC*, qui est une bibliothèque de communication par échange de messages dans un environnement *CORBA*. Nous avons présenté le protocole de communication mis en œuvre puis quelques résultats expérimentaux de la première implantation de cette bibliothèque. Ce premier prototype nous a montré la faisabilité de cette bibliothèque. Cependant, plusieurs critiques et propositions pour améliorer cette bibliothèque nous ont été faites :

- Au niveau de l'interface de communication, la question qui était posée est pourquoi ne pas utiliser celui de *MPI* qui est actuellement le standard au niveau de la programmation par échange de messages. La réponse à cette question est que nous avons essayé dans une première étape de montrer juste l'intérêt pour proposer une bibliothèque d'échange de messages dans *CORBA* en faisant abstraction de son interface de communication. Pour parvenir à ce but, nous avons proposé une interface beaucoup plus simple que celle de *MPI*. Cependant, choisir comme interface celle de *MPI* est l'une de nos perpectives en prenant comme point de départ les démarches et les protocoles de communication utilisés dans *MPC* ;
- au niveau des performances : dès qu'on parle de *CORBA* la question se pose au niveau des performances de communications. Il est vrai que les premières implantations de *CORBA* n'étaient pas performantes. Cependant, et comme nous l'avons vu dans les deux derniers chapitres, il existe de plus en plus des bus *CORBA* dont les performances sont compétitives à celles des bibliothèques d'échange de messages. Par conséquent, il est évident que les performances de la bibliothèque *MPC* sont liées à celles du bus *CORBA* utilisé ;
- au niveau des protocoles de communication : la question concernait la correction du protocole de communication point-à-point utilisé. Pour ce faire, nous avons validé ce protocole en utilisant l'outil de la validation formelle TLA. Nous présenterons cette validation dans la dernière partie de ce rapport.

Cinquième partie

Expérimentation et validation

Chapitre 11

Expérimentation du langage MeDLey

11.1 Introduction

Dans ce chapitre, nous présentons l'expérimentation du langage MeDLey et qui est basée sur l'implantation du code "Vlasov" déjà existant dans une version parallèle en Fortran.

Ce travail [Es-sqalli 99b] a été effectué dans le cadre de notre collaboration avec le laboratoire de recherche en physique *LPMI* à l'Université Henri Poincaré de Nancy.

11.2 Le code Vlasov

11.2.1 Généralités

Les plasmas de fusions thermonucléaires sont le siège d'instabilités non linéaires d'origine hydro-dynamique et cinétique [Ghizzo 93], évoluant sur des échelles de temps très différentes.

Une meilleure compréhension de ces phénomènes peut être apportée par la simulation numérique. Celle-ci, située à mi-chemin entre la théorie et l'expérience, permet soit de valider ou non une théorie, soit de décrire plus précisément les mécanismes mis en jeu.

Le modèle eulérien de Vlasov est adapté à l'étude d'instabilités de type cinétique. Il consiste à résoudre l'équation de Vlasov qui décrit l'évolution de la dynamique des particules en terme de fonction de distribution. Il est dit "eulérien" par opposition au modèle "lagrangien" (code PIC, Particules-In-Cell) qui lui résout les équations du mouvement.

L'équation de Vlasov est l'analogue semi-classique de l'équation de Hartree-Fock. Elle régit l'évolution de la fonction de distribution à 1-corps.

11.2.2 Code existant

Une version parallèle du code Vlasov a été déjà développée par des physiciens de LPMI [Sonnendrucker 98], cette version écrite en langage Fortran s'appuie sur l'utilisation des appels de la bibliothèque de communication *MPI*.

Dans les paragraphes suivants, nous présentons brièvement les transformations et les améliorations apportées au code initial. Par la suite, nous décrivons comment implanter ce code en utilisant le langage MeDLey. Enfin nous présentons nos résultats expérimentaux.

11.2.3 Etapes préliminaires

Deux étapes ont été effectuées sur la version parallèle du code Vlasov :

- Etape de transformation du Fortran en langage C ;
- étape d'optimisation.

11.2.4 Phase de transformation

Le programme initial du Code Vlasov a été écrit en Fortran. Toutefois, l'utilisation des sources générées par la version de MeDLey disponible jusqu'à présent doit se faire en C ou en C++. Par conséquent, la première étape de cette expérimentation était la phase de réécriture qui a consisté à traduire le corps du programme du Fortran en C.

11.2.5 Phase d'optimisation

Le recours au parallélisme peut accélérer considérablement l'exécution des algorithmes. Mais avant de paralléliser un code, et pour une utilisation plus efficace de cette technique, le temps d'exécution séquentielle de ce code doit être « optimal ». De nombreuses méthodes peuvent être envisagées pour optimiser le temps d'exécution des codes. Nous donnons ici une liste de quelques techniques qui nous ont servi à optimiser la version initiale du code Vlasov :

- Elimination des sous-expressions communes : il arrive fréquemment que l'adresse d'une donnée dans un tableau soit calculée plusieurs fois. Il est possible de conserver cette adresse afin d'éviter de la recalculer ;
- elimination du code inutile : certaines parties de code sont générées mais ne peuvent être atteintes. Il n'est pas nécessaire de les conserver ;
- déplacement du code : les calculs effectués dans une boucle qui sont indépendants de celle-ci, peuvent être sortis de la boucle ;
- variable d'induction : les calculs sur les variables d'induction peuvent être transformés afin d'utiliser des opérations moins coûteuses en temps.
- réduction de force : comme pour les variables d'induction, il est possible de transformer les expressions arithmétiques en expressions moins coûteuses en temps de calcul ;
- transformations directes des boucles : les différents types de transformations peuvent être classés en fonction du type de modification qu'elles effectuent. On considère habituellement les transformations de l'ordre d'exécution, les transformations effectuant des destructions de dépendances, les transformations liées aux accès mémoire, et certaines autres transformations comme la fusion de deux ou plusieurs boucles en une boucle unique, et l'éclatement d'une boucle en plusieurs.

L'utilisation de ces techniques pour optimiser la version initiale du code Vlasov nous a permis d'obtenir un gain de 15 % sur le temps d'exécution global.

11.3 Implantation avec MeDLey

Le code Vlasov consiste en une série de transformations appliquées à la matrice de distribution de départ. Les transformations successives font converger les valeurs de cette matrice jusqu'à ce que la différence entre deux itérations soit négligeable.

La parallélisation de ce code consiste donc à distribuer la matrice de départ sur l'ensemble des processeurs qui vont effectuer des transformations parallèles et similaires sur leurs données locales. Une étape de rassemblement est effectuée après chaque série de transformations. Cette étape est suivie de nouveau par une distribution si la condition d'arrêt n'est pas satisfaite. Une

méthode de parallélisation possible de ce code consiste à définir une tâche qui s'occupe de la distribution et de rassemblement, et une autre tâche dont les instances effectueront en parallèle des transformations sur la matrice de distribution. Dans ce cas, la première tâche sera inactive pendant la phase de transformation. Pour remédier à ce problème, on peut définir une seule tâche, une instance particulière de cette tâche s'occupe de la distribution et du rassemblement. Cette instance avec les autres instances de la même tâche vont effectuer en parallèle les transformations sur la matrice de distribution. Pour utiliser MeDLey dans un tel cas, on se place dans le mode d'interconnexion correspondant au modèle SPMD. On définit ensuite les données utilisées par cette tâche, puis on spécifie les messages à envoyer et à recevoir.

11.3.1 Spécification des messages

Nous spécifions dans ce paragraphe les communications que requiert la tâche Vlasov utilisée pour paralléliser le code de notre application. Les données utilisées dans la phase de communication sont déclarées en premier lieu dans la partie *uses*, suivie par la partie *sends* où l'on spécifie les messages à envoyer, enfin la partie *receives* qui contient la spécification des messages à recevoir.

```
Task Vlasov(i) connected with setof (Vlasov)

uses
 {
    // Données utilisées dans  engu.c, density.c
    vector<double>[NXPROC] Rhops;
    vector<double>[NXPROC] Work;

    // Données utilisées dans engu.c
    vector<double>[NXPROC] Ekp;
    vector<double>[NXPROC] Xwork;

    // Données utilisées dans transf.c, itransf.c
    matrix<double> [NXPROC,NVXPROC] fWork;

    // Données utilisées dans density.c
    vector<double>[N] Ef;

 }

sends
 {
    // Structures envoyées dans la fonction engy, density
     Srhop = sequence{Rhop};
     SEkp  = sequence{Ekp};

    // Structures envoyées dans la fonction transf, itransf
     Sfwork = sequence{fWork};
```

```
    // Structures envoyées dans la fonction density
     Set to Vlasov = sequence{Ef};
  }

receives
  {
    // Structures reçues dans la fonction engy
     RWork = sequence{Work} matches SRhop;
     RXwork = sequence{Xwork} matches SEkp;

    // Structures reçues dans la fonction transf
     RfWork = sequence{fWork} matches SfWork;

    // Structures reçues dans la fonction density
     REf from Vlasov = sequence{Ef} matches SEf;
  }
```

11.3.2 Résultats expérimentaux

Dans ce paragraphe, nous donnons les différents résultats numériques concernant l'exécution de la version parallèle du code Vlasov en utilisant MeDLey. Ces résultats ont été obtenus sur un Power-challenge array (Silicon Graphics) à 18 processeurs [Es-sqalli 98]. Le code source du code Vlasov en utilisant *MeDLey* se trouve dans l'annexe D.

Nous avons exécuté ce code en fonction de la taille (N) de la matrice de distribution initiale, du nombre de processeurs utilisés (NPROC), ainsi que de la version de l'implantation MeDLey à savoir celle qui utilise les primitives de communication de la bibliothèque MPI (MDL (MPI)), et l'autre version utilisant les primitives de gestion de mémoire partagée (MDL (SHM)). Les différents résultats numériques de cette expérimentation sont donnés dans le tableau 11.1. Le temp de calcul (Calcul-Time) représente le temps d'exécution sans l'utilisation des primitives de communication. (I.e. Temps d'exécution - temps de communication).

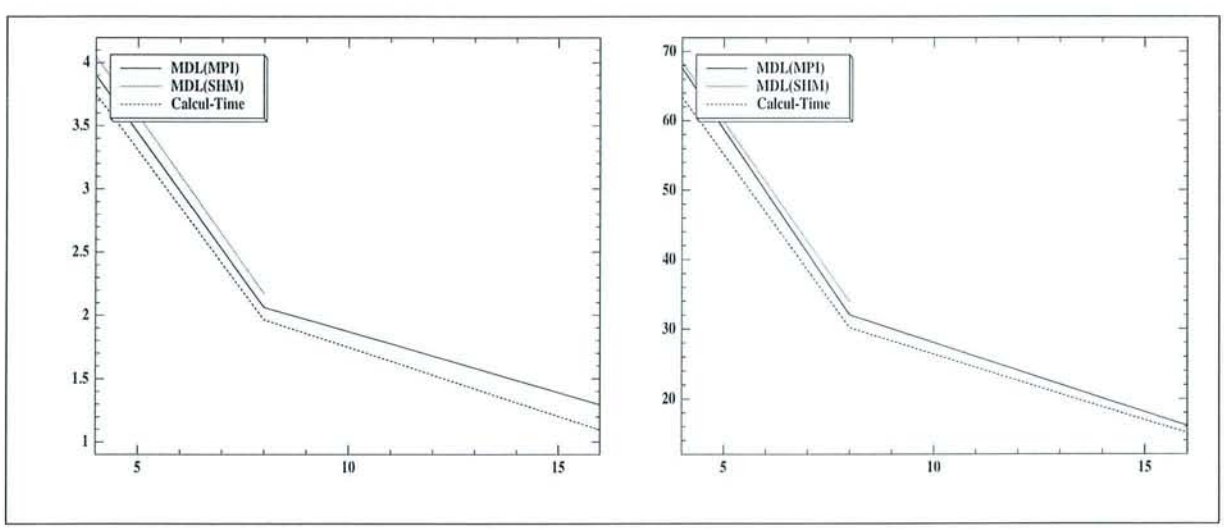

FIG. 11.1 – *Temps d'exécution du code Vlasov en fonction de NPROC et de la version de l'implantation MeDLey pour N = 256 (à gauche) et N=1024 (à droite)*

Cas 1 : NPROC = 4

N	256	512	1024	2048
MDL (MPI)	3.901	15.361	67.597	367.048
MDL (SHM)	4.048	15.952	68.401	377.062
Temps de calcul	3.751	14.668	63.398	349.664

Cas 2 : NPROC = 8

N	256	512	1024	2048
MDL (MPI)	2.062	7.986	32.021	269.331
MDL (SHM)	2.173	8.225	33.932	211.876
Temps de calcul	1.964	7.506	30.175	175.304

Cas 3 : NPROC = 16

N	256	512	1024	2048
MDL (MPI)	1.296	4.489	16.138	78.733
MDL (SHM)	?	?	?	?
Temps de calcul	1.095	4.385	15.084	66.805

TAB. 11.1 – *Temps d'exécution (en seconde) du code Vlasov en fonction de N, NPROC et de la version de l'implantation MeDLey sur Power Challenge Silicon Graphics*

11.3.3 Constatations

- Au niveau de l'accélération :
  - Indépendamment de la taille de la matrice de distribution, on obtient une accélération moyenne de 1.9 quand le nombre de processeurs passe de 4 à 8, et de 1.8 dans le cas du passage du nombre de processeurs de 8 à 16, sauf pour la version de MDL(SHM) qui se bloque quand on dépasse 8 processeurs. Ces valeurs montrent qu'en augmentant le nombre de processeurs, on obtient une accélération presque linéaire (voir figures 11.1). De ce fait, ce code utilise efficacement les capacités de calcul des processeurs ajoutés à chaque fois.
  - Indépendamment du nombre de processeurs, on obtient une accélération moyenne de 3.8 quand la taille de la matrice de distribution passe de 512 à 256, de 4.5 dans le cas (1024 à 512) et de 6 dans le cas (2048 à 1024). Ces résultats sont illustrés par la figure 11.2 qui nous montre que la taille de la matrice de distribution influence d'une façon 'exponentielle' sur le temps d'exécution.
- Au niveau du rapport de temps d'exécution des versions : MDL(MPI) et MDL(SHM) On remarque que le temps d'exécution du code Vlasov de la version de l'implantation MeDLey qui utilise les primitives de communication de la bibliothèque MPI constitue 98 % de celui de la version de l'implantation MeDLey avec la mémoire partagée. Cela est dû essentiellement au faite que la version de *MPI* que nous avons utilisée est une version optimisée de la bibliothèque *MPI* et est implantée en utilisant les primitives de communication par mémoire partagée. Ce qui explique la différence légère au niveau du temps d'exécution de ces deux versions de MeDLey.

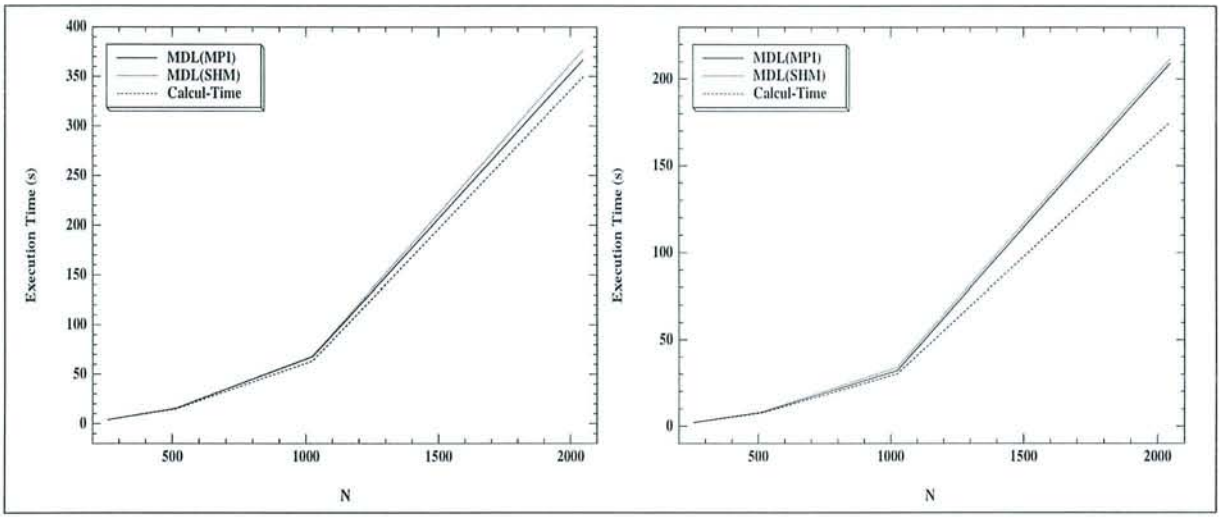

FIG. 11.2 – *Temps d'exécution du code Vlasov en fonction de la taille de la matrice de distribution (N) et de la version de l'implantation MeDLey pour NPROC = 4 (à gauche) et NPROC = 8 (à droite)*

11.4 Conclusion

Le code Vlasov initial était "extensible" (accélération presque linéaire). Avec MeDLey, nous avons pu garder cette caractéristique en obtenant un facteur d'accélération significatif. C'est en fait une preuve de la meilleure utilisation des ressources des processeurs. D'autre part, cette expérimentation nous a montrée la faisabilité de l'approche MeDLey qui peut être facilement utilisée et permet de simplifier notamment les appels des primitives de communication.

En contre partie, cette expérimentation nous a permis de découvrir certaines lacunes de ce nouveau formalisme telle que la version de l'implantation à mémoire partagée qui semble avoir besoin d'être optimisée et corrigée, surtout dans le cas où on utilise un grand nombre de processus.

Chapitre 12

Validation formelle de la bibliothèque *MPC*

La mise en œuvre des techniques formelles pour le développement et la validation des systèmes concurrents, réactifs et distribués est très courant. Les domaines d'application visés concernent les systèmes opératoires, les contrôles de processus industriels et des systèmes plus spécifiques :

- les systèmes embarqués, et en particulier, leur vérification et test garantissant le respect de contraintes temporelles ;
- les systèmes distribués, et en particulier, la validation et la vérification de protocoles.

L'une des principales activités de validation consiste à vérifier qu'une spécification respecte certaines propriétés, en particulier des propriétés comportementales. Dans le cadre d'un système formel, l'énoncé d'une propriété s'exprime et s'interprète sans ambiguïté en fonction de la sémantique associée ; différentes techniques de validation existent ; nous pouvons citer les techniques de preuve formelle, de vérification par modèle et de test (en particulier génération automatique de tests à partir d'une spécification formelle).

L'objectif est de mieux maîtriser la conception, en cherchant d'une part à éviter les erreurs, et d'autre part en assistant la détection et la correction des erreurs qui n'ont pu être évitées.

De nombreuses techniques de modélisation et de spécification existent. Elles peuvent être détaillées de la façon suivante :

- les approches basées sur la logique : théorie des ensembles et substitutions généralisées (B), logiques temporelles (TRIO, TLA, TLA+, TLCO), logiques d'ordre supérieur (Coq,PVS), logique linéaire.
- les approches algébriques : algèbre de processus (CCS, LCS, LOTOS, RTL), approche algébrique pour la modularité (Moka).
- les approches par modèles à états tels que les systèmes de transitions (avec StateCharts, SDL) et les Réseaux de Petri (VAL, RdPTS).
- les approches synchrones, les langages à flots de données synchrones (LUSTRE).

Concernant la bibliothèque *MPC*, notamment son protocole de communication point à point, nous avons choisi comme outil de validation *TLA+* qui est un langage de spécification fondé sur *TLA* (Temporal Logic of Actions) et l'étendant par une plus grande structuration. Nous nous inspirons librement de [Lamport 98] [Lamport 97a] [Lamport 97b].

Dans ce chapitre, nous allons donner une brève introduction sur le langage *TLA+*, puis dans une deuxième partie, nous présenterons la spécification et la validation formelle du protocole de communication point à point de la bibliothèque *MPC* mis en œuvre.

12.1 TLA+

La logique temporelle des actions (*TLA*) est une logique temporelle linéaire, introduite par Lamport [Lamport 98] pour la spécification et la vérification des systèmes réactifs. *TLA* combine une logique des actions et la logique temporelle. De ce fait, elle présente l'avantage de présenter une structure unifiée pour spécifier une application et ses propriétés.

La logique *TLA* peut être considérée comme un langage de spécification. Cependant, il est préférable de séparer l'aspect syntaxique pour la modélisation de l'aspect raisonnement dans la logique. C'est pourquoi, Lamport a introduit *TLA+*. Ce dernier est un langage de spécification ; il supporte la spécification des systèmes par la construction d'un modèle mathématique. Ainsi, un système est spécifié par la description de ses états, un nombre d'actions sur ces états et ses propriétés.

TLA+ est fondé sur *TLA* et utilise des objets mathématiques : structures de données principalement les enregistrements, les ensembles et les fonctions. Les modules, respectivement les définitions, ont été introduits pour structurer les spécifications, respectivement les formules.

12.2 Exemple introductif

Nous considérons l'exemple de la *file* (voir figure 12.1). La spécification de cet exemple en utilisant *TLA+* est présentée dans la figure 12.2 et son fichier de configuration dans la figure 12.3. Nous essaierons via cet exemple d'introduire différents concepts relatifs au langage *TLA+*. Nous n'allons pas définir exhaustivement toute la structure d'une spécification en *TLA+*, mais seulement les aspects qui nous seront les plus utiles.

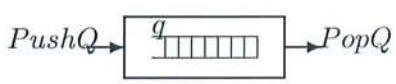

FIG. 12.1 – *File*

12.2.1 Modules, inclusions, déclarations et définitions

Modules

module *File*

En *TLA+*, une spécification est organisée en un ou plusieurs modules. Chaque module est une collection de déclarations, d'inclusions, de définitions, d'hypothèses et de théorèmes.

Inclusions

EXTENDS *Sequences*,*Naturals*

TLA+ permet de construire des spécifications d'une manière compositionnelle (syntaxique) par la construction d'un nouveau module en utilisant les anciens modules. Un module peut utiliser d'autres modules en les incluant comme sous-modules et/ou par l'utilisation des primitives

module *File*

EXTENDS *Sequences, Naturals*

VARIABLE q
CONSTANTS max, $Messages$

$$
\begin{aligned}
&InitQ \triangleq q = \langle\rangle \\
&Next \triangleq [PopQ \vee \exists\, a \in Messages : PushQ(a)]_q \\
&PushQ(a) \triangleq \wedge\ Len(q) < max \\
&\qquad\qquad \wedge\ q' = q \circ \langle a \rangle \\
&PopQ \triangleq \wedge\ q \neq \langle\rangle \\
&\qquad\qquad \wedge\ q' = Tail(q)
\end{aligned}
$$

$$LegalSize \triangleq Len(q) \leq max$$

FIG. 12.2 – *Module File*

CONSTANTS
$max = 100$
$Messages = \{\text{“}data1\text{”},\text{“}data2\text{”}\}$
INIT *InitQ*
NEXT *Next*
INVARIANT *LegalSize*

FIG. 12.3 – *Fichier de configuration du module File*

EXTENDS et INSTANCE. Dans notre exemple, l'utilisation de la primitive EXTENDS permet d'utiliser les opérateurs sur les nombres naturels qui sont définis dans le module *Naturals* ainsi que les opérateurs sur les séquences. Il s'agit en particulier de $\langle\rangle$, pour désigner la séquence vide, $head(q)$, pour désigner le premier élément de la séquence, $tail(q)$, pour désigner la séquence obtenue en supprimant le premier élément, $Len(q)$, pour désigner la langueur de la séquence et $\circ$ pour désigner la composition de deux séquences.

Déclarations

VARIABLE q
CONSTANTS max,$Messages$

Chaque symbole figurant dans un module doit être un opérateur prédéfini dans *TLA+* sinon il doit être déclaré. Dans la partie déclaration, on peut déclarer les variables flexibles sous le mot-clé VARIABLE(S) et les variables rigides sous le mot-clé CONSTANT(S). Les constantes peuvent être des simples constantes ou des opérateurs. Les variables déclarées dans cette partie constituent les seules variables libres du module et peuvent être vues comme des paramètres. Donc elles peuvent être instanciées par d'autres modules.

Dans notre exemple, la variable flexible q, modélisée par une séquence, représente la structure interne de la file. La variable rigide max représente le nombre maximum de messages qu'une file peut contenir tandis que ces messages sont représentés par la variable rigide $Messages$.

Définitions

$$InitQ \triangleq q = \langle\rangle$$

Le *prédicat d'état INIT* dans le fichier de configuration décrit l'état initial. Il est l'équivalent de l'état *InitQ* dans le module file qui stipule par l'utilsation du symbole $\triangleq$ (l'équivalence sémantique) que la valeur q est égale à une séquence vide. Syntaxiquement, le prédicat d'état est une formule de la logique des prédicats qui peut contenir des variables flexibles. Sémantiquement, le prédicat d'état décrit un ensemble d'états.

$$\begin{aligned} PushQ(a) &\triangleq \wedge\ Len(q) \leq max \\ &\quad\ \wedge\ q' = q \circ \langle a \rangle \\ PopQ &\triangleq \wedge\ q \neq \langle\rangle \\ &\quad\ \wedge\ q' = Tail(q) \end{aligned}$$

La formule *NEXT* est une action. Dans notre exemple, cette action est soit *PushQ* ou *PopQ* sur la file initialisée dans *INIT*. Syntaxiquement, les actions sont des formules de la logique des prédicats. Sémantiquement, une action décrit une relation entre un ensemble d'états. Par exemple la définition $PushQ(a)$ introduit un opérateur *PushQ* paramétré par le nouveau symbole a. L'expression associée à cet opérateur est une action qui utilise les opérateurs *Len* (longueur d'une séquence) et $\circ$ (composition de deux séquences) définis dans le module *Sequences* et l'opérateur $\leq$ défini dans le module *Naturals*. Intuitivement, l'opérateur *PushQ* permet d'ajouter un élément à la file q si la longueur de cette dernière est inférieure à la limite max. q' est la nouvelle valeur de la séquence q et $\wedge$ est l'opérateur de conjonction.

La propriété d'invariance

En *TLA+*, il n'y a pas de distinction entre une spécification et une propriété : toutes les deux sont des formules *TLA+*.

La logique temporelle de TLA est fondée sur des traces, c'est-à-dire des suites d'états. Une trace représente un comportement ou une exécution. Un programme est représenté par l'ensemble de ses exécutions possibles.

Dès lors que nous avons une suite linéaire d'états, nous pouvons définir une logique temporelle linéaire sur ce support. Lors du passage d'un état à son état successeur, il faut souvent vérifier si certaines propriétés sont respectées. La propriété d'un système qui nous intéressent le plus souvent est la propriété d'invariance. L'invariance est une propriété de sûreté. Généralement, prouver une propriété de sûreté revient à prouver que quelque chose de mauvais n'arrivera pas. Un comportement mauvais est un comportement anormal.

$$LegalSize \triangleq Len(q) \leq max$$

Dans l'exemple de la file, cette propriété s'exprime en utilisant la formule *INVARIANT* dans le fichier de configuration. Cette propriété vérifie si la formule *LegalSize* est toujours vraie. C'est-à-dire que sa correction est un théorème. La formule *LegalSize* vérifie, après la fin de chaque état si la taille de la file est toujours inférieure ou égale à la constance max.

12.3 Spécification du protocole de communication point à point de la bibliothèque *MPC* avec *TLA+*

Dans cette section, nous allons présenter les différents composants de la spécification du protocole de communication point à point de la bibliothèque *MPC*.

Comme nous avons vu dans l'exemple introductif, un système, programme ou algorithme est spécifié en *TLA+* en donnant tous les comportements autorisés de ce système : l'état initial puis les étapes successeurs.

Nous commençons notre spécification par la description de son entête (voir figure 12.4) qui contient la partie inclusion et les déclarations puis son fichier de configuration (voir figure 12.5).

module *MPC*

EXTENDS *Sequences, Naturals*

VARIABLES *recvBuffer, recvDescBuffer, synchronizeVar, recvBufferMonitor, recvDescBufferMonitor, synchronizeVarMonitor, addRecvBufferStep1Var, addRecvBufferStep2Var, addRecvDescBufferStep1Var, addRecvDescBufferStep2Var, waitSynchronizeVar, waitMonitorVar*

CONSTANTS *NPROCS, TAG, DATA*

$Proc \triangleq 1..NPROCS$

$Message \triangleq [emt : Proc, tag : TAG, data : DATA, isSendSync : BOOLEAN]$

$DescMessage \triangleq [emt : Proc, tag : TAG, isRecvSync : BOOLEAN]$

FIG. 12.4 – *L'entête du Module MPC*

CONSTANTS
$NPROCS = 3$
$DATA = \{\text{“}d1\text{”}, \text{“}d2\text{”}\}$
$TAG = \{1,2,3\}$
INIT
InitMPC
NEXT
NextMPC
INVARIANT
NoDeadlock

FIG. 12.5 – *Fichier de configuration du module MPC*

Dans notre exemple, l'utilisation de la primitive EXTENDS permettra d'utiliser les opérateurs sur les nombres naturels qui sont définis dans le module Naturals ainsi que les opérateurs sur les séquences. Dans la partie VARIABLES, nous avons déclarés l'ensemble des variables à utiliser dans notre spécification. Il s'agit de celles utilisées dans la spécification du protocole (voir chapitre 10) :

- $recvBuffer$: c'est le buffer de réception, il contient l'ensemble des messages envoyés par les autres tâches et dont la primitive de réception n'est pas encore appelée ;

- $recvDescBuffer$: le buffer de descripteurs des messages à recevoir : il contient la description des différents messages attendus ;
- $synchronizeVar$: c'est la variable de synchronisation, elle est utilisée pour assurer la synchronisation dans certains modes de communication ;
- les moniteurs associés à ces trois variables : $recvBufferMonitor$, $recvDescBufferMonitor$ et $synchronizeVarMonitor$.

Les autres variables utilisées seront décrites au moment de leur utilisation.

Comme constantes, nous utilisons $NPROCS$ qui est le nombre de processus (i.e. de tâches), TAG qui contient les valeurs des étiquettes utilisées lors de l'envoi ou la réception d'un message (elles varient de 1 à 3 dans l'exemple), et enfin les données à échanger représentées par la constante $DATA$ qui peut prendre comme valeurs possibles les deux chaînes de caractères « d1 » ou « d2 ». Rappelons dans ce cadre que le contenu des messages est facultatif dans notre spécification. C'est pour cette raison que le type de données échangé ne sera pas décrit dans notre description car il s'agit d'un détail d'implantation et non pas de spécification.

Ensuite, nous définissons la variable $Proc$ qui est le numéro de processus (varie de 1 à NPROCS), puis les deux structures de données qui correspondent respectivement à un message $Message$ puis à un descripteur de message $DescMessage$.

12.3.1 Définitions

Initialisation

$$
\begin{aligned}
InitMPC \triangleq\ & \wedge\ recvBuffer \triangleq [p \in Proc \mapsto \langle\rangle] \\
& \wedge\ recvBuffer \triangleq [p \in Proc \mapsto \langle\rangle] \\
& \wedge\ synchronizeVar \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ recvBufferMonitor \triangleq [p \in Proc \mapsto TRUE] \\
& \wedge\ recvDescBufferMonitor \triangleq [p \in Proc \mapsto TRUE] \\
& \wedge\ synchronizeVarMonitor \triangleq [p \in Proc \mapsto TRUE] \\
& \wedge\ addRecvBufferStep1Var \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ addRecvBufferStep2Var \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ addRecvDescBufferStep1Var \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ addRecvDescBufferStep2Var \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ waitSynchronizeVar \triangleq [p \in Proc \mapsto FALSE] \\
& \wedge\ waitMonitorVar \triangleq [p \in Proc \mapsto FALSE]
\end{aligned}
$$

Le *prédicat d'état INIT* dans le fichier de configuration décrit l'état initial. Il est l'équivalent de l'état $InitMPC$ dans le module MPC. Ce prédicat initialise les différentes variables utilisées. On remarque que chaque processus à ses propres variables. Par exemple :

$$recvBuffer \triangleq [p \in Proc \mapsto \langle\rangle]$$

Associe à chaque processus une variable recvBuffer initialisée à la séquence vide ($\langle\rangle$). A chaque moniteur, on associe une variable booléenne initialisée à la valeur « TRUE » pour indiquer que

la variable associée à ce moniteur est en libre accès. En outre, la variable $waitMonitorVar$, initialisée à « FALSE », indique si un processus et en attente d'accéder à une variable (en disposant de son moniteur). Enfin, la variable $waitSynchronizeVar$ permet de savoir si un processus est en attente d'une synchronisation.

Les autres prédicats

Comme nous avons vu pendant la description du protocole de communication point à point de la bibliothèque MPC, la méthode $addRecvBuffer$ verrouille l'accès aux moniteurs du buffer de réception et du buffer de descripteurs des messages à recevoir avant de commencer le traitement associé à cette méthode. Chacune de ces étapes est une étape indépendante qu'il faut spécifier comme une action avec $TLA+$. Par conséquent, la méthode *addRecvBuffer* est spécifiée en utilisant 3 actions :

- $addRecvBufferStep1$: il s'agit de l'action qui verrouille l'accès au moniteur associé au buffer de réception ;
- $addRecvBufferStep2$: c'est l'action qui permet de verrouiller l'accès au moniteur associé au buffer des descripteurs des messages à recevoir.
- $addRecvBuffer$: il s'agit de l'action qui exécute le traitement associé à la méthode addRecvBuffer.

Il est évident que lorsqu'un un processus finisse l'action $addRecvBufferStep1$, il doit passer directement à l'action $addRecvBufferStep2$ pendant les prochaines étapes. Pour y parvenir, il doit disposer d'une information qui lui permet de déterminer si l'action $addRecvBufferStep1$ a été déjà exécutée. C'est le rôle de la variable $addRecvBufferStep1Var$ qui est de type booléen. Du même, la variable $addRecvBufferStep2Var$ permet d'indiquer si un processus a déjà exécuté l'action $addRecvBufferStep2$. Ces deux variables sont initialisées à la valeur « FALSE ».

La spécification du prédicat $addRecvBufferStep1$ est la suivante :

$$
\begin{array}{l}
addRecvBufferStep1(p) \triangleq \\
\textbf{if } recvBufferMonitor[p] = TRUE \\
\quad \textbf{then } \wedge\ recvBufferMonitor' = [recvBufferMonitor \text{ EXCEPT } ![p] = FALSE] \\
\qquad \wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE] \\
\qquad \wedge\ addRecvBufferStep1Var' = [addRecvBufferStep1Var \text{ EXCEPT } \\
\qquad\qquad\qquad\qquad\qquad\qquad ![p] = TRUE] \\
\qquad \wedge \text{ UNCHANGED } \langle recvBuffer, recvDescBuffer, synchronizeVar, \\
\qquad recvDescBufferMonitor, synchronizeVarMonitor, \\
\qquad addRecvBufferStep2Var, addRecvDescBufferStep1Var, \\
\qquad addRecvDescBufferStep2Var, waitSynchronizeVar\rangle \\
\quad \textbf{else } \wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = TRUE] \\
\qquad \wedge \text{ UNCHANGED } \langle recvBuffer, recvBufferMonitor, recvDescBuffer, \\
\qquad recvDescBufferMonitor, synchronizeVar, synchronizeVarMonitor, \\
\qquad addRecvBufferStep2Var, addRecvBufferStep1Var, \\
\qquad addRecvDescBufferStep1Var, addRecvDescBufferStep2Var, waitSynchronizeVar\rangle
\end{array}
$$

Ce prédicat vérifie d'abord si la variable $recvBuffer$ est en libre accès puis y verrouille l'accès en utilisant son moniteur. Le verrouillage est effectué en changeant la valeur de la variable

recvBufferMoniteur du processus qui a fait l'appel à la valeur « FALSE » par l'utilisation de l'instruction suivante :

$$recvBufferMonitor' = [recvBufferMonitor \text{ EXCEPT } ![p] = FALSE]$$

Cette instruction indique que la nouvelle valeur de $recvBufferMonitor$ ($recvBufferMonitor'$) sera celle de l'ancienne valeur sauf pour le processus p où elle prendra comme valeur « FALSE ».

Le prédicat UNCHANGED permet de designer les variables qui ne changeront pas de valeurs au cours de l'action courante.

La spécification du prédicat $addRecvBufferStep2$ suit les mêmes règles à la différence qu'il faut utiliser le moniteur du buffer de descripteurs de messages à recevoir au lieu de celui de réception :

$addRecvBufferStep2(p) \triangleq$
if $recvDescBufferMonitor[p] = TRUE$
then $\wedge\ recvDescBufferMonitor' = [recvBufferMonitor \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ addRecvBufferStep2Var' = [addRecvBufferStep2Var \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvBuffer, recvDescBuffer, synchronizeVar, recvBufferMonitor, synchronizeVarMonitor, addRecvBufferStep1Var, addRecvDescBufferStep1Var, addRecvDescBufferStep2Var, waitSynchronizeVar\rangle$
else $\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvBuffer, recvBufferMonitor, recvDescBuffer, recvDescBufferMonitor, synchronizeVar, synchronizeVarMonitor, addRecvBufferStep2Var, addRecvBufferStep1Var, addRecvDescBufferStep1Var, addRecvDescBufferStep2Var, waitSynchronizeVar\rangle$

La spécification complète de la méthode $addRecvBuffer$ est par conséquent la suivante :

```
addRecvBuffer(p,m) ≜
if ¬addRecvBufferStep1Var[p]
  then addRecvBufferStep1(p)
  else if ¬addRecvBufferStep2Var[p]
         then addRecvBufferStep2(p)
         else (∨ (∃ k ∈ 1..Len(recvDescBuffer[p]) :
                  ∧ m.emt = recvDescBuffer[p][k].emt
                  ∧ m.tag = recvDescBuffer[p][k].tag
                  ∧ if (recvDescBuffer[p][k].isRecvSync ∧ m.isSendSync)
                      then ∧ synchronizeVar' = [synchronizeVar
                                  EXCEPT ![p] = TRUE,![m.emt] = TRUE]
                           ∧ waitSynchronizeVar' = [waitSynchronizeVar
                                  EXCEPT ![p] = FALSE,![m.emt] = FALSE]
                      else if recvDescBuffer[p][k].isRecvSync
                             then synchronize(p)
                             else if m.isSendSync
                                    then synchronize(m.emt)
                                    else UNCHANGED ⟨synchronizeVar,waitMonitorVar,
                                                    waitSynchronizeVar⟩
                  ∧ recvDescBuffer' = [recvDescBuffer
                      EXCEPT ![p] = [j ∈ 1..(Len(recvDescBuffer[p]) − 1) ↦
                                      if j < k then @[j]
                                               else @[j + 1]]]
                  ∧ UNCHANGED ⟨recvBuffer⟩)
                ∨ (∀ k ∈ 1..Len(recvDescBuffer[p]) :
                  ∧ ¬(m.emt = recvDescBuffer[p][k].emt∧
                        m.tag = recvDescBuffer[p][k].tag))
                  ∧ recvBuffer' = [recvBuffer EXCEPT ![p] = Append(@,m)]
                  ∧ UNCHANGED ⟨recvDescBuffer,synchronizeVar,waitSynchronizeVar,
                        waitMonitorVar⟩))
                ∧ recvBufferMonitor' = [recvBufferMonitor
                                          EXCEPT ![p] = TRUE]
                ∧ recvDescBufferMonitor' = [recvDescBufferMonitor
                                          EXCEPT ![p] = TRUE]
                ∧ addRecvBufferStep1Var' = [addRecvBufferStep1Var
                                          EXCEPT ![p] = FALSE]
                ∧ addRecvBufferStep2Var' = [addRecvBufferStep2Var
                                          EXCEPT ![p] = FALSE]
                ∧ UNCHANGED ⟨synchronizeVarMonitor,
                  addRecvDescBufferStep1Var,addRecvDescBufferStep2Var⟩
```

Dans le prédicat *addResvBuffer(p, m)* p est le processus récepteur et m le message à ajouter (i.e envoyer) dans le buffer de réception. Ce prédicat appelle celui de *addRescBufferStep1* dans le cas où ce dernier n'est pas encore exécuté, sinon, il appelle directement *addRescBufferStep2*. Dans le cas où *addRescDescBufferStep2* est déjà exécuté, le prédicat *addRecvBuffer* commence directement le traitement associé à cette méthode. D'abord, elle cherche si le descripteur de

message à ajouter dans le buffer de réception existe dans le buffer de descripteurs de messages à recevoir en utilisant les instructions suivantes :

$$\begin{array}{l}\exists k \in 1..Len(recvDescBuffer[p]) : \\ \quad \wedge\ m.emt = recvDescBuffer[p][k].emt \\ \quad \wedge\ m.tag = recvDescBuffer[p][k].tag\end{array}$$

Si un tel descripteur existe, en fonction des modes d'envoi et de réception, les variables $synchronizeVar$ et $waitSynchronizeVar$ seront modifiées en appelant le prédicat *synchronize* :

$$\begin{array}{l}synchronize(p) \triangleq \\ \textbf{if}\ synchronizeVarMonitor[p] = TRUE \\ \quad \textbf{then} \wedge\ synchronizeVar' = [synchronizeVar\ \text{EXCEPT}\ ![p] = TRUE] \\ \qquad \wedge\ waitSynchronizeVar' = [waitSynchronizeVar\ \text{EXCEPT}\ ![p] = FALSE] \\ \qquad \wedge\ waitMonitorVar' = [waitMonitorVar\ \text{EXCEPT}\ ![p] = FALSE] \\ \quad \textbf{else}\ \ waitMonitorVar' = [waitMonitorVar\ \text{EXCEPT}\ ![p] = TRUE] \\ \qquad \text{UNCHANGED}\ \langle synchronizeVar, waitSynchronizeVar\rangle\end{array}$$

Enfin, le descripteur du message est effacé du buffer de descripteurs de messages par l'utilisation des instructions suivantes :

$$\begin{array}{l}recvDescBuffer' = [recvDescBuffer\ \text{EXCEPT}\ ![p] = \\ \qquad [j \in 1..(Len(recvDescBuffer[p]) - 1) \mapsto \\ \qquad\qquad \textbf{if}\ j < k\ \textbf{then}\ @[j] \\ \qquad\qquad\qquad \textbf{else}\ \ @[j+1]]]\end{array}$$

Dans le cas où le descripteur de message à ajouter n'existe pas dans le buffer de descripteurs de messages, ce message sera ajouté directement au buffer de réception par l'utilisation du prédicat *append* :

$$recvBuffer' = [recvBuffer\ \text{EXCEPT}\ ![p] = Append(@,m)]$$

Une fois le prédicat *addRecvBuffer* est définie, on pourrait donc présenter les deux modes d'envoi, nous commençons par l'envoi asynchrone MPC_Asend. Ce type d'envoi se présente comme suit :

$$MPC\_Asend(p,m) \triangleq addRecvBuffer(p,m)$$

Pour envoyer un message m asynchrone à un processus p, le predicat MPC_Asend appelle celui de *addRecvBuffer(p,m)*.

Passons maintenant à l'envoi synchrone. Ce mode d'envoi est spécifié comme suit :

$$\begin{array}{l}MPC\_Ssend(p,m) \triangleq \\ \qquad \wedge\ addRecvBuffer(p,m) \\ \qquad \wedge\ \textbf{if}\ (addRecvBufferStep1Var[p] \wedge addRecvBufferStep2Var[p]) \\ \qquad\quad \textbf{then}\ waitSynchronize(m.emt)\end{array}$$

Le prédicat d'envoi synchrone appelle celui de $addRecvBuffer(p,m)$ puis met l'émetteur en attente de synchronisation en utilisant le prédicat $waitSynchronize$. Remarquons que l'appel au prédicat $waitSynchronize$ ne se fait que lorsque le prédicat $addRecvBuffer$ est complément achevé (i.e. Les deux prédicats $addRcvBufferStep1$ et $addRecvBufferStep2$ sont exécutés).

Le prédicat waitSynchronize se présente comme suit :

$$
\begin{array}{l}
waitSynchronize(p) \triangleq \\
\textbf{if } synchronizeVarMonitor[p] = TRUE \\
\quad \textbf{then if } synchronizeVar[p] = FALSE \\
\qquad \textbf{then } \wedge\ waitSynchronizeVar' = [waitSynchronizeVar \text{ EXCEPT } ![p] = TRUE] \\
\qquad\qquad \wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE] \\
\qquad\qquad \wedge \text{ UNCHANGED } \langle synchronizeVar \rangle \\
\qquad \textbf{else } \wedge\ synchronizeVar' = [synchronizeVar \text{ EXCEPT } ![p] = FALSE] \\
\qquad\qquad \wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE] \\
\qquad\qquad \wedge\ waitSynchronizeVar' = [waitSynchronizeVar \text{ EXCEPT } ![p] = FALSE] \\
\qquad \textbf{else } \wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = TRUE] \\
\qquad\qquad \wedge \text{ UNCHANGED } \langle synchronizeVar, waitSynchronizeVar \rangle
\end{array}
$$

Ce prédicat met le processus p en attente de synchronisation si la valeur de synchronizeVar est à « FALSE », sinon il remet la valeur de cette variable à « FALSE ».

Jusqu'à présent, nous avons présenté les prédicats utilisés pour les deux modes mis en œuvre dans la bibliothèque MPC. La spécification, avec $TLA+$ des deux modes de réception suit les mêmes règles comme décrit dans le protocole de communication (voir chapitre 10). La spécification complète du protocole de communication point à point de la la bibliothèque MPC se trouve en annexe E.

Nous présentons dans la suite la formule $NEXT$ qui se présente comme l'action à exécuter pour atteindre l'état successeur. Dans notre protocole, cette action est soit un envoi ou une réception et se présente comme suit :

$$
\begin{array}{l}
Next \triangleq \exists p \in Proc : \\
\quad \vee\ (\exists m \in Message : \neg m.isSendSync \wedge \neg waitSynchronizeVar[m.emt] \wedge MPC\_Asend(p,m)) \\
\quad \vee\ (\exists m \in Message : m.isSendSync \wedge \neg waitSynchronizeVar[m.emt] \wedge MPC\_Ssend(p,m)) \\
\quad \vee\ (\exists d \in DescMessage : \neg d.isRecvSync \wedge \neg waitSynchronizeVar[p] \wedge MPC\_Arecv(p,d)) \\
\quad \vee\ (\exists d \in DescMessage : d.isRecvSync \wedge \neg waitSynchronizeVar[p] \wedge MPC\_Arecv(p,d))
\end{array}
$$

Remarquons dans ce prédicat qu'un processus ne peut émettre ou recevoir que lorsqu'il n'est pas en attente d'une synchronisation.

12.3.2 La propriété d'invariance

Dès lors que la spécification de notre protocole est faite. Nous devons déduire certaines de ses propriétés notamment celle de l'invariance pour le valider et vérifier sa correction.

Comme nous avons vu au niveau de l'exemple introductif, la propriété d'invariance s'exprime en utilisant la formule $INVARIANT$ dans le fichier de configuration qui est l'equivalent de $NoDeadlock$ dans le fichier de spécification en $TLA+$ de notre protocole.

La propriété qui nous intéresse est le manque d'inter-blocage entre les différents processus (i.e. où tous les processus sont inactifs en attente d'un moniteur). Nous exprimons cette propriété

comme suit :

$$NoDeadlock \triangleq \neg(\forall p \in Proc : waitMonitorVar[p])$$

Respecter cette propriété revient à vérifier qu'il n'y a pas un cas où tous les processus sont en attente d'un moniteur (i.e. pas en attente d'une synchronisation). Cette propriété peut facilement être vérifiée dans le cas où l'application contient deux processus.

A remarquer qu'on peut avoir des cas où tous les processus sont inactifs sans avoir un interblocage. Par exemple si tous les processus appellent, en même temps, une primitive de réception synchrone. Ce cas n'implique pas une erreur au niveau du protocole mais au niveau de l'application elle même.

On peut également vérifier des autres propriétés d'invariance mais pour l'instant, puisqu'il s'agit encore d'un protocole simple, nous nous contentons de cette propriété.

12.4 Conclusion

La validation formelle des spécifications joue un rôle crucial dans le développement des protocoles de communication. En effet, elle permet de réduire les erreurs de modélisation avant que le processus de développement soit entamé (raisonnement et implantation). C'est pourquoi, la correction d'un système s'appuie pleinement sur la complémentarité de la validation et de la vérification.

Dans ce chapitre, nous avons validé le protocole de communication point à point de la bibliothèque *MPC*. Dans ce cadre, nous avons utilisé *TLA+* qui est un langage de spécification fondé sur *TLA* (Temporal Logic of Actions). Nous avons spécifié quelques modes de communication utilisés dans ce protocole. Une extension de cette spécification est en cours de réalisation pour les protocoles de communication collectives mis en œuvre dans la bibliothèque *MPC*.

Sixième partie

Conclusion

Conclusion et perspectives

Les communications apparaissent comme un point crucial lors du développement des applications parallèles. Ainsi, devant la richesse de ce domaine en terme d'architecture et de modèles de programmation et la diversité des besoins des utilisateurs, plusieurs outils de communication ont vu le jour. Cependant, les utilisateurs du parallélisme qui sont le plus souvent des physiciens, mathématiciens ou chimiste ne sont pas prêts à investir dans ces différents modèles de communication. D'où le besoin de développement d'environnements et d'outils qui combinent l'efficacité, l'évolutivité et surtout la facilité de l'utilisation des outils de communication développés.

C'est dans ce cadre que se situe l'un des axes de recherche de l'équipe RESEDAS qui vise à fournir un environnement d'aide au développement des applications parallèles. Le langage `MeDLey` est le composant permettant la spécification des communications pour ce type d'application. Son principe repose sur l'implantation des communications sur différentes plates-forme de communication en utilisant un langage et une syntaxe unifiés.

L'apport original de la thèse a été guidé par le besoin de faire évoluer ce langage pour couvrir un champ d'application plus vaste et de l'expérimenter sur des applications réelles. Nous avons focalisé nos recherches sur les architectures parallèles à mémoire distribuée, en particulier celles composées d'un réseau de stations.

Fondée sur une analyse aussi large que possible de l'état de l'art en systèmes de communications parallèles et distribués, la contribution de cette thèse se situe à trois niveaux :

1. D'une part, nous avons étendu la version initiale du langage `MeDLey`. Une première extension a concerné les communications collectives ; nous avons proposé une nouvelle approche et des nouvelles fonctions permettant de spécifier et réaliser ce type de communication. La deuxième extension a concerné les méthodes de décomposition de domaine ; nous avons définit un nouveau mode d'interconnexion, puis proposé des fonctions permettant de couvrir les besoins pour appliquer ces méthodes.
2. D'autre part, si les premières extensions concernaient le langage `MeDLey` lui même, la deuxième extension avait pour objectif l'implantation de ce langage dans un environnement *CORBA*. L'idée de l'utilisation de cette architecture pour le calcul parallèle était un peu novatrice et il fallait par conséquent justifier ce choix, surtout que *CORBA* a toujours été critiqué au niveau des performances. En tenant compte de tous ces éléments, une évaluation de performances s'avérait nécessaire comme première étape de notre approche. Pour cette raison, nous avons développé un ensemble de tests de performances des deux bus *CORBA* ORBacus et TAO et nous les avons comparés à ceux en utilisant les bibliothèques d'échange de messages *MPI* (MPICH et LAM) et *PVM*. Comme conclusion pour l'ensemble de ces tests, nous avons remarqué que les implantations de *CORBA* étaient compétitives, voir même meilleures, au point de vue performances que les bibliothèques de communication par échange de messages.

   Ces résultats nous ont permis de progresser dans notre démarche initiale concernant l'im-

plantions du lange MeDLey dans un environnement *CORBA*. Dans ce cadre, nous avons développé une première approche de cette implantation. Si cette approche reste faisable, nous avons pu remarquer lors de sa mise en pratique qu'elle était lourde et difficile à utiliser. Pour remédier à ce problème, nous avons développé une bibliothèque de communication par échange de messages au dessus du *CORBA* appelée *MPC* (Message Passing in Corba environment). Nous avons présenté les différents protocoles de communication que nous avons développés pour sa mise en œuvre. Cependant, si la motivation initiale qui nous a conduit à développer cette bibliothèque est l'implantation du langage MeDLey dans un environnement *CORBA* ; nous avons aussi voulu qu'elle puisse être également utilisée, comme toute autre bibliothèque de communication par échange de messages, indépendamment du langage MeDLey. Pour parvenir à ce but, nous avons développé une interface de communication de cette bibliothèque qui lui permet d'être considérée comme un environnement complet pour la programmation par échange de messages dans un environnement *CORBA*. Les premiers résultats expérimentaux de la première implantation de cette bibliothèque étaient encourageants et nous ont montrés la faisabilité de cette bibliothèque.

3. Enfin, nos derniers travaux ont portés sur l'expérimentation et la validation des outils développés. Dans ce contexte, une expérimentation du langage MeDLey a été effectuée dans le cadre d'une collaboration avec le laboratoire de recherche en physique (LPMI) de l'UHP, et a consisté à l'implantation d'un code de simulation numérique utilisé par les physiciens. Cette expérience a fourni des résultats utiles au niveau du modèle physique et nous a montré la faisabilité et l'intérêt des méthodes et outils proposés au niveau de ce langage. En particulier, elle nous a permis de détecter quelques lacunes de ce formalisme. Ensuite, nous avons validé le protocole de communication point à point de la bibliothèque *MPC* en utilisant *TLA+* qui est un langage de spécification fondé sur (Temporal Logic of Actions). Cette validation nous a permis d'affirmer la correction du protocole de communication mis en œuvre.

Comme perspectives, les propositions faites au niveau de cette thèse constituent un point de départ pour des extensions futures. En effet, nos travaux autours de *CORBA* peuvent être considérés parmi les premiers dans ce domaine. Par conséquent, le challenge est maintenant de réussir à convaincre les utilisateurs de l'intérêt de ces travaux afin d'établir un dialogue permanent qui permettra certainement leur évolution. Cela nécessite d'abord de compléter le prototype de la bibliothèque *MPC* développé en fournissant une interface de communication plus riche et en proposant des autres alternatives pour les protocoles de communication mis en oeuvre. Ensuite, des tests de performances plus approfondis de cette bibliothèque semblent nécessaires avant de lancer le processus de son expérimentation et son utilisation à grande échelle.

Septième partie

Annexes

Annexe A

Code MonteCarlo en C++ utilisant MeDLey

Cette partie contient les codes sources en C++ des deux types de tâches définies dans l'exemple du calcul de π par la méthode de MonteCarlo avec MeDLey.

A.1 Le code de la tâche superviseur *father*

Nous présentons tout d'abord le code C++ de la tâche *superviseur*, correspondant au module MeDLeyfather.

Dans ce code, nous pouvons tout d'abord remarquer à la ligne 14 l'inclusion du fichier généré par le générateur automatique MeDLey. Ce fichier contient les définitions correspondant au module *father*.

Ensuite l'utilisation du code MeDLey se traduit à la ligne 26 par l'instanciation d'un objet *father*, et à l'appel sur cet objet de la méthode MDL_Init à la ligne 29. Cette initialisation permet à l'ensemble des instances de tâches MeDLey de s'identifier.

Ensuite, l'algorithme commence. Nous pouvons remarquer l'utilisation de variables définies dans le module MeDLey aux lignes 39 et 40. Ces variables sont des variables d'instance de l'objet.

Les appels aux fonctions d'envoi et de réception de messages se trouvent aux lignes 45, 47, et 57. Dans l'exemple de l'appel de la ligne 57, nous pouvons remarquer que la réception a pour effet la mise jour des variables *t.in* et *t.who* qui sont utilisées aux lignes suivantes.

Enfin, une fois le calcul terminé, un appel à la méthode MDL_End est effectué sur l'objet *t*. Cet appel libère les différentes structures utilisées avant de quitter le programme.

```
1  //
2  //        File father.C
3  //
4
5  #include  <stdlib.h>     // drand48, srand48
6  #include  <math.h>       // sqrt,, M_PI
7  #include  <limits.h>     // INT_MAX, ULONG_MAX
8  #include  <iostream.h>   // cin, cout
9  #include  <iomanip.h>    // setw, setprecision
10 #include  <stdio.h>
11
12 const int    BATCH = 20480;
```

```
13
14 #include "fatherMDL.hh"
15
16 int main(int argc, char **argv)
17 {
18
19   // Some data not involved in communications
20   unsigned long int MINIMUM, IN = 0, TOTAL =0;
21   int sonNumber;
22   int i;
23   int taskSharing[20];
24
25   //
26   father t;
27
28   // the initialization
29   t.MDL_Init(&argc, &argv);
30   sonNumber = t.MDL_sonSize();
31
32   for(i=0;i<sonNumber;i++)
33     taskSharing[i] = 0;
34
35   cout << "\n Enter the minimum number of shots:    (<" << ULONG_MAX
36        << ")  " << flush;
37   cin >> MINIMUM;
38
39   t.number = MINIMUM;
40   t.who = 0;
41
42   cout << "Hello : " << sonNumber << endl;
43
44   // receives box and coeff
45   t.MDL_RecvFrom_son_startMessage();
46
47   t.MDL_SendTo_son_shotMessage();
48
49   // the computation
50
51   do
52     {
53       cout << " left:    " << MINIMUM-TOTAL << "    i.e.   "
54            << 100*(MINIMUM-TOTAL)/MINIMUM << "%  \r" << flush;
55
56       // let's receive a shot
57       t.MDL_RecvFrom_son_inMessage();
58
59       IN += t.in;
60       TOTAL += BATCH;
```

```
61
62          ++taskSharing[t.who];
63        } while (TOTAL<MINIMUM);
64
65      double I = t.box * double(IN)/TOTAL;
66
67      const double  PI = 3.14159265358979e+00;
68      double        pi = I * t.coeff;
69
70      cout.setf( ios::scientific );    //.......................... Displaying
71      cout << setprecision(14)
72           <<   " number of shots : " << TOTAL << "  "
73           << "\n lirary's pi       = " << PI
74           << "\n computed pi       = " << pi
75           << "\n absolute error    : " << fabs(PI-pi) << endl;
76
77      cout << "\n Slaves' respective task sharing:"
78        "\n  Inst      Batch" << endl;
79      for(i=0 ; i<sonNumber ; i++ )
80        {
81          cout << setw(6) << i
82               << setw(8) << taskSharing[i]
83               << "   i.e.  " << 100*taskSharing[i]*BATCH/TOTAL << "%" << endl;
84        }
85
86      // The end !!
87      t.MDL_End();
88
89      return 1;
90  }
```

A.2 Le code des tâches de calcul *son*

Cette section présente le code des tâches *son*. Comme dans le paragraphe précédent, tout commence par l'inclusion du fichier généré par l'outil `MeDLey` à la ligne 13. Ensuite vient l'initialisation de l'objet après son instanciation (aux lignes 39 et 45), puis les appels aux fonctions de communication (lignes 54, 57, 69).

Vient enfin l'appel final à la méthode MDL_End avant la fin du code. Ici encore on peut remarquer l'utilisation des variables déclarées dans le module `MeDLey` qui sont implantées comme des variables d'instance de l'objet *son*.

```
1  //
2  //        File son.C
3  //
4
5  #include  <stdlib.h>      // drand48, srand48
6  #include  <math.h>        // sqrt,, M_PI
7  #include  <limits.h>      // INT_MAX, ULONG_MAX
```

```
8  #include  <iostream.h>  // cin, cout
9  #include  <iomanip.h>   // setw, setprecision
10 #include  <stdio.h>
11 #include  <math.h>
12
13 #include "sonMDL.hh"
14
15 #define TRUE 1
16 #define FALSE 0
17
18 #if !defined  RAND_MAX
19 #    define   RAND_MAX  INT_MAX    // pour g++
20 #endif
21
22 const int    BATCH = 20480;
23
24 double    a=2,   b=3,  c=5,   box = 8*a*b*c,    coeff = (double)3/4 /(a*b*c);
25
26 // tire un point et determine s'il appartient au domaine:
27 inline  int  shot()
28 {
29   double  X  =  (double)rand()/RAND_MAX  *2*a     - a,
30     Y  =  (double)rand()/RAND_MAX  *2*b     - b,
31     Z  =  (double)rand()/RAND_MAX  *2*c     - c,
32     rr =  ( X/a )*( X/a ) + ( Y/b )*( Y/b ) + ( Z/c )*( Z/c );
33   return(  (rr>1) ?  FALSE:TRUE  );
34 }
35
36 int main(int argc, char **argv)
37 {
38   // the son representation
39   son t;
40
41   // Some data not involved in communications
42   int i;
43
44   // the initialization
45   t.MDL_Init(&argc, &argv);
46
47   t.box = 8*a*b*c;
48   t.coeff = (double)3/4 /(a*b*c);
49
50   t.who = t.MDL_MyRank();
51
52   if (t.who == 0)
53     {
54       t.MDL_SendTo_father_startMsg();
55     }
```

```
56
57    t.MDL_RecvFrom_father_numberOfShot();
58
59    int total = 0;
60    //  tfprintf(tstdout, "son(%d)=%d\n", t.who, t.shotNumber);
61    srand(t.who);
62
63    do
64      {
65        t.in = 0;
66        for (i=0; i<BATCH; t.in += shot(), i++);
67
68        // Send one shot to father
69        t.MDL_SendTo_father_oneShot();
70
71  //        if (t.who == 3)
72  //          tfprintf(tstdout, "%d\n", total);
73        total += BATCH;
74      } while (total < t.shotNumber);
75
76    // the end !
77    t.MDL_End();
78
79    return 1;
80  }
```

Annexe B

Les fonctions de l'extension de MeDLey en communication collective

B.1 Opérations collectives de transfert de données

Bcast : Permet de diffuser un message d'une instance source sur l'ensemble des instances d'une tâche.

Syntaxe :

```
Bcast(root) // root : le rang de l'instance contenant les données
            //        à diffuser, par défaut = 0
        {
            m = sequence{Send_buf, Recv_buf, count};
        }
```

Le contenu du message à diffuser peut être défini statiquement (avant l'exécution) ou dynamiquement (au moment de l'exécution). Ainsi, la taille des données envoyées par l'instance source (root) doit être égale à la taille des données reçues par chacune des instances. Une version utile de cette fonction est la suivante :

Mcast : Permet de diffuser un message sur un sous-ensemble des instances d'une tâche.

Syntaxe :

```
Mcast(root,table)  // root : le rang de l'instance source,
                   // par défaut = 0
                   // table : table d'entiers contenant les rangs
                   // des instances participant dans la diffusion
        {
            m = sequence{..};
        }
```

À signaler que l'argument table peut être fourni au moment de la déclaration de la tâche ou peut être instancié au moment de l'utilisation. L'utilisation de l'une de ces deux opérations avec comme message à diffuser m donne lieu à une fonction du nom MDL_Bcast_m(...), respectivement MDL_Mcast_m(...).

Scatter : Permet de fractionner des données provenant de l'instance source en n segments égaux, n étant le nombre des instances pendant l'exécution, le ième segment sera envoyé à la ième instance.

Syntaxe :

```
Scatter(root) // root : le rang de l'instance source,
              // par défaut =0
        {
            m = sequence{Send_data,Recv_buf};
        }

Send_data : Variable contenant le message à distribuer
            (significatif seulement pour le root).
Recv_buf  : Adresse de début du tampon de réception.
```

Le contenu du message à distribuer peut être défini statiquement (avant l'exécution) ou dynamiquement (au moment de l'exécution).

Si on veut distribuer des segments de tailles différentes à chacune des instance d'une tâche dans des déplacements différents, on peut utiliser une nouvelle variante de Scatter qui est la suivante :

ScatterV : Permet de fractionner les données de l'instance source en n segments de tailles différentes, n est le nombre des instances pendant l'exécution. En outre, cette opération permet de spécifier le début de chaque segment par rapport au début du tampon de distribution de l'instance source.

Syntaxe :

```
ScatterV(root) // root : le rang d'instance source,
               // par défaut = 0
        {
            m = sequence{Send_buf,Taille,Disp,Recv_buf};
        }

Send_buf : Tampon de l'émission (significatif seulement pour le root).
Taille   : Table des entiers (de taille = nombre des instances)
           spécifiant la taille de segments à envoyer à chaque
           instance.
Displs   : Table des entiers  (de taille = nombre des instances).
           Le ième élément indique le déplacement relatif à Send_buf
           où doit commencer le segment à envoyer à la ième instance.
Recv_buf : Tampon de réception.
```

L'utilisation de l'une de ces deux opérations avec comme message à distribuer m donne lieu à une fonction du nom MDL_Scatter_m(....), respectivement MDL_ScatterV_m(....).

Gather : C'est l'opération inverse de Scatter, elle permet de rassembler des segments de même taille, provenant de chacune des instances de la tâche, dans le tampon de l'instance destination (root). Ces différents segments seront concaténés en fonction des rangs des leurs instances sources.

Syntaxe :

```
Gather(root) // root : le rang d'instance destination,
             // par  défaut = 0
        {
            m = sequence{Recv_buf,Send_data, count};
        }

Recv_buf  : Tampon de réception (significatif seulement pour le root).
Send_data : Tampon de l'émission.
Count     : Taille de données envoyées par chaque instance.
```

Les segments envoyés par chaque instance doivent avoir la même taille. Pour pouvoir rassembler des segments de tailles différentes et les mettre dans des déplacements différents par rapport au début du tampon de réception, on peut utiliser une nouvelle variante de Gather qui est GatherV.

GatherV : Permet de rassembler des segments de tailles différentes et les mettre dans des déplacements différents par rapport au début du tampon de réception de l'instance root.

Syntaxe :

```
GatherV(root) // root : le rang de l'instance contenant
              //        les données à diffuser

        {
            m = sequence{Recv_buf,Send_buf,Taille,Displs};
        }

Recv_buf : Tampon de réception (significative seulement pour le root).
Send_buf : Tampon d'émission.
Taille   : Table des entiers (de taille = nombre des instances)
           spécifie la taille du segment à envoyer par chaque
           instance.
Displs   : Table des entiers (de taille = nombre des instances).
           Le ième élément indique le déplacement relatif à Recv_buf
           où doit se placer le segment reçu par la ième instance.
```

À signaler que les deux arguments Taille et Displs ne peuvent être instanciés qu'au moment de l'exécution. L'utilisation de l'une de ces deux opérations avec comme message de rassemblement m donne lieu à une fonction du nom MDL_Scatter_m(...), respectivement MDL_ScatterV(...).

Allgather : Il s'agit d'une variante de l'opération Gather où toutes les instances de la tâche recevront le résultat du rassemblement, le bloc de données envoyé par la ième instance sera reçu par toutes les instances et sera placé dans le ième bloc dans le tampon de réception.

Syntaxe :

```
Allgather
        {
            m = sequence{Send_data,Recv_buf};
        }

Send_data : Début et taille du segment à envoyer par chaque instance.
Recv_buf  : Tampon de réception.
```

L'utilisation de cette opération avec comme message de rassemblement m donne lieu à une fonction du nom MDL_Allgather_m(...).

Alltoall : Il s'agit d'une extension de l'opération Allgather où toute instance envoie des données distinctes à chacune des autres instances. Le jème bloc envoyé par la ième instance sera reçu par l'instance j est sera placé dans le ième bloc.

Syntaxe :

```
Alltoall
        {
            m = sequence{Send_data, Recv_buf, Taille};
        }

Send_data : Début du tampon de l'émission.
Recv_buf  : Début du tampon de réception.
```

L'utilisation de cette opération avec comme message du rassemblement m donne lieu à une fonction du nom MDL_Alltoall_m(...).

B.2 Les fonctions de calculs collectifs

Reduce : Cette fonction permet de retourner le résultat de calcul d'une opération ayant comme arguments des données fournies par toutes les instances d'une tâche. Ce résultat sera retourné à une seule instance destination appelée root.

Parmi les différentes opérations possibles on trouve :

- MDL_MAX : pour le calcul du maximum.
- MDL_MIN : pour le calcul du minimum.
- MDL_SUM : pour le calcul du la somme.
- MDL_PROD : pour le calcul du produit.

Syntaxe :

```
Reduce(root) // root : le rang d'instance destination,
             // par  défaut égale 0
        {
            m = sequence{Input_buf,Output_buf,Count,OP};
        }
Input_buf : Tampon contenant les données à calculer.
Ouput_buf : Tampon de réception
            (significatif seulement pour le root).
Count     : Nombre des éléments à calculer, par défaut =1.
OP        : Type d'opération.
```

L'utilisation de cette opération avec comme message de réduction m donne lieu à une fonction du nom MDL_Reduce_m().

Allreduce : Il s'agit d'une extension de l'opération Reduce où le résultat de calcul sera retourné à toutes les instances.

Syntaxe :

```
Allreduce
        {
            m = sequence{Input_buf,Output_buf,Count,OP};
        }
Input_buf : Tampon contenant les données à calculer.
Ouput_buf : Tampon de réception.
Count     : Nombre des éléments à calculer, par défaut = 1.
OP        : Type d'opération.
```

L'utilisation de cette opération avec comme message de calcul m donne lieu à une fonction du nom MDL_Allreduce_m().

Annexe C

Les fonctions de l'extension de MeDLey pour les méthodes de décomposition de domaine

C.1 Les fonctions proposées

MDL_Grid_Create : Permet de construire une grille de processus ainsi que de déterminer le nombre de processus dans chaque dimension.

Syntaxe :

```
MDL_Grid_Create(dims)
dims : Vecteur des entiers de taille Ndim (nombre de dimensions)
       spécifiant le nombre de noeuds (processus) dans chaque
       dimension.
```

En entrée, si $dims[i]$ est un nombre positif, cette fonction ne modifiera pas ce nombre. Seules les entrées telles que $dims[i] = 0$ vont être modifiées. D'autre part, une valeur négative de dims[i] provoquera une erreur. Une erreur peut se provoquer aussi si le nombre des noeuds n'est pas un multiple d'un $dims[i]$ avec $dims[i] <> 0$ comme décrit dans le tableau suivant :

dims avant l'appel	Ndim	Nombre de tâches	dims retournée
(0,0)	2	6	(3,2)
(0,0)	2	7	(7,1)
(0,3,0)	3	6	(2,3,1)
(0,3,0)	3	7	erreur

Une version simple de cette fonction est la suivante :

MDL_Grid_Screate : Cette fonction utilise dims comme argument de sortie. Dans ce cas, MeDLey initialise ce vecteur à 0 et exécute ensuite MDL_Crid_Create. Le nombre de noeuds dans chaque dimension sera attribué par MeDLey d'une façon décroissante : $dims[0] >= .... >= dims[Ndim - 1]$.

MDL_Grid_Rank : Renvoie le rang du processus associé aux coordonnées locales.

```
MDL_Grid_Coords(coords,rang);

coords : Tableau des entiers représentant des coordonnées.
rang : Rang du processus dont les coordonnées sont spécifiées.
```

MDL_Get_Neighbors : Permet de déterminer les voisins d'un processus.

Syntaxe :

```
MDL_Get_Neighbors(direction,disp,rang_source,rang_dest);

direction : Sens de déplacement, 0 indique un déplacement en première
            dimension, 1 en deuxième, etc. [Entier]
disp : Pas de déplacement [entier].
rang_source : Rang du processus source.
rang_dest : Rang du processus destination.
```

Exemple d'utilisation : Dans une grille de dimension 2, pour définir les rangs des voisins East et West d'un sous-domaine, on appelle l'une des deux fonctions suivantes :

```
MDL_Get_Neighbors(0,1,rang_West,rang_East);
ou
MDL_Get_Neighbors(0,-1,rang_East,rang_West);
```

De même, pour définir les rangs des voisins North, et South, on appelle l'une des deux fonctions suivantes :

```
MDL_Get_Neighbors(1,1,rang_South,rang_North);
ou
MDL_Get_Neighbors(1,-1,rang_North,rang_South);
```

MDL_PROC_NULL sera retournée dans rang_source ou rang_dest si l'un de ces processus ne se trouvent pas dans la grille.

MDL_Grid_Coords : Renvoie les coordonnées du processus dont le rang est spécifié dans l'appel.

Syntaxe :

```
MDL_Grid_Coords(rang,coords);

rang   : Rang de processus.
coords : Coordonnées du processus spécifié par son rang
         sous forme d'une table d'entiers.
```

Une version très utile de cette fonction est la suivante :

MDL_Grid_Mycoords : Il s'agit d'une demande des coordonnées du processus qui fait l'appel.

Syntaxe :

```
MDL_Grid_Mycoords(coords);
```

```
coords : Coordonnées du processus qui fait l'appel
         sous forme d'une table d'entiers.
```

Annexe D

Code de vlasov en C++ en utilsant MeDLey

```
/*
 * Version parallèle en C du code Vlasov
 *
 * Echanges de messages réalisés avec MeDLey
 *
 * avec transferts de messages et transpositions de la fonction
 * de distribution
 */

#include "vlasovMDL.h"
#include "vlasov.h"
#include <unistd.h>
#include <sys/time.h>

double t,xl,vmax,dt,tmax,q,dv,wk,dx,dmu,zeta;
int nm1, nd2p1, nd2, m2m2, m2m1;

struct vlasov *Vdata;

/*
 * msecs:
 *   Return the time value in milliseconds.
 */

double msecs (t)
struct timeval t;
{
  return ((double) t.tv_sec * 1000 + (double) t.tv_usec / 1000
);
} /* msecs */
```

```
main(int argc, char*argv[])
{
  struct timeval  tps, tpr;        /* Time before send and after        */
  struct timezone tzps, tzpr;      /*  receive                          */
  double          inter_sr;        /* Time interval send/recv (result)  */

  double f[N][NVPROC], ftrans[NXPROC][M2], a[NVPROC];
  double b[NVPROC], c[NVPROC], d[NVPROC], e[NP1], rho[N];
  double p[NP1], g[NP1], gfield[NVPROC], x[M2P1];
  double efp[NXPROC], fv0[NVPROC];
  int ierr,i,j;
  int moi, nprocs, itime, nprnt, ngrph, nplot;
  double pie, w, xkmax;
  /*
   * Initialisation pour MeDLey
   */
  ierr = MDL_Init(&Vdata, &argc, &argv);

  moi = MDL_MyRank(Vdata);
  nprocs = MDL_FamilySize(Vdata);

  for(i=0; i<N; i++)
    {
      Vdata->Ef[i]= 0.;
      rho[i] = 0.;
      for(j=0; j<NVPROC; j++)
        f[i][j] = 0.;
    }

  for(i=0; i<NP1; i++)
    {
      e[i] = 0.;
      p[i] = 0.;
      g[i] = 0.;
    }

  for (i=0; i<M2P1; i++)
    x[i] = 0.;

  for(i=0; i<NXPROC; i++)
    {
      Vdata->Xwork[i] = 0.;
      Vdata->Work[i] = 0.;
      Vdata->Ekp[i] = 0.;
      Vdata->Rhop[i] = 0.;
      efp[i] = 0.;
      for(j=0; j<NXPROC; j++)
```

```
      Vdata->fWork[i][j] = 0.;
    for(j=0; j<M2; j++)
      ftrans[i][j] = 0.;
  }

for(i=0; i<NVPROC; i++)
  {
    a[i] = 0.;
    b[i] = 0.;
    c[i] = 0.;
    d[i] = 0.;
    gfield[i] = 0.;
    fv0[i] = 0.;
  }

gettimeofday(&tps, &tzps);


/*  NDIM=2**NCUBE
 */
itime = 0;
nprnt = 40;
ngrph = 10000;
nplot = 80;
pie = 2.00*acos(0.00);
vmax = 10.00;
dv = vmax/((double)M-0.50);
dmu = 1.00;
zeta = 0.95;
xkmax = sqrt( (2.0*zeta-1.0)/( 3.0-2.0*zeta));
xkmax = xkmax * .5;


xl = pie*2.00*((double)NMODE/xkmax);
wk = pie*2.00/xl;

tmax = 10.;

dt = 0.25;
q = 0.0;
dx = xl/((double)N);
w=0.;

if ( moi == 0)
  {
    printf("  Integration of the Vlasov equ. in V-X space\n");
    printf("  moi=%d, nprocs=%d, nproc=%d\n",
           moi, nprocs, NPROC);
```

```
        printf("  xl=%f, vmax=%f, dt=%f, tmax=%f, q=%f\n",
               xl, vmax, dt, tmax, q);
        printf("  dmu=%f, zeta=%f, dx=%f, dv=%f, xkmax=%f, w=%f\n",
               dmu, zeta, dx, dv, xkmax, w);
    }
  /*
   * generation de la fonction de distribution initiale
   */

  inicon(f, moi);

  prep(gfield, x, e, g, a, b, c, d, moi);

  ItransF(moi, f, ftrans);

  engy(ftrans, rho, Vdata->Ef, moi);

  do
    {
      t=t+dt;
      itime = itime +1;
      shift(f, gfield, e, g, a, b, c, d, moi);

      ItransF( moi, f, ftrans);

      density(ftrans, rho, efp, Vdata->Ef, moi);

      acc(ftrans, x, efp);

      if ((itime % nprnt) == 0)
        {
          engy(ftrans, rho, Vdata->Ef, moi);
        }

       transF( moi, ftrans, f);
    } while (t < tmax);

  gettimeofday(&tpr, &tzpr);

  if ((inter_sr = msecs(tpr) - msecs(tps)) < 0 )
    inter_sr = inter_sr + 1000000;
  if (moi == 0)
    printf("time = %8.3f s \n", inter_sr/1000);

  ierr = MDL_End(Vdata);
}
```

Annexe E

Spécification formelle du protocole de communication point à point de la bibliothèque *MPC*

module *MPC*

EXTENDS *Sequences, Naturals*

VARIABLES *recvBuffer, recvDescBuffer, synchronizeVar, recvBufferMonitor, recvDescBufferMonitor, synchronizeVarMonitor, addRecvBufferStep1Var, addRecvBufferStep2Var, addRecvDescBufferStep1Var, addRecvDescBufferStep2Var, waitSynchronizeVar, waitMonitorVar*

CONSTANTS *NPROCS, TAG, DATA*

$Proc \triangleq 1..NPROCS$

$Message \triangleq [emt : Proc, tag : TAG, data : DATA, isSendSync : BOOLEAN]$

$DescMessage \triangleq [emt : Proc, tag : TAG, isRecvSync : BOOLEAN]$

$$
\begin{aligned}
\mathit{InitMPC} \triangleq\ & \wedge\ \mathit{recvBuffer} \triangleq [p \in \mathit{Proc} \mapsto \langle\rangle] \\
& \wedge\ \mathit{recvBuffer} \triangleq [p \in \mathit{Proc} \mapsto \langle\rangle] \\
& \wedge\ \mathit{synchronizeVar} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{recvBufferMonitor} \triangleq [p \in \mathit{Proc} \mapsto \mathit{TRUE}] \\
& \wedge\ \mathit{recvDescBufferMonitor} \triangleq [p \in \mathit{Proc} \mapsto \mathit{TRUE}] \\
& \wedge\ \mathit{synchronizeVarMonitor} \triangleq [p \in \mathit{Proc} \mapsto \mathit{TRUE}] \\
& \wedge\ \mathit{addRecvBufferStep1Var} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{addRecvBufferStep2Var} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{addRecvDescBufferStep1Var} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{addRecvDescBufferStep2Var} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{waitSynchronizeVar} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}] \\
& \wedge\ \mathit{waitMonitorVar} \triangleq [p \in \mathit{Proc} \mapsto \mathit{FALSE}]
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{addRecvBufferStep1}(p) \triangleq \\
& \textbf{if}\ \mathit{recvBufferMonitor}[p] = \mathit{TRUE} \\
& \quad \textbf{then} \wedge\ \mathit{recvBufferMonitor}' = [\mathit{recvBufferMonitor}\ \text{EXCEPT}\ ![p] = \mathit{FALSE}] \\
& \qquad \wedge\ \mathit{waitMonitorVar}' = [\mathit{waitMonitorVar}\ \text{EXCEPT}\ ![p] = \mathit{FALSE}] \\
& \qquad \wedge\ \mathit{addRecvBufferStep1Var}' = [\mathit{addRecvBufferStep1Var}\ \text{EXCEPT} \\
& \qquad\qquad\qquad ![p] = \mathit{TRUE}] \\
& \qquad \wedge\ \text{UNCHANGED}\ \langle \mathit{recvBuffer}, \mathit{recvDescBuffer}, \mathit{synchronizeVar}, \\
& \qquad \mathit{recvDescBufferMonitor}, \mathit{synchronizeVarMonitor}, \\
& \qquad \mathit{addRecvBufferStep2Var}, \mathit{addRecvDescBufferStep1Var}, \\
& \qquad \mathit{addRecvDescBufferStep2Var}, \mathit{waitSynchronizeVar} \rangle \\
& \quad \textbf{else}\ \wedge\ \mathit{waitMonitorVar}' = [\mathit{waitMonitorVar}\ \text{EXCEPT}\ ![p] = \mathit{TRUE}] \\
& \qquad \wedge\ \text{UNCHANGED}\ \langle \mathit{recvBuffer}, \mathit{recvBufferMonitor}, \mathit{recvDescBuffer}, \\
& \qquad \mathit{recvDescBufferMonitor}, \mathit{synchronizeVar}, \mathit{synchronizeVarMonitor}, \\
& \qquad \mathit{addRecvBufferStep2Var}, \mathit{addRecvBufferStep1Var}, \\
& \qquad \mathit{addRecvDescBufferStep1Var}, \mathit{addRecvDescBufferStep2Var}, \mathit{waitSynchronizeVar} \rangle
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{addRecvBufferStep2}(p) \triangleq \\
& \textbf{if}\ \mathit{recvDescBufferMonitor}[p] = \mathit{TRUE} \\
& \quad \textbf{then} \wedge\ \mathit{recvDescBufferMonitor}' = [\mathit{recvBufferMonitor}\ \text{EXCEPT}\ ![p] = \mathit{FALSE}] \\
& \qquad \wedge\ \mathit{waitMonitorVar}' = [\mathit{waitMonitorVar}\ \text{EXCEPT}\ ![p] = \mathit{FALSE}] \\
& \qquad \wedge\ \mathit{addRecvBufferStep2Var}' = [\mathit{addRecvBufferStep2Var}\ \text{EXCEPT} \\
& \qquad\qquad\qquad ![p] = \mathit{TRUE}] \\
& \qquad \wedge\ \text{UNCHANGED}\ \langle \mathit{recvBuffer}, \mathit{recvDescBuffer}, \mathit{synchronizeVar}, \\
& \qquad \mathit{recvBufferMonitor}, \mathit{synchronizeVarMonitor}, \\
& \qquad \mathit{addRecvBufferStep1Var}, \mathit{addRecvDescBufferStep1Var}, \\
& \qquad \mathit{addRecvDescBufferStep2Var}, \mathit{waitSynchronizeVar} \rangle \\
& \quad \textbf{else}\ \wedge\ \mathit{waitMonitorVar}' = [\mathit{waitMonitorVar}\ \text{EXCEPT}\ ![p] = \mathit{TRUE}] \\
& \qquad \wedge\ \text{UNCHANGED}\ \langle \mathit{recvBuffer}, \mathit{recvBufferMonitor}, \mathit{recvDescBuffer}, \\
& \qquad \mathit{recvDescBufferMonitor}, \mathit{synchronizeVar}, \mathit{synchronizeVarMonitor}, \\
& \qquad \mathit{addRecvBufferStep2Var}, \mathit{addRecvBufferStep1Var}, \\
& \qquad \mathit{addRecvDescBufferStep1Var}, \mathit{addRecvDescBufferStep2Var}, \mathit{waitSynchronizeVar} \rangle
\end{aligned}
$$

```
addRecvBuffer(p,m) ≜
if ¬addRecvBufferStep1 Var[p]
  then addRecvBufferStep1(p)
  else if ¬addRecvBufferStep2 Var[p]
         then addRecvBufferStep2(p)
         else (∨ (∃ k ∈ 1..Len(recvDescBuffer[p]) :
                  ∧ m.emt = recvDescBuffer[p][k].emt
                  ∧ m.tag = recvDescBuffer[p][k].tag
                  ∧ if (recvDescBuffer[p][k].isRecvSync ∧ m.isSendSync)
                      then ∧ synchronizeVar' = [synchronizeVar
                                                EXCEPT ![p] = TRUE,![m.emt] = TRUE]
                           ∧ waitSynchronizeVar' = [waitSynchronizeVar
                                                EXCEPT ![p] = FALSE,![m.emt] = FALSE]
                      else if recvDescBuffer[p][k].isRecvSync
                             then synchronize(p)
                             else if m.isSendSync
                                    then synchronize(m.emt)
                                    else UNCHANGED ⟨synchronizeVar,waitMonitorVar,
                                                    waitSynchronizeVar⟩
                  ∧ recvDescBuffer' = [recvDescBuffer
                      EXCEPT ![p] = [j ∈ 1..(Len(recvDescBuffer[p]) − 1) ↦
                                      if j < k then @[j]
                                               else @[j + 1]]]
                  ∧ UNCHANGED ⟨recvBuffer⟩)
                ∨ (∀ k ∈ 1..Len(recvDescBuffer[p]) :
                  ∧ ¬(m.emt = recvDescBuffer[p][k].emt∧
                        m.tag = recvDescBuffer[p][k].tag))
                  ∧ recvBuffer' = [recvBuffer EXCEPT ![p] = Append(@,m)]
                  ∧ UNCHANGED ⟨recvDescBuffer,synchronizeVar,waitSynchronizeVar,
                               waitMonitorVar⟩)))
                ∧ recvBufferMonitor' = [recvBufferMonitor
                                        EXCEPT ![p] = TRUE]
                ∧ recvDescBufferMonitor' = [recvDescBufferMonitor
                                        EXCEPT ![p] = TRUE]
                ∧ addRecvBufferStep1 Var' = [addRecvBufferStep1 Var
                                        EXCEPT ![p] = FALSE]
                ∧ addRecvBufferStep2 Var' = [addRecvBufferStep2 Var
                                        EXCEPT ![p] = FALSE]
                ∧ UNCHANGED ⟨synchronizeVarMonitor,
                  addRecvDescBufferStep1 Var,addRecvDescBufferStep2 Var⟩
```

$addRecvDescBufferStep1(p) \triangleq$
if $recvDescBufferMonitor[p] = TRUE$
then $\wedge\ recvDescBufferMonitor' = [recvDescBufferMonitor \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ addRecvDescBufferStep1Var' = [addRecvDescBufferStep1Var \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvDescBuffer, recvBuffer, synchronizeVar, recvBufferMonitor, synchronizeVarMonitor, addRecvDescBufferStep2Var, addRecvBufferStep1Var, addRecvBufferStep2Var, waitSynchronizeVar\rangle$
else $\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvDescBuffer, recvDescBufferMonitor, recvBuffer, recvBufferMonitor, synchronizeVar, synchronizeVarMonitor, addRecvDescBufferStep2Var, addRecvDescBufferStep1Var, addRecvBufferStep1Var, addRecvBufferStep2Var, waitSynchronizeVar\rangle$

$addRecvDescBufferStep2(p) \triangleq$
if $recvBufferMonitor[p] = TRUE$
then $\wedge\ recvBufferMonitor' = [recvDescBufferMonitor \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = FALSE]$
$\wedge\ addRecvDescBufferStep2Var' = [addRecvDescBufferStep2Var \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvDescBuffer, recvBuffer, synchronizeVar, recvDescBufferMonitor, synchronizeVarMonitor, addRecvDescBufferStep1Var, addRecvBufferStep1Var, addRecvBufferStep2Var, waitSynchronizeVar\rangle$
else $\wedge\ waitMonitorVar' = [waitMonitorVar \text{ EXCEPT } ![p] = TRUE]$
$\wedge$ UNCHANGED $\langle recvDescBuffer, recvDescBufferMonitor, recvBuffer, recvBufferMonitor, synchronizeVar, synchronizeVarMonitor, addRecvDescBufferStep2Var, addRecvDescBufferStep1Var, addRecvBufferStep1Var, addRecvBufferStep2Var, waitSynchronizeVar\rangle$

```
addRecvDescBuffer(p,d) ≜
if ¬addRecvDescBufferStep1Var[p]
  then addRecvDescBufferStep1(p)
  else if ¬addRecvDescBufferStep2Var[p]
         then addRecvDescBufferStep2(p)
         else (∨ (∃ k ∈ 1..Len(recvBuffer[p]) :
                  ∧ d.emt = recvBuffer[p][k].emt
                  ∧ d.tag = recvBuffer[p][k].tag
                  ∧ if (recvBuffer[p][k].isSendSync ∧ d.isRecvSync)
                      then ∧ synchronizeVar' = [synchronizeVar
                                                EXCEPT ![p] = TRUE,![recvBuffer[p].emt] = TRUE]
                           ∧ waitSynchronizeVar' = [waitSynchronizeVar
                                                EXCEPT ![p] = FALSE,![recvBuffer[p].emt] = FALSE]
                      else if recvBuffer[p][k].isSendSync
                             then synchronize(recvBuffer[p][k].emt)
                             else if d.isRecvSync
                                    then synchronize(p)
                                    else UNCHANGED ⟨synchronizeVar,waitMonitorVar,
                                                    waitSynchronizeVar⟩
                  ∧ recvBuffer' = [recvBuffer
                      EXCEPT ![p] = [j ∈ 1..(Len(recvBuffer[p]) − 1) ↦
                                       if j < k then @[j]
                                                else @[j + 1]]]
                  ∧ UNCHANGED ⟨recvDescBuffer⟩)
                ∨ (∀ k ∈ 1..Len(recvBuffer[p]) :
                  ∧ ¬(d.emt = recvBuffer[p][k].emt∧
                        d.tag = recvBuffer[p][k].tag))
                  ∧ recvDescBuffer' = [recvDescBuffer EXCEPT ![p] = Append(@,d)]
                  ∧ UNCHANGED ⟨recvBuffer,synchronizeVar,waitSynchronizeVar,
                        waitMonitorVar⟩))
                ∧ recvDescBufferMonitor' = [recvDescBufferMonitor
                                              EXCEPT ![p] = TRUE]
                ∧ recvBufferMonitor' = [recvBufferMonitor
                                              EXCEPT ![p] = TRUE]
                ∧ addRecvDescBufferStep1Var' = [addRecvDescBufferStep1Var
                                              EXCEPT ![p] = FALSE]
                ∧ addRecvDescBufferStep2Var' = [addRecvDescBufferStep2Var
                                              EXCEPT ![p] = FALSE]
                ∧ UNCHANGED ⟨synchronizeVarMonitor,
                  addRecvBufferStep1Var,addRecvBufferStep2Var⟩
```

$synchronize(p) \triangleq$
if $synchronizeVarMonitor[p] = TRUE$
  then $\wedge\ synchronizeVar' = [synchronizeVar$ EXCEPT $![p] = TRUE]$
  $\wedge\ waitSynchronizeVar' = [waitSynchronizeVar$ EXCEPT $![p] = FALSE]$
  $\wedge\ waitMonitorVar' = [waitMonitorVar$ EXCEPT $![p] = FALSE]$
  else $waitMonitorVar' = [waitMonitorVar$ EXCEPT $![p] = TRUE]$
  UNCHANGED $\langle synchronizeVar, waitSynchronizeVar\rangle$

$waitSynchronize(p) \triangleq$
if $synchronizeVarMonitor[p] = TRUE$
  then if $synchronizeVar[p] = FALSE$
    then $\wedge\ waitSynchronizeVar' = [waitSynchronizeVar$ EXCEPT $![p] = TRUE]$
    $\wedge\ waitMonitorVar' = [waitMonitorVar$ EXCEPT $![p] = FALSE]$
    $\wedge$ UNCHANGED $\langle synchronizeVar\rangle$
    else $\wedge\ synchronizeVar' = [synchronizeVar$ EXCEPT $![p] = FALSE]$
    $\wedge\ waitMonitorVar' = [waitMonitorVar$ EXCEPT $![p] = FALSE]$
    $\wedge\ waitSynchronizeVar' = [waitSynchronizeVar$ EXCEPT $![p] = FALSE]$
    else $\wedge\ waitMonitorVar' = [waitMonitorVar$ EXCEPT $![p] = TRUE]$
    $\wedge$ UNCHANGED $\langle synchronizeVar, waitSynchronizeVar\rangle$

$(***AsynchronousSend***)$

$MPC\_Asend(p,m) \triangleq addRecvBuffer(p,m)$

$(***SynchronousSend***)$

$MPC\_Ssend(p,m) \triangleq$
  $\wedge\ addRecvBuffer(p,m)$
  $\wedge$ **if** $(addRecvBufferStep1Var[p] \wedge addRecvBufferStep2Var[p])$
    then $waitSynchronize(m.emt)$
    else $UNCHANGED\langle\rangle$

$(***AsynchronousReceive***)$

$MPCArecv(p,d) \triangleq AddRecvDescBuffer(p,d)$

$(***SynchronousReceive***)$

$MPCSrecv(p,d) \triangleq$
  $\wedge\ AddRecvDescBuffer(p,d)$
  $\wedge$ **if** $(AddRecvDescBufferStep1Var[p] \wedge AddRecvDescBufferStep2Var[p])$
    then $waitSynchronise(p)$
    else $UNCHANGED\langle\rangle$

$(* * * * NEXT * * * *)$

$Next \triangleq \exists\, p \in Proc\ :$
$\quad \vee\ (\exists\, m \in Message\ :\ \neg m.isSendSync \wedge \neg waitSynchronizeVar[m.emt] \wedge MPC\_Asend(p,m))$
$\quad \vee\ (\exists\, m \in Message\ :\ m.isSendSync \wedge \neg waitSynchronizeVar[m.emt] \wedge MPC\_Ssend(p,m))$
$\quad \vee\ (\exists\, d \in DescMessage\ :\ \neg d.isRecvSync \wedge \neg waitSynchronizeVar[p] \wedge MPC\_Arecv(p,d))$
$\quad \vee\ (\exists\, d \in DescMessage\ :\ d.isRecvSync \wedge \neg waitSynchronizeVar[p] \wedge MPC\_Arecv(p,d))$

$(* * * * Invariance * * * *)$

$NoDeadlock \triangleq \neg(\forall\, p \in Proc\ :\ waitMonitorVar[p])$

Liste des publications

[Es-sqalli 98] T. Es-sqalli and E. Dillon and J. Guyard. *Experimentation and Extension of a Specification Language of the Communications for Distributed Computation.* Technical report, INRIA RT-3455. July, 1998.

[Es-sqalli 99a] T. Es-sqalli and E. Dillon and J. Guyard. *Using MeDLey for the Grid-Decomposition Methods.* In proceedings of PDPTA'99 (International Conference on Parallel and Distributed Processing Techniques and Applications). Las Vegas, Nevada, USA. June, 1999

[Es-sqalli 99b] T. Es-sqalli and E. Dillon and J. Guyard. *Using MeDLey to Resolve the Vlasov Equation.* In proceedings of HPCN (international Conference on High Performance Computing and Networking). Amsterdam, The Netherlands. April. 1999

[Es-sqalli 99c] T. Es-sqalli and E. Fleury and E. Dillon and J. Guyard. *Message-passing specification in a CORBA environment.* In proceedings of Euro-Par'99 (European conference on parallel computing). Toulouse, France. August, 1999.

[Es-sqalli 99d] T. Es-sqalli and E. Fleury and E. Dillon and J. Guyard. *Spécification des communications pour le calcul M-SPMD dans un environnement CORBA.* In proceedings of RenPar'11 (Rencontres Francophones du Parallélisme). Rennes, France. Juin, 1999.

[Es-sqalli 00a] T. Es-sqalli and E. Fleury and E. Dillon and J. Guyard. *MeDLey: From Point-to-Point to Collective Communications.* In proceedings of HPCASIA'2000 (IEEE International Conference on High Performance Computing in Asia). Beijing, China. May, 2000.

[Es-sqalli 00b] T. Es-sqalli and E. Fleury and J. Guyard. *A Broadcast Message Passing Protocol Based on CORBA Event Service.* In proceedings of PDPTA'2000 (International Conference on Parallel and Distributed Processing Techniques and Applications). Las Vegas, Nevada, USA. June 2000.

[Es-sqalli 00c] T. Es-sqalli and E. Fleury and J. Guyard. *MPC: a new Message Passing Library in Corba.* In proceedings of MSA'2000 (IEEE International Workshop on Metacomputing Systems and Applications). Toronto, Canada. Auguest, 2000.

[Es-sqalli 01] T. Es-sqalli and E. Fleury and J. Guyard and S. Bhiri *Evaluating the Performance of CORBA for Distributed and Grid Computing Applications.* To appear in proceedings of CC-GRID 2001 (IEEE/ACM International Symposium on Cluster Computing and the Grid). Brisbane,Australia. May 2001.

[Sonnendrucker 98] E. Sonnendrucker and A. Ghizzo and E. Dillon and T. Es-sqalli and P. Bertrand and O. Coulaud *Parallelization of Semi-Lagrangian Vlasov Codes.* In proceedings of 16th International Conference on the Numerical Simulation of Plasmas. Santa-Barbara, Ca, USA. February, 1998.

Bibliographie

[Anderson 93] Thomas E. Anderson, David E. Culler and David A. Patterson. A case for now (networks of workstations). *IEEE Micro, 15(1)*, February 1993.

[Antoine 98] J-L. Antoine and P. Chatonnay. Corba : un environnemt d'exécution pour now? *RENPAR10*, pages 149–152, 1998.

[Arjomandi 95] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.Ch Eigler and G. R. Gao. Abc++: Concurrency by inheritance in c++. *IBM Systems Journal*, pages 34(1):120–137, 1995.

[Beaugendre 98] P. Beaugendre, T. Priol and G. Alléon. A client/server approach for hpc applications within a networking environment. In *Proceeding of HPCN'98, Amsterdam, The Netherlands*, pages 518–525, April 1998.

[Beguelin 94] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek and V. Sunderam. Hence: A users' guide. Technical report, Carnergie Mellon University, Oak Ridge National Laboratory and university of Tennesse, June 1994.

[Boden 95] D. Boden, N.and Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic and Su Wen-King. Myrinet a gigabit per-second local-area network. *IEEE Micro*, page 15(1) : 2936, February 1995.

[Brugeas 96] I. Brugeas. Utilisation de mpi en décomposition de domaine. Technical report, IDRIS, 1996.

[Cap 93] C. H Cap and V.J-L. Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, November 1993.

[Chandy 93] M. Chandy and C. Kesselman. Cc++: A declarative concurrent object oriented programming notation. Technical report, California Institute of Technology, March 1993.

[Chauvet 97] J. M. Chauvet. *Corba, ActiveX et Java Beans.* Eyrolles, 1997.

[Cheriton 91] David R. Cheriton and Hendrik A. Goosen. Paradigm: A highly scalable sharedmemory multicomputer architecture. *IEEE Computer, vol. 24, n 2*, February 1991.

[Coulaud 99] O. Coulaud and E. Dillon. *Para++: C++ Bindings for Message Passing Libraries.* http://www.loria.fr/projets/para++/, 1999.

[Culler 93] D. Culler, R. Karp, D. Petterson, A. Sahay, H. E. , K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proc. 4th ACM Symposium on Principles and Pratice of Parallel Programming*, page 112, 1993.

[Dillon 97a] E. Dillon. Medley : User's guid. Technical report, CRIN-CNRS/INRIA-Lorraine, February 1997.

[Dillon 97b] E. Dillon. *Proposition pour la maîtrise de la programmation par échange de messages.* PhD thesis, Université Henri Poincaré, Nancy1, October 1997.

[Dongarra 93] J. D. Dongarra, R. Pozo and D. Walker. *An objet-oriented linear algebra library for scalable systems. pages 216–233.* Rogue Wave Software Publisher, 1993.

[Dongarra 94] J.D. Dongarra, A. Lumsdaine, X. Niu, R. Pozo and Remington. K. *Sparse matrix libraries in c++ for high performance architectures. pages 214–218.* Rogue Wave Software Publisher, 1994.

[Es-sqalli 98] T. Es-sqalli, E. Dillon and J. Guyard. Experimentation and extension of a specification language of the communications for distributed computatio n. Technical report, INRIA RT-3455, July 1998.

[Es-sqalli 99a] T. Es-sqalli, E. Dillon and J. Guyard. Using medley for the grid-decomposition methods. In *Proceedings of PDPTA'99 (International Conference on Parallel and Distributed Processing Techniques and Applications) Las Vegas, Nevada, USA*, June 1999.

[Es-sqalli 99b] T. Es-sqalli, E. Dillon and J. Guyard. Using medley to resolve the vlasov equation. In *Proceedings of HPCN (international Conference on High Performance Computing and Networking) Amsterdam, The Netherlands*, April 1999.

[Es-sqalli 99c] T. Es-sqalli, E. Fleury, E. Dillon and J. Guyard. Message-passing specification in a corba environment. In *Proceedings of Euro-Par'99 (European conference on parallel computing), Toulouse, France*, Aug 1999.

[Es-sqalli 99d] T. Es-sqalli, E. Fleury, E. Dillon and J. Guyard. Spécification des communications pour le calcul m-spmd dans un environnement corba. In *Proceedings of RenPar'11 (Rencontres Francophones du Parallélisme), Rennes, France*, Jun 1999.

[Es-sqalli 00a] T. Es-sqalli, E. Fleury, E. Dillon and J. Guyard. Medley: From point-to-point to collective communications. In *Proceedings of HPCA-SIA2000 (IEEE International Conference on High Performance Computing in Asia) Beijing, China*, May 2000.

[Es-sqalli 00b] T. Es-sqalli, E. Fleury and J. Guyard. A broadcast message passing protocol based on corba event service. In *Proceedings of PDPTA'2000 (International Conference on Parallel and Distributed Processing Techniques and Applications), Las Vegas, Nevada, USA*, June 2000.

[Es-sqalli 00c] T. Es-sqalli, E. Fleury and J. Guyard. Mpc: a new message passing library in corba. In *Proceedings of MSA'2000 (IEEE International Workshop on Metacomputing Systems and Applications), Toronto, Canada*, Aug 2000.

[Flynn 95] Michael J. Flynn. *Computer architecture.* Sudbury, Mass, 1995.

[Forum 95] MPI Forum. *Message Passing Interface.* NSF and ARPA, 1995.

[Gamboa Dos Santos 98] C. Gamboa Dos Santos. *Apport de l'approche gestion de réseau pour le placement de tâches dans le modèle de programmation par échange de messages.* PhD thesis, Université Henri Poincaré, Nancy1, October 1998.

[Gannon 93] D. Gannon, F. Bodin, P. Beckman, X. Shelby and S. Narayana. Distributed pc++: Basic ideas for an object parallel language. *Journal of Scientific Programming*, pages 2(3):7–22, 1993.

[Geib 97] J-M. Geib, C. Gransart and P. Merle. *CORBA, des concepts à la pratique.* InterEditions, 1997.

[Gengler 96] M. Gengler, S. Ubéda and F. Desprez. *Initiation au parallélisme.* Masson, 1996.

[Ghizzo 93] A. Ghizzo. *Application à la modélisation des interactions laser-plasmas et de l'équation gyrocinétique de Vlasov.* PhD thesis, UHP Nancy, LPMI, Juin 1993.

[Gibbons 89] P. G Gibbons. A more practical pram model. In *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 168–158, 1989.

[Globus 00] Globus. *The globus Project.* http://www.globus.org/, Argonne National Laboratory, May 2000.

[Gokhale 97] A. Gokhale. Evaluating corba latency and scalability over high-speed atm networks. In *IEEE 17th International Conference on Distributed Systems (ICDCS97)*, May 1997.

[Gressier 95] E. Gressier. *Concept d'Appel de Procédure à Distance.* CNAM-CEDRIC, November 1995.

[Gropp 94] W. Gropp, E. Lusk and A. Skjellum. *Using MPI : Portable Programmaing with the Message-Passing Interface.* Mass-Ins-Tech, 1994.

[Gropp 99] W. Gropp, E. Lusk and R Thakur. *Using MPi-2 Advanced Features of the Message-Passing Interface.* Scientific and Engineering Computation Series, 1999.

[Group. 98] Xopen Group. *DCE Today.* ISBN:0-13-575697-9, 1998.

[Group 99] Distributed System Research Group. *CORBA Comparaison Project Project.* http://nenya.ms.mff.cuni.cz/thegroup/COMP/, June 1999.

[Guide 96] Guide. Architectures parallèles. *Informatiques Magazine*, pages 96–99, 1996.

[Henning 98] M. Henning. *Binding, Migration ans Scalibility in CORBA.* DSTC, St. Lucia, http://www.dstc.edu.au/BDU/staff/michi-henning.html, October 1998.

[HPFIT 99] HPFIT. *High Performance Fortran Integrated Tools.* http://www.ens-lyon.fr/ desprez/FILES/RESEARCH/SOFT/HPFIT/, 1999.

[Jain 94] R. Jain. *FDDI Handbook: High-Speed Networking with Fiber and Other Media.* Addison-Wesley, April 1994.

[J.Chergui 97] J.Chergui, I.Dupays, D.Girou and S.Requena. *Message Pasing Interface*, Juin 1997.

[Kale 93] L.V. Kale and S. Krishnann. Charm++ : A portable concurrent object oriented system based on c++. In *Proceedings of OOPSLA-93, Washington, D.C.*, pages 91–108, October 1993.

[Koelbel 94] Charles H. Koelbel. *The High performance fortran handbook.* Mit Press, Cambridge, London, 1994.

[Kofman 96] D. Kofman. *Réseaux Haut Débit.* InterEditions, 1996.

[Kohl 96] J. A. Kohl and G. A. Geist. Xpvm 1.0 users's guide. Technical Report DE-AC05-84OR21400, Oak Ridge National Laboratory, Oak Ridge, Tennesse 37831, November 1996.

[Kuhns 99] F. Kuhns and D. Levine. The design and performance of rio – a real-time i/o subsystem for orb endsystems. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS99), Vancouver, British Columbia, Canada*, June 1999.

[Laboratory 99] Argonne National Laboratory. *MPICH-A Portable Implementation of MPI, version 1.2.0.* http://www-unix.mcs.anl.gov/mpi/mpich/, 1999.

[Lamport 97a] L. Lamport. The operators of TLA+. Technical Report 1997-006a, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1997.

[Lamport 97b] L. Lamport and L. C. Paulson. Should Your Specification Language Be Typed? Technical Report 1997-147, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1997.

[Lamport 98] L. Lamport. The module structure of TLA+. Technical Report 1996-002a, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, January 1998.

[Lastovetsky 00] A. Lastovetsky. *The mpC Parallel Programming Environment.* http://www.ispras.ru/ mpc/, 2000.

[Lefevre 97] L. Lefevre. *Conception et Mise en OEuvre d'un Environnement de Programmation Parallèle fondé sur un système de Mémoire Distribuée Virtuellement Partagée.* PhD thesis, Ecole nationale supérieure de Lyon, January 1997.

[Levine 97] D. Levine and T. Harrison. The design and performance of a real-time corba object event service. In *Proceedings of OOPSLA '97, Atlanta, Georgia*, October 1997.

[Levine 99] D. Levine and S. Flores-Gaitan. Measuring os support for real-time corba orbs. In *Fourth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99), Santa Barbara, California*, January 1999.

[Li 86] K. Li. Shared virtual memory on looselycoupled multiprocessors. Technical Report yaleurr492, Yale University, October 1986.

[Li 95] Z. Li, P. H. Mills and J. H. Reif. Models and resource metrics for parallel and distributed computation. In *Proc. 28th Annual Hawaii Internationnal Conference on System Sciences, Wailea, Maui, Hawai*, January 1995.

[Litzkow 88] M. Litzkow, M. Livnym and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of ICDCS'88 : International Conference on Distributed Computer Systems, San Jose, CA, USA*, pages 104–111, 1988.

[LSF 96] LSF. *LSF 2.2 Programmer's Guide.* Canada, February 1996.

[Mentat 94] Mentat. The mentat 2.7 programming language reference manual. Technical report, University of Virginia The Mentat Research Group., 1994.

[NQE 97] NQE. *NQE 3.2 User's Guide.* Cray Research, January 1997.

[OMG 95] OMG. *Unified Modelling Language.* http://www.omg.org/library/schedule/, November 1995.

[OMG 98a] OMG. *The Common Object Request Broker: Architecture and Specification, revision 2.2.* http://www.omg.org/corba/corbiiop.htm, February 1998.

[OMG 98b] OMG. *CORBAservices : Common Object Services Specification.* http://www.omg.org/corba/sectrans.htm, July 1998.

[OOC 97] OOC. *ORBacus for C++ and Java.* http://www.ooc.com/. Object-Oriented Concepts. Inc, 1997.

[OOMPI 00] OOMPI. *The Object Oriented MPI.* http://www.lsc.nd.edu/research/oompi/, 2000.

[Parsons 94] R. Parsons and D. Quinlan. *A++/p++ array classes for architecture independent finite difference computations.* 1994.

[Plasil 98a] F Plasil. Charles university response to the benchmark rfi. Technical Report OMG document bench, Dep. of SW Engineering, Charles University, Prague., October 1998.

[Plasil 98b] F. Plasil, P. Tuma and A. Buble. Corba benchmarking. Technical Report 98/7, Dep. of SW Engineering, Charles University, Prague., July 1998.

[Plasil 98c] M Plasil, F.and Stal. An architectural view of distributed objects and components in corba, java rmi and com/dcom. *Software Concepts and Tools (vol. 19, no. 1)*, 1998.

[Priol 98] T. Priol and C. René. Cobra: A corba-compliant programming environment for high performance computing. In *Proceeding of Europar'98, Southmton, UK*, pages 1114–1122, September 1998.

[Prycker 95] M. Prycker. *ATM : Mode de transfert asynchrone. Collection Systèmes Distribués.* Prentice-Hall/Masson, 1995.

[Ptools 99] Ptools. *The Parallel Tools Consortium.* http://www.ptools.org/, 1999.

[Pyarali 99] I. Pyarali, C. O'Ryan, V. Kachroo, A. Gokhale and N. Wang. Applying optimization patterns to design real-time orbs. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, San Diego, CA.*, May 1999.

[Quinton 97] R. Quinton. An introduction to socket programming. Technical report, University of Western Ontario, London, Ontario Canada., May 1997.

[Schmidt 98] D.C Schmidt. *Real-time CORBA with TAO (The ACE ORB).* http://siesta.cs.wustl.edu/ schmidt/TAO.html, 1998.

[Scott 95] P. Scott, L. Mario and C. Andrew. Hight performance messaging on workstations : Illinois fast messages (fm) for myrinet. In *Proceedingss of the 1995 ACM/IEEE Supercomputing Conference, San Diego, CA, USA*, December 1995.

[Shanely 98] K. Shanely. *History and Overview of the TPC.* Tansaction Processing Performance Council, DSTC, http://www.tpc.org/articles/tpc.overview.history.1.html, February 1998.

[Skillicorn 96] D. B. Skillicorn, Jonathan M. D., J. M. D. Hill and W. F McColl.

Questions and answers about bsp. Technical Report Rapport Technique 1596, University Computing Laboratory, August 1996.

[Soley 95] R.M Soley and C. M. Stone. Object management architecture guide, revision 3.0. Technical Report ISBN : 0-471-14193-3, Sons, Inc., USA, July 1995.

[Sonnendrucker 98] E. Sonnendrucker, A. Ghizzo, E. Dillon, T. Es-sqalli, P. Bertrand and O. Coulaud. Parallelization of semi-lagrangian vlasov codes. In *Proceedings of 16th Conference on the Numerical Simulation of Plasmas, Santa-Barbara, Ca, USA*, February 1998.

[Team 94] PVM Team. *PVM 3 users's guide and reference manual.* ORNL/TM-12187, Oak Ridge National Laboratory, 1994.

[Team 00] LAM Team. *LAM / MPI Parallel Computing, version 6.3.2.* http://www.mpi.nd.edu/lam/6.3/, 2000.

[Thai 99] Thuan L. Thai. *DCOM Learnin.* ISBN:1-56592-581-5, April 1999.

[Thorsten 92] E. V. Thorsten, D. E. Culler, S. C. Goldstein and K. E. Schauser. Active messages : a mecanism for integrated communication and computation. Technical report, University of California at Berkely, Departement of Computer Science, March 1992.

université HENRI POINCARÉ
NANCY 1

Monsieur **ES-SQALLI Tawfik**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 27 NOV. 2000 n° 447

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY I
LE PRÉSIDENT
V. BURLET

Université Henri Poincaré, Nancy-I
24-30 rue Lionnois - B.P. 3069 - 54013 NANCY CEDEX
Tél. : 03 83 85 48 00 - Fax : 03 83 85 48 48

Résumé

MeDLey (*Message Defintion Language*) est un langage de spécification des communications pour les applications parallèles. Son principe repose sur l'implantation des communications sur différentes plates-formes de communication en utilisant un langage et une syntaxe unifiés.

L'apport original de la thèse a été guidé par le besoin de faire évoluer ce langage pour couvrir un champ d'application plus vaste. Nous avons focalisé nos recherches sur les architectures parallèles à mémoire distribuée composées d'un réseau de stations.

Dans ce cadre, une première extension a concerné les communications collectives ; nous avons proposé une nouvelle approche permettant de spécifier et réaliser ce type de communication. La deuxième extension a concerné les méthodes de décomposition de domaine ; nous avons définit un nouveau mode d'interconnexion, puis proposé des fonctions permettant d'appliquer ces méthodes. La troisième extension avait pour objectif l'implantation de ce langage dans un environnement *CORBA*. Ce choix est justifié d'une part par les performances de *CORBA* au niveau du temps de communication au vu des tests que nous avons réalisés ; et d'autre part par les avantages de cet environnement en terme de services, en particulier celui d'interopérabilité. Pour y parvenir, nous avons développé une bibliothèque de communication par échange de messages au dessus du *CORBA* appelée *MPC* (*Message Passing in Corba environment*) qui propose un environnement complet pour la programmation par échange de messages dans un environnement *CORBA*.

Enfin, nos derniers travaux ont portés sur l'expérimentation et la validation des outils développés. Dans ce contexte, une expérimentation du langage *MeDLey* a consisté à l'implantation d'un code de simulation numérique utilisé par les physiciens. Ensuite, nous avons validé le protocole de communication point à point de la bibliothèque *MPC* en utilisant *TLA* (*Temporal Logic of Actions*).

Mots clés : CORBA, passage de messages, systèmes distribués, calcul distribué, parallélisme, metacomputing

Resume

MeDLey (Message Definition Language) is a specification language of communications for distributed computation. Its aim is to provide to the user an abstract vision of his/her application in terms of tasks and exchanges between these tasks independently of material architecture and the means of communication used. Based on this specification, the MeDLey compiler generates several levels of implementations for various communication primitives.

The original contribution of our thesis was guided by the need to evolve this language. We focused our search on the parallel architectures with distributed memory; especially those composed from a network of workstations.

Within this framework, a first extension related to the collective communications; we proposed a new approach allowing to specify and carry out this type of communication. The second extension related to the `grid-decomposition methods`; we defined a new interconnection model, then proposed functions allowing the use of these methods. The third extension aimed to implement the MeDLey language in a CORBA environment. This choice is justified by the advantage of CORBA in term of interoperability and isolation between client and server, using network protocol, material infrastructure, programming language and data transport mechanisms. In order to implement the MeDLey language in CORBA, we developed a message-passing library called MPC (Message Passing in CORBA). This library is based on the use of the methods invocation mechanism on remote objects of CORBA and proposes a complete and usual message-passing environment in CORBA.

Finally, our last work concerned the experimentation and the validation of the developed tools. In this context, an experimentation of the MeDLey language consisted on the implementation of a simulation code used by the physicists. Then, we validated the point-to-point communication protocol of the MPC library by using TLA (Temporal Logic of Actions).

Keywords : CORBA, message passing, distributed systems, distributed computation, parallelism, metacomputing.