N° d'ordre : 2946

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Monsieur Guillaume MERCIER**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Communications à hautes performances portables
en environnements hiérarchiques, hétérogènes et dynamiques

Soutenue le : 20 Décembre 2004

Après avis des rapporteurs :

Franck CAPPELLO .	Directeur de Recherche à l'INRIA
Thierry PRIOL	Directeur de Recherche à l'INRIA

Devant la commission d'examen composée de :

Françoise BAUDE .	Maître de Conférences	
Jacques BRIAT	Maître de Conférences	
Franck CAPPELLO .	Directeur de Recherche à l'INRIA	
Olivier COULAUD .	Directeur de Recherche à l'INRIA	Président et Rapporteur
Raymond NAMYST	Professeur des Universités	Directeur de thèse
Thierry PRIOL	Directeur de Recherche à l'INRIA	

– 2004 –

Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques

Résumé : Cette thèse a pour cadre les communications dans les machines parallèles dans une optique de calcul haute-performance. Les évolutions du matériel ont rendu nécessaire les adaptations des logiciels destinés à exploiter les machines parallèles. En effet, les architectures de type "grappes" sont maintenant très répandues et l'apparition des grilles de calcul complique encore plus la situation car l'obtention des hautes performances passe par une exploitation des différents réseaux rapides disponibles et une prise en compte de la hiérarchie intrinsèque des configurations considérées. Au niveau applicatif, de nouvelles exigences émergent comme la dynamicité. Or, ces aspects sont trop souvent partiellement traités, en particulier dans les implémentations du standard de programmation par passage de messages MPI. Les solutions existantes se concentrent sur la hiérarchie et l'hétérogénéité ou la dynamicité, exceptionnellement les deux. En ce qui concerne les premiers aspects, des simplifications conduisent à une exploitation suboptimale du matériel potentiellement disponible.

Nous avons analysé des implémentations existantes de MPI et avons proposé une architecture répondant aux besoins formulés. Cette architecture repose sur une forte interaction entre communications et processus légers et son cœur est constitué par un moteur de progression des communications qui permet d'améliorer substantiellement les mécanismes existants. Les deux éléments logiciels fondamentaux sont une bibliothèque de processus légers (Marcel) ainsi qu'une couche générique de communication (Madeleine). L'implémentation de cette architecture a débouché sur le logiciel MPICH-Madeleine, utilisé ou évalué par plusieurs équipes et projets de recherche en France comme à l'étranger. L'évalution des performances (comparaisons avec Madeleine, mesures des opérations point-à-point, noyaux applicatifs) menée avec plusieurs réseaux haut-débit sur des grappes homogènes de machines multi-processeurs et les comparaisons avec MPICH-G2 ou PACX-MPI en environnement hétérogène démontrent que MPICH-Madeleine atteint des résultats de niveau similaire voire supérieur à ceux d'implémentations spécialisées de MPI.

Mots-clefs : Grappes de PC, MPI, réseaux rapides, hiérarchie, hétérogénéité, dynamicité, haute-performance.

Remerciements

Je souhaite tout d'abord remercier MM. Cappello et Priol d'avoir accepté d'être les rapporteurs de ce travail : leurs conseils m'ont permis d'améliorer substantiellement la qualité de ce document. Je remercie également Mme Françoise Baude et MM. Briat et Coulaud d'avoir bien voulu être membres du jury.

Cette thèse n'auraient pas pu voir le jour si je n'avais rencontré certains professeurs au collège, au lycée ou encore à l'université ; aussi je profite de cette occasion pour remercier MM. Boukalaba et Fluckiger de m'avoir donné le goût des mathématiques, Mme Odile Millet-Botta de m'avoir soutenu pendant les années passées à l'ENS-Lyon et M. Jean Duprat de m'avoir autorisé à poursuivre mes projets linguistiques extrême-orientaux durant ces mêmes années.

Bien entendu, tout ceci n'aurait pas été possible si Raymond Namyst ne m'avait pas accepté en tant qu'étudiant. J'ai beaucoup appris à son contact et profite de ces remerciements pour lui témoigner ma gratitude. J'y associe les membres du projet RUNTIME et en particulier Pierre-André Wacrenier, pour ses conseils avisés et ses avis toujours nuancés. L'INRIA Futurs a tenu une place prépondérante pour l'accomplissement de ce travail et je le remercie, tout comme Daniel Balkanski pour m'avoir aidé à améliorer la qualité de mon travail.

Ces années de thèse n'auraient pas été les mêmes sans les nombreux compagnons de fortune (ou d'infortune, c'est selon) qui ont partagé les bureaux que j'ai occupés : Arnaud Legrand, Vincent Danjean, Nicolas Bonichon, Guillaume Latu, Jean-Christophe Aval, Rodrigue Ossami, Irek Tobor.

Je remercie vivement ma famille de m'avoir laissé faire à ma guise durant autant d'années sans (trop) poser de questions et de m'avoir fait confiance.

Enfin, je souhaite remercier ma femme, Aurélie, pour son soutien, sa compréhension et son amour, qui m'ont permis de tenir tous mes objectifs ; notre collaboration ne fait que commencer . . .

Table des matières

Introduction 1

1 Les communications dans les architectures parallèles 5

1.1 Évolution des architectures parallèles et de leurs outils d'exploitation 5

1.1.1 Les plate-formes matérielles 5

1.1.2 Les interfaces de communication pour architectures parallèles 10

1.1.3 Vers des configurations hiérarchiques, hétérogènes et dynamiques . . 13

1.2 Les nouveaux défis des bibliothèques de communication hautes-performances 17

1.2.1 Introduction 17

1.2.2 L'exploitation des grappes de PC 18

1.2.3 La gestion des configurations hiérarchiques 19

1.2.4 Le support des réseaux hétérogènes 21

1.2.5 Le support des configurations et applications dynamiques 22

1.3 Nos objectifs 23

1.3.1 Définitions des services recherchés 23

1.3.2 Cahier des charges 24

2 Un état de l'art 25

2.1 Architectures supportant hiérarchie et hétérogénéité 25

2.1.1 Deux propriétés pour une unique problématique 26

2.1.2 Le cas particulier des grappes de machines multiprocesseurs 26

2.1.3 Les architectures inter-opérables 34

2.1.4 Les architectures intrinsèquement hétérogènes 42

2.1.5 Autres approches 48

2.2 Systèmes supportant des applications dynamiques 51

2.2.1 Introduction 51

2.2.2 Solutions avec simple gestion dynamique des processus 52

2.2.3 Solutions intégrant des mécanismes de tolérance aux pannes 52

2.3 Bilan de l'existant 60
2.3.1 Gestion de la hiérarchie et de l'hétérogénéité 60
2.3.2 Gestion de la dynamicité 61
2.3.3 Conclusion 61
2.3.4 Tableau récapitulatif des fonctionnalités supportées 62

3 Proposition : une architecture extensible pour un support efficace des environnements hiérarchiques, hétérogènes et dynamiques dans MPI 63
3.1 Analyse et démarche 63
3.1.1 Rappels des exigences 63
3.1.2 Les raisons de l'absence actuelle de solutions 65
3.1.3 Enseignements retenus 66
3.1.4 Démarche de recherche 67
3.2 Proposition d'architecture 70
3.2.1 Exigences architecturales 70
3.2.2 Des communications performantes sur réseaux hétérogènes 75
3.2.3 Des sessions dynamiques sans impact notable pour les performances . 77
3.3 Exploitation de configurations à plus large échelle 79
3.3.1 À propos des niveaux de hiérarchie dans MPI 79
3.3.2 Compatibilité et complémentarité des approches 80
3.4 Bilan de la proposition 81
3.4.1 Une architecture flexible et extensible 81
3.4.2 Des communications potentiellement efficaces sur réseaux homogènes et hétérogènes 82
3.4.3 Des applications fusionnables et séparables aisément 82

4 Réalisation : MPICH-Madeleine, une implémentation multi-grappes, multi-réseaux et multi-sessions du standard MPI 83
4.1 Le substrat : le logiciel MPICH 83
4.1.1 Une architecture stratifiée 84
4.1.2 L'Abstract Device Interface (ADI) 85
4.1.3 MPICH et l'évolution de MPI 87
4.2 L'outil : la suite logicielle PM2 87
4.2.1 Évolutions de PM2 87
4.2.2 La bibliothèque de communication Madeleine 88
4.2.3 La bibliothèque de processus légers Marcel 93
4.3 Mise en œuvre du support multi-protocoles 94

4.3.1 Vue d'ensemble de la réalisation 94
4.3.2 MPICH et Le multithreading 96
4.3.3 Support multi-réseaux et multi-grappes : le module `ch_mad` 99
4.3.4 Support des communications par mémoire partagée : le module `ch_smp` 111
4.4 Mise en œuvre du support multi-sessions 114
4.4.1 Le lanceur d'applications : Leony 114
4.4.2 Implémentation dans Madeleine 116
4.4.3 Extension de l'interface MPI 118
4.5 Stratégies alternatives d'implémentation 119
4.5.1 Support de la hiérarchie 119
4.5.2 Support de l'hétérogénéité 120
4.6 Conclusion 120
4.6.1 Bilan du travail réalisé 121
4.6.2 Réflexions sur la mise en œuvre 121
4.6.3 À propos de MPICH2 123

5 Validation : résultats expérimentaux 125

5.1 Description de la configuration matérielle 125
5.1.1 Dalton 125
5.1.2 Jack 126
5.2 Comparaison des performances avec Madeleine 126
5.2.1 SCI/SISCI 126
5.2.2 Myrinet/GM 130
5.2.3 GigaBitEthernet/TCP 133
5.2.4 Conclusion 136
5.3 Évaluation sur grappes homogènes de machines multi-processeurs 137
5.3.1 Représentativité des tests 137
5.3.2 Performance des communications point-à-point 137
5.3.3 Amélioration de la progression des opérations non-bloquantes 142
5.3.4 Conservation de la puissance de calcul 145
5.3.5 Conclusion 148
5.4 Évaluation sur grappes hétérogènes et grappes de grappes 150
5.4.1 Comparaisons avec MPICH-G2 150
5.4.2 Comparaison avec PACX-MPI 152
5.4.3 Conclusion 155
5.5 Conclusion générale 155

Conclusion et perspectives 157

Bibliographie 167

Table des figures

1 Un réseau de stations de travail 7
2 Une grappe de PC 7
3 Une grappe de grappes faiblement couplées 9
4 Une grappe multiplement câblée (avec des partitions) 9
5 Approche hybride 27
6 Approche hybride – Optimisation des communications intra-nœuds 29
7 Approche hybride – Exploitation du parallélisme intra-processus 29
8 Architecture de *thread MPI* 30
9 Architecture bimodulaire 32
10 L'architecture fédérative multi-MPI 35
11 L'architecture fédérative avec module exterieur de communication 36
12 Topologies d'interconnexion 38
13 Retransmissions dans IMPI lors d'un appel à `MPI_Send` 38
14 L'architecture récursive 39
15 Architecture de GridMPI 41
16 L'architecture multi-modulaire 43
17 Structure modulaire de LAM/MPI 44
18 L'architecture unimodulaire 46
19 Architecture de MPI/I-WAY 47
20 Architecture de MPIConnect 48
21 Structure d'un processus-routeur 49
22 Principe du routage/retransmission dans MetaMPICH 50
23 Structure des processus tuMPI et du coordinateur central 54
24 Architecture de FT-MPI 55
25 Architectures comparées de MPICH-V et MPICH-V2 57
26 Principe de fonctionnement de MPI/FT(TM) 58
27 Structure des processus Starfish MPI 60
28 Architecture proposée 71

29 Cohabitation moteur/application 73
30 Deux niveaux pour le routage 76
31 Une possibilité d'extension en utilisant notre architecture en tant que *VendorMPI* pour MPICH-G2 81
32 Structure stratifiée de MPICH 84
33 Organisation de l'ADI 85
34 Envoi de message avec MADELEINE. 91
35 Exemple de grappe de grappes hétérogène 92
36 Fichiers de configuration 92
37 Couches logicielles de MPICH-Madeleine 95
38 Structure des messages dans MPICH-Madeleine 101
39 Principe du mode de transfert *eager* 103
40 Implantation du mode de transfert *eager* dans le cas de messages attendus . . 104
41 Implantation du mode de transfert *eager* dans le cas de messages inattendus . 104
42 Implantation du mode de transfert *eager* dans le cas de messages très courts . 105
43 Principe du mode de transfert *rendez-vous* 106
44 Implantation du mode de transfert *rendez-vous* 107
45 Organisation de la mémoire partagée (trois processus et un pipe-line à deux étages) 112
46 Comparaison des latences sur SISCI/SCI (canal physique) 127
47 Comparaison des débits sur SISCI/SCI (canal physique) 128
48 Comparaison des latences sur SISCI/SCI (canal virtuel) 129
49 Comparaison des débits sur SISCI/SCI (canal virtuel) 129
50 Comparaison des latences sur Myrinet/GM (canal physique) 130
51 Comparaison des débits sur Myrinet/GM (canal physique) 131
52 Comparaison des latences sur Myrinet/GM (canal virtuel) 132
53 Comparaison des débits sur Myrinet/GM (canal virtuel) 132
54 Comparaison des latences sur GigaBitEthernet/TCP (canal physique) 133
55 Comparaison des débits sur GigaBitEthernet/TCP (canal physique) 134
56 Comparaison des latences sur GigaBitEthernet/TCP (canal virtuel) 135
57 Comparaison des débits sur GigaBitEthernet/TCP (canal virtuel) 135
58 Comparaison des latences en mémoire partagée 138
59 Comparaison des débits en mémoire partagée 138
60 Comparaison des latences sur SISCI/SCI 139
61 Comparaison des débits sur SISCI/SCI 140
62 Comparaison des latences sur Myrinet/GM 140

63 Comparaison des débits sur Myrinet/GM 141
64 Comparaison des latences sur GigaBitEthernet/TCP 142
65 Comparaison des débits sur GigaBitEthernet/TCP 142
66 Amélioration de la progression des communications non-bloquantes (exemple avec quatre processus) 143
67 Test HPL sur un nœud avec 4 processus 146
68 Test HPL sur 2 nœuds avec 8 processus (HyperThreading activé) 147
69 Test HPL sur 2 nœuds avec 4 processus (HyperThreading désactivé) 147
70 Test HPL sur 4 nœuds et 16 processus 148
71 Latences des communications inter-grappes pour MPICH-G2 et MPICH-Madeleine 151
72 Débits des communications inter-grappes pour MPICH-G2 et MPICH-Madeleine 151
73 Latences de plusieurs *Vendor MPI* pour MPICH-G2 152
74 Débits de plusieurs *Vendor MPI* pour MPICH-G2 153
75 Latences des communications inter-grappes entre PACX-MPI et MPICH-Madeleine 153
76 Débits des communications inter-grappes entre PACX-MPI et MPICH-Madeleine 154

Introduction

Si le recours au parallélisme en informatique est désormais courant, l'utopie d'un parallélisme automatisé pour les applications à gros grain est quant à elle révolue. La complexité de mise en œuvre des techniques, conjuguée à l'évolution incessante du matériel, rend caduque toute tentative d'automatisation de la parallélisation des applications. Cette étape ne pouvant se concrétiser, des outils et environnements de programmation permettant à leurs utilisateurs de paralléliser leur code ont vu le jour. Le domaine du calcul haute-performance n'échappe pas à ce constat et si la tâche d'écriture du code parallèle incombe toujours à l'être humain, ce dernier peut en revanche se reposer sur des logiciels pour l'aider à mieux exploiter la configuration matérielle.

Cependant, si le nombre et la nature des outils indispensables demeurent stables, il n'en va pas de même pour les architectures parallèles. Le visage qu'elles présentent à considérablement changé au cours des dernières années ce qui a poussé les logiciels à s'adapter. Ainsi, nous sommes passés d'architectures de type supercalculateurs – dispendieux et peu disponibles – aux grappes de PC, moins onéreuses mais plus complexes dans la façon d'aborder leur programmation. Ces mêmes architectures continuent d'évoluer avec le contexte des grilles de calcul et du *metacomputing* puisque les grappes de grappes ont maintenant fait leur apparition. Ces machines sont très hiérarchiques, hétérogènes et les applications destinées à leur exploitation ont de plus en plus des exigences de support pour la dynamicité.

Cette exploitation passe par l'utilisation d'outils nouveaux et spécialement conçus à cette fin mais qui ont l'inconvénient de ne pas être standard, et les programmeurs cherchant à développer des applications portables n'y ont donc pas recours. L'emploi d'interfaces bien définies et standardisées perdure mais les implémentations ne parviennent pas toujours à répondre aux besoins et exigences des nouvelles configurations. Ce décalage est sensible en particulier pour le seul standard de programmation d'applications parallèles qui utilisent comme paradigme le passage de messages : MPI. Cet outil est incontournable à l'heure actuelle et pourtant, les nombreuses instances dont il fait l'objet sont incapables de prendre en compte l'intégralité des contraintes matérielles et applicatives. Le résultat est que l'exploitation des grappes hétérogènes ou des grappes de grappes n'est pas encore totalement satisfaisant, en dépit des nombreuses recherches menées dans ce domaine.

Cadre de la thèse et contribution

L'objectif du projet RUNTIME de l'INRIA est de s'atteler à l'étude, la conception et la réalisation d'outils permettant une exploitation optimale des configurations matérielles (processeurs et réseaux) de type "grappes". Les outils en question sont appelés des ***supports d'exécution*** et sont destinés de par leur localisation à bas niveau dans la pile logicielle à être intégrés au sein d'intergiciels (***middleware***) ou d'interfaces de plus haut niveau, comme le standard de programmation par passage de messages : *Message Passing Interface* (MPI). Ce standard, conçu à l'origine pour la programmation d'architectures "régulières" comme les grappes ou les supercalculateurs, a vu naître nombre d'adaptations répondant partiellement aux besoins de fonctionnalités ou de performances apparaissant dans le cas des grappes de grappes.

Cette thèse s'inscrit dans le cadre de ce projet et poursuit plusieurs objectifs : le premier est de faire un état des lieux de la situation actuelle. L'établissement d'un état de l'art nous permettra d'étudier les travaux accomplis ainsi que les idées développées afin de pouvoir exploiter simplement et efficacement des grappes de grappes hétérogènes avec un outil tel que MPI. Ainsi que nous le verrons, le manque dans ce domaine est criant, malgré le nombre important d'équipes de recherches travaillant sur cette problématique.

Notre contribution consiste premièrement à proposer une architecture originale répondant à l'ensemble des problèmes posés (hiérarchie, hétérogénéité et dynamicité). En second lieu, cette architecture doit être mise en œuvre effectivement afin de valider nos idées par l'expérience. Nous allons donc procéder à l'évaluation de l'implémentation de l'architecture proposée. Ce travail a entraîné la publication des articles suivants : [ABD+02], [ABD+00], [AMN01] et [AM03] et il est de plus disponible librement, évaluable et actuellement utilisé par plusieurs équipes de recherches en France comme à l'étranger.

Organisation du document

Ce document est divisé en cinq chapitres : le premier présente le contexte d'étude et souligne les évolutions du matériel et des logiciels. Ce chapitre est l'occasion pour nous d'insister sur les nouvelles tendances observées et de montrer que la hiérarchie, l'hétérogénéité et la dynamicité deviennent des caractéristiques incontournables pour les bibliothèques de communication, et en particulier pour un standard de programmation d'applications parallèles tel que MPI. Ce standard est la cible d'un nombre impressionnant de recherches à travers le monde et cette foison d'adaptations en vue d'intégrer dans l'implémentation les nouvelles fonctionnalités répondant aux caractéristiques ci-dessus fait l'objet de l'état de l'art au deuxième chapitre. Nous y décrivons et analysons les solutions mises en œuvre depuis une douzaine d'années. Les implémentations actuelles n'offrent qu'un support partiel et limité des fonctionnalités désirées, si bien qu'il n'existe en pratique pas de version libre de MPI capable de répondre simultanément aux défis de la hiérarchie, de l'hétérogénéité et de la dynamicité. Ce constat nous amène à formuler au chapitre 3 une proposition d'architecture répondant à l'ensemble de ces défis. Cette architecture repose sur des axes directeurs qui n'étaient pas ou plus explorés jusqu'à présent car ils n'exhibaient pas de résultats suffisants ou bien n'avaient pu être mis en pratique en raison de difficultés techniques. Ces dernières sont en effet nombreuses et décrites dans le chapitre 4, qui est donc la partie pratique de notre contribution. Ce chapitre expose la réalisation du logiciel implémentant notre architecture et

passe en revue les aspects essentiels comme la gestion des protocoles multiples de communication ou celle des applications dynamiques. Nous montrons les solutions que nous avons apportées pour finalement disposer d'une version de MPI compatible avec nos exigences. Cette version de MPI, MPICH-Madeleine est ensuite évaluée dans le dernier chapitre. Ces évalutions sont effectuées sur des grappes homogènes comme hétérogènes et démontrent que notre réalisation est capable de tirer le meilleur parti de configurations matérielles réputées complexes à programmer. Nous concluons ensuite ce document et apportons des perspectives de recherches.

Chapitre 1

Les communications dans les architectures parallèles

L'exploitation des machines parallèles n'est possible actuellement que par le biais d'outils spécialisés et dont l'usage n'est pas foncièrement des plus intuitifs. Encore faut-il savoir de quelles machines il est question car les architectures sont en constante évolution, si bien que les actuelles n'ont plus forcément grand'chose à voir avec celles conçues il y a de cela deux ou trois décennies. Il en va de même pour les outils logiciels qui doivent d'adapter à ces évolutions architecturales mais également technologiques car de nouveaux paradigmes de communication apparaissent et leur exploitation demeure essentielle pour l'amélioration des performances.

Ce premier chapitre va nous permettre d'établir le cadre de notre recherche. Nous allons examiner tout d'abord les évolutions du matériel mais également des logiciels servant à son exploitation. La nature de ces logiciels est restreinte aux bibliothèques de communication. Dans un second temps, nous verrons les défis auxquels sont confrontées ces bibliothèques avant de définir précisément les cibles de notre travail, aussi bien matérielles et logicielles que fonctionnelles.

1.1 Évolution des architectures parallèles et de leurs outils d'exploitation

Cette section nous donnera l'occasion de faire un tour d'horizon des architectures actuelles pour examiner les efforts accomplis au niveau des logiciels qui évoluent simultanément.

1.1.1 Les plate-formes matérielles

Nous avons décomposé en quatre phases l'évolution des architectures parallèles qui n'est bien entendu pas achevée, mais il nous semble que les phases décrites ci-dessous reflètent correctement la situation.

1.1.1.1 Le commencement : les supercalculateurs

Les premières plate-formes possédant une architecture parallèle sont les machines désignées fréquemment par le terme de *supercalculateurs*. Il s'agit là d'un matériel très conséquent, aussi bien au niveau de la taille que des moyens nécessaires à leur exploitation. En effet, l'implantation et l'entretien d'une machine de ce type posent des problèmes logistiques concrets qui induisent un coût non négligeable en plus de la machine proprement dite, dispendieuse.

Ces machines ont un succès cyclique et il est si courant de lire dans la presse les difficultés rencontrées par tel ou tel constructeur, que l'on peut se poser la question du maintien d'une niche de ce type tant le marché paraît moribond. Cependant, au moment où nous écrivons ces lignes, les sorties de plusieurs modèles de supercalculateurs semblent indiquer un regain d'intérêt pour ces architectures : citons par exemple les Cray X-1 et RedStorm, l'*Earth Simulator* ou encore l'IBM BlueGene/L. Ce dernier est même premier du dernier classement du top 500[1] montrant bien l'enracinement des supercalculateurs (au moins de type vectoriel) dans le paysage du calcul haute-performance.

Du point de vue de l'utilisation, ces machines sont en général conçues pour des besoins applicatifs spécifiques et surtout gourmands en puissance de calcul. L'approche est de type *High Performance Computing* (HPC) : on cherche à obtenir une importante puissance de calcul sur une durée de temps limitée. Ces architectures ne sont pas aisément extensibles et lorsque les performances deviennent insuffisantes, le changement de matériel s'impose.

1.1.1.2 La succession : les réseaux de stations de travail

Le coût et la logistique nécessaires pour l'implantation d'un supercalculateur étant prohibitifs pour une majorité de laboratoires et d'universités, des solutions de repli (ou de rechange) ont été adoptées afin de disposer d'un accès à une machine parallèle. Or, les laboratoires possèdent des moyens de calcul souvent inutilisés qui sont les stations de travail. L'idée de base est la suivante : les temps de cycles inutilisés sont exploités en fédérant un ensemble de stations avec un réseau d'interconnexion (cf. Figure 1). Ce matériel étant plus standard, il est moins onéreux qu'un supercalculteur et surtout plus facilement implantable.

L'approche est quelque peu différente du cas précédent, car il s'agit d'une vision de type *High-Throughput Computing* (HTC) où la puissance de calcul désirée doit être maintenue sur une longue période de temps (plusieurs mois, voire années). Nous avons moins affaire au parallélisme qu'aux systèmes répartis, et le projet Condor ([Con]) est typique de cette approche. C'est à partir de l'exploitation des configurations de cette nature que se sont posés les enjeux du support de la gestion dynamique des processus et des applications ainsi que de la tolérance aux pannes. Ces aspects étaient le plus souvent ignorés dans la programmation des supercalculateurs, entités fiables et offrant un modèle d'éxécution statique.

[1] Novembre 2004

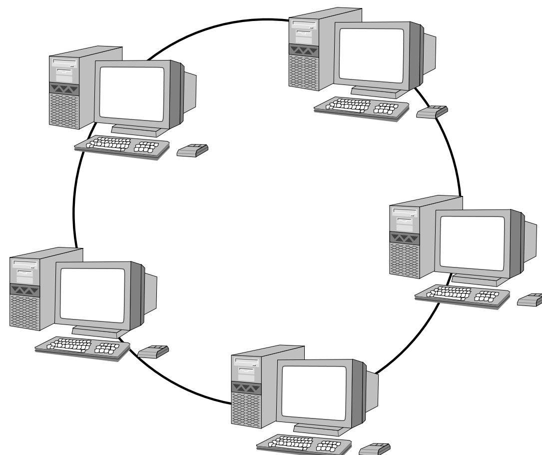


FIG. 1 – Un réseau de stations de travail

1.1.1.3 La continuation : des réseaux de stations aux grappes de PC

Les réseaux de stations de travail ont été massivement adoptés par les laboratoires, mais les réseaux d'interconnexion étaient souvent peu rapides et constituaient un goulot d'étranglement pour les performances. L'arrivée des réseaux rapides, avec une amélioration substantielle du débit (plusieurs ordres de grandeur) a bousculé cette situation et permis l'émergence d'un nouveau type d'architecture : les grappes, dont le projet NOW est un réprésentant ([NOW]).

Architecturalement, il s'agit d'une interconnexion de PC standards (encore moins chers que les stations de travail) avec un réseau haut-débit. Cet ensemble de machines est localisé dans un même lieu physique (e.g la même pièce), à la différence des réseaux de stations qui pouvaient s'étendre sur une échelle plus grande, comme un bâtiment par exemple. La figure 2 schématise une telle grappe de PC.

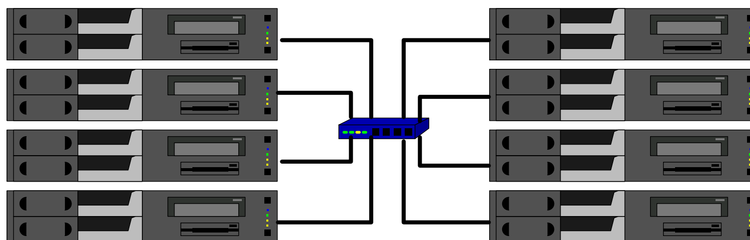


FIG. 2 – Une grappe de PC

Cette approche est un mélange des deux précédentes : les grappes sont dédiées au calcul haute-performance (HPC), mais avec des composants standards. L'extensibilité est très bonne, puisqu'il suffit de rajouter des nœuds pour augmenter la puissance de calcul. Cependant, la difficulté d'exploitation et de programmation est plus importante qu'avec les supercalculateurs car ces derniers sont équipés d'outils spécifiques. Dans le cas des grappes,

les outils d'exploitation sont souvent calqués sur ceux des supercalculateurs et la gestion dynamique des processus ou la tolérance aux pannes sont souvent reléguées au second plan. Malgré ces quelques désagréments, l'excellent *ratio* performances/prix favorise les grappes qui tendent à s'imposer : plus de la moitié des machines classées au top 500 sont des grappes.

Quelques définitions de base Nous profitons du fait que le sujet des grappes est abordé pour donner quelques éléments de vocabulaire. Il s'agit du vocabulaire couramment employé dans le document.

Définition 1 (Processus) *: Un processus est un programme en cours d'exécution (même sens que dans UNIX).*

Définition 2 (Noeud) *: une machine sur laquelle est lancé un processus. En pratique, cela correspond à un PC ou une station de travail. Un noeud peut-être multi-processeur ou uniprocesseur.*

Définition 3 (Grappe) *: ensemble de noeuds connectés par un réseau à haut-débit et sur une distance faible (dans une même pièce, par exemple)*

Définition 4 (Session) *: déroulement d'une application, et ensemble des processus exécutant cette application.*

1.1.1.4 L'évolution : les nouvelles architectures

Ce sont ces architectures de type «grappes» qui connaissent des évolutions multiples. Nous trouvons donc les catégories décrites dans les paragraphes suivants.

les grappes de grappes Cette évolution est naturelle et découle des excellentes capacités d'extensibilité des grappes. Le principe consiste en une interconnexion de plusieurs grappes, potentiellement séparées par une forte distance géographique. En quelque sorte, il s'agit d'une *méta-grappe* avec une approche plutôt de type HTC, en remplaçant les stations de travail par des unités plus importantes (les grappes). Une telle évolution doit être replacée dans le contexte du *Grid computing* et du *metacomputing*, dont le but est l'exploitation de ressources réparties. Cette agrégation pose de nouveaux problèmes car les composants sont hétérogènes : processeurs, systèmes d'exploitation et réseaux d'interconnexion varient potentiellement d'une grappe à l'autre. La nature du réseau d'interconnexion est importante car cela permet de créer des sous-classes de grappes de grappes. L'une de ces classes de grappes de grappes est très présente : les grappes faiblement couplées où les différentes grappes sont reliées par un nombre de liens tel que l'ensemble des nœuds ne forme pas un graphe complet (mais les grappes de départ continuent à l'être, cependant). Les liens peuvent être à haut-débit et potentiellement distincts de ceux utilisés à l'intérieur de ces sous-grappes (cf. Figure 3).

les grappes multiplement câblées Une autre tendance consiste à équiper une grappe avec plusieurs réseaux haut-débit. La multiplicité des technologies disponibles favorise cette situation. La grappe sera soit totalement câblée avec les multiples réseaux, soit organisée en

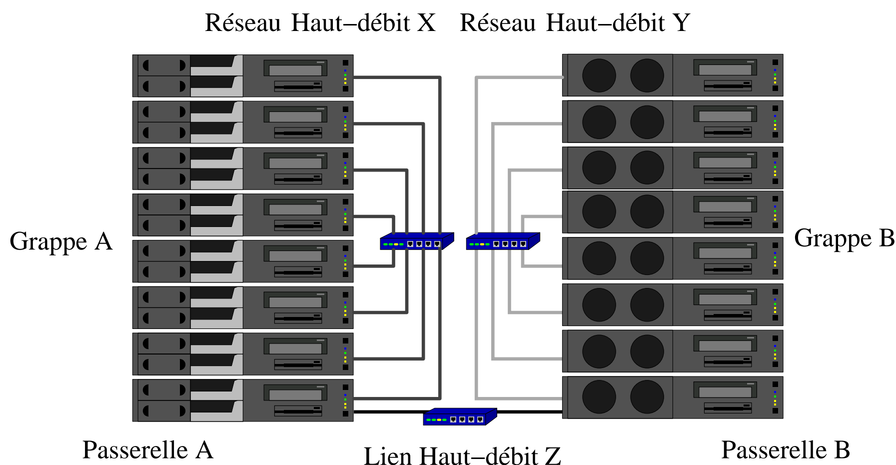


FIG. 3 – Une grappe de grappes faiblement couplées

partitions, avec un réseau haut-débit dédié à une partition particulière (cf. Figure 4). La difficulté réside alors dans la capacité du logiciel à prendre en compte ou non cette multiplicité des réseaux.

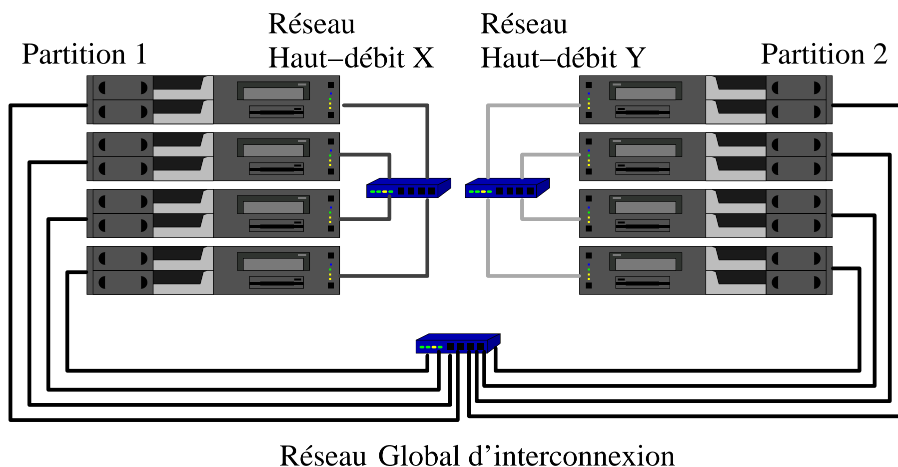


FIG. 4 – Une grappe multiplement câblée (avec des partitions)

les grappes de grande taille Comme les grappes sont facilement extensibles, il devient possible de construire des machines avec de nombreux nœuds. Si les premières générations de grappes étaient constituées par quelques dizaines de nœuds, les générations actuelles peuvent aller jusqu'à plusieurs centaines, voire milliers d'unités (par exemple, le projet Colombus de Fujitsu avec 16384 nœuds). Dans ce cas, le problème réside dans la capacité des logiciels à exploiter de telles configurations. Les mécanismes mis en place sont-ils aussi extensibles pour suivre l'évolution du matériel ?

Définitions formelles Nous introduisons ici encore les éléments de vocabulaire nécessaires à la lecture du document. D'après ce qui précède, nous aurons les définitions suivantes :

Définition 5 (Grappe hétérogène) *: grappe possédant plusieurs réseaux d'interconnexion à haut-débit qui forment des partitions de l'ensemble des noeuds.*

Définition 6 (Grappe de grappes) *: une interconnexion de grappes, soit avec un ensemble de liens à haut-débit dont le graphe de connexion ne recouvre pas l'ensemble de noeuds de la grappe de grappes, soit par un ensemble de liens à longue distance recouvrant l'ensemble de noeuds de la grappe de grappes. Si les grappes possèdent les mêmes réseaux d'interconnexion, on parlera de grappe de grappes homogène, dans le cas contraire on parlera de grappe de grappes hétérogène. La différence entre une grappe simple et une grappe de grappe réside dans l'absence de réseau haut-débit global interconnectant tous les noeuds.*

Définition 7 (Passerelle) *: dans une grappe hétérogène, on appellera passerelle un noeud appartenant à plusieurs partitions. En pratique, un passerelle possédera plusieurs cartes réseau.*

1.1.2 Les interfaces de communication pour architectures parallèles

L'évolution du matériel s'est accompagnée de celle de leurs outils d'exploitation et en particulier des interfaces et bibliothèques de communication. La mise au point de ces dernières a permis d'aboutir au constat selon lequel le goulot d'étranglement des performances se déplace du matériel vers le logiciel. Par exemple, les technologies réseau actuelles permettent d'atteindre des débits très importants, mais les couches logicielles et les protocoles de communication limitent ces possibilités.

1.1.2.1 Les interfaces de bas-niveau : transition vers le haut-débit

La recherche et le développement dans le domaine des interfaces de bas-niveau a connu – après une période de sommeil – un regain de vitalité dû à l'apparition de nouvelles technologies à haut-débit pour les réseaux. Ces derniers freinaient jusqu'ici l'amélioration des performances surtout en comparaison des processeurs. Les grappes ont le plus profité de ces nouveautés et l'apparition de technologies originales se poursuit régulièrement.

Exemples Pour ces interfaces de bas-niveau, les standards comme UDP ou TCP, longtemps utilisés, ont été remplacés par d'autres bibliothèques non-standard mais plus efficaces. L'analyse de la pile logicielle de TCP a montré que les appels-systèmes ne permettaient pas d'obtenir des performances satisfaisantes et l'arrivée de nouvelles technologies matérielles a permis de repenser les relations entre les communications et le système d'exploitation. Des solutions comme BIP ([PT98]), GAMMA ([CC97]), VIA ([vV98]), PMv2 ([TSH+00]) ou Active Messages ([vCGS92]) travaillent essentiellement en espace utilisateur, ce qui permet de se passer du système mais pose des problèmes de sécurité (accès et gestion de la mémoire). Cette multiplicité des interfaces a favorisé des efforts de standardisation avec des systèmes comme SCI ([HR99]) ou VIA. Ces initiatives se sont soldées par des échecs, relatif pour SCI (peu de constructeurs sont sur le créneau et les fonctionnalités les plus intéressantes de la

norme ne sont pas réalisées car trop coûteuses) et plus marqué pour VIA, l'effort consenti l'ayant peut-être été à trop bas niveau et l'offre matérielle demeure faible.

Idées principales Ces interfaces de bas-niveau partagent des caractéristiques de mise en œuvre : nous avons déjà indiqué que ces bibliothèques travaillent autant que possible en espace utilisateur pour recourir de façon limitée aux fonctionnalités du système d'exploitation. Ainsi, la plupart d'entre elles implémentent des modes de transfert de données de type *zéro copie*[2]. Mais ces interfaces sont souvent peu simples à utiliser et ne s'adressent pas aux programmeurs d'applications ne désirant pas se plonger dans les arcanes d'un développement à très bas niveau et dépendant d'une technologie particulière. Ces interfaces souffrent également d'un manque de portabilité et sont peu fonctionnelles : des services comme le contrôle de flux, un support pour la fiabilité ou encore le multiplexage des communications sont rarement assurés.

1.1.2.2 Les interfaces de niveau intermédiaire : vers une plus grande abstraction

Ces manques de fonctionnalités et de portabilité justifient le développement d'interfaces de niveau intermédiaire, dont l'effort d'abstraction du matériel sous-jacent autorise une utilisation plus large. C'est à ce niveau qu'apparaît la notion de modèle de programmation, avec notamment le passage de messages, la mémoire distribuée ou encore les appels de procédures à distance. Les réalisations sont nombreuses et ont connu beaucoup de succès. Citons en particulier des systèmes comme FASTMessage ([PKC97]), Madeleine ([ABD$^+$02]), VMI ([PP]) ou encore PM2 ([NM95]). Ces systèmes sont destinés soit à des développeurs d'applications, soit à des développeurs d'interfaces de plus haut-niveau. Ces interfaces masquent celles des technologies réseau sous-jacentes et offrent des services et fonctionnalités inexistants à plus bas niveau. Le nombre important de ces bibliothèques de niveau intermédiaire permet de répondre une diversification croissante des besoins.

1.1.2.3 Les interfaces de haut-niveau : un effort de standardisation

Cependant, cette multiplicité de l'offre cause des problèmes aux développeurs d'applications car ces bibliothèques ne sont pas standard. Les utilisateurs préfèrent donc se tourner vers des outils peut-être moins performants mais donnant plus de garanties de portabilité. Ces interfaces de haut niveau ne sont pas très nombreuses, mais standard.

Les acteurs principaux Les solutions agissant à un tel niveau sont principalement les trois que nous décrivont ci-dessous :

– **Parallel Virtual Machine** (PVM) est un outil de développement d'applications distribuées en environnement hétérogène. les performances ne sont pas le but de PVM, mais les services offerts contrebalançent ceci. PVM est apparu à une époque où il n'existait pas encore de standard de programmation parallèle si bien qu'il a été adopté très rapidement (car c'était l'unique outil dans son genre) pour devenir un standard *de facto*. PVM est très flexible est a été pensé en tant que système portable *et* interopérable. Les

[2] il s'agit d'un léger raccourci car il faut lire en effet zéro copie *additionnelle*

surcoûts induits par des caractéristiques telles que le support pour la dynamicité, ou la gestion des appels non-bloquants grèvent les performances suffisamment pour que l'usage de MPI se généralise ;

- **Message Passing Interface** (MPI, [DHLO$^+$96], [GLDA96] ,[GL94]) est à l'heure actuelle l'unique standard pour le développement d'applications parallèles. Il est le produit – à la différence de PVM – de réflexions entre des utilisateurs, des développeurs et des constructeurs. Ces derniers avaient en effet tendance à offrir avec leurs machines des outils de développement non portables. Ces réflexions ont commencé au début des années 90 pour aboutir à une première version vers 1993. MPI est un ensemble de spécifications et de fonctionnalités, et ne fait aucune supposition sur le matériel sous-jacent. Cette indépendance lui a assuré son succès : de multiples implémentations sont disponibles, aussi bien libres que commerciales. Les premières visent la portabilité et le support des configurations hétérogènes tandis que les secondes ont pour objectif une exploitation optimale du matériel. Cependant, la majorité des implémentations de cette interface exhibent des défauts en ce qui concerne la réactivité face aux événements réseaux. MPI est complémentaire de PVM au niveau des fonctionnalités ([LG98] ,[GKP98]) même si ce dernier a cédé du terrain. MPI est une réussite car des programmes peuvent effectivement fonctionner sur des architectures très différentes[3] tout en ayant de bonnes performances. MPI a cependant échoué sur un point, puisque deux implémentations différentes se révèlent incapables communiquer (le *MPI Paradox*), chose possible avec PVM ;
- **le standard Interoperable MPI** ([IMP00]) a été conçu pour mettre fin au *MPI Paradox*, et définit des procédures standard pour mettre en œuvre l'interopérabilité entre des implémentations différentes de MPI. Cependant, les performances ne sont pas la priorité de IMPI :

  > *« While efficient intersystem communication is important, the main performance goal of the design will be not to slow down intrasystem communication : native communication performance should not be affected by the hooks added to support interoperability, as long as there is no intersystem communication." »*

  Et la gestion des communications entre deux implémentations différentes vient confirmer ceci (cf. 2.1.3.3). En pratique, IMPI n'est pas très répandu et est soutenu par une des deux implémentations libres de MPI, à savoir LAM/MPI.

Portabilité ou interopérabilité ? L'examen des interfaces existantes permet d'établir les objectifs de chacune des solutions en ce qui concerne la portabilité, les performances et l'interopérabilité. Ainsi, nous pouvons affirmer que :

- PVM est portable, interopérable mais obsolète ;
- MPI est portable et assure une portabilité des performances ;
- IMPI permet une véritable interopérabilité dans MPI.

[3]modulo une recompilation

1.1.2.4 Conclusion

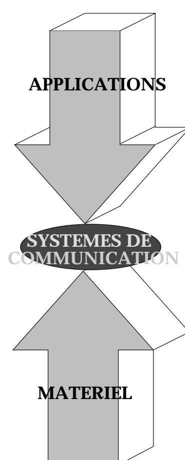
D'abord, les interfaces de bas niveau ne sont pas assez mûres pour être adoptées largement par les développeurs d'applications. La complexité intrinsèque et le manque de portabilité sont les principaux responsables.

Ensuite, les interfaces de niveau intermédiaire possèdent le défaut de ne pas être standard pour être massivement utilisées dans le développement d'applications. En revanche, elle peuvent jouer un rôle central dans la conception d'intergiciels (*middleware*) de niveau supérieur, et notamment MPI.

Enfin, MPI est le seul standard disponible à l'heure actuelle et travailler à ce niveau permet l'accès à une pléthore d'applications et une diffusion plus large et plus rapide du travail effectué, car les utilisateurs ne se poseront pas la question de la migration d'un outil vers un autre. la perte de performances probable est négligeable de toutes façons, les implémentations de MPI au-dessus de réseaux comme Quadrics ou Infiniband le prouvent amplement (cf. [LCW+03], [LJW+04]).

1.1.3 Vers des configurations hiérarchiques, hétérogènes et dynamiques

L'examen des architectures matérielles et de leurs outils d'exploitation permet de dessiner la tendance vers laquelle nous nous acheminons. Elle est conditionnée par deux contraintes : une première matérielle et une seconde applicative. Les systèmes de communication sont pris dans cet étau et doivent s'en accomoder.



1.1.3.1 Tendance actuelle

La tendance est donc encadrée par ces deux contraintes :

- d'un côté les architectures sont de plus en plus complexes et difficiles à programmer. Elles sont naturellement hétérogènes, et les communications doivent être organisées en niveaux correspondants à la topologie (hiérarchie) sous-jacente ;

- d'un autre côté, les applications ont de nouveaux besoins dans les domaines de la collaboration (*computational steering*), outre ceux habituels de puissance de calcul et de communications rapides. De telles applications ont en particulier des exigeances au niveau de la gestion dynamique des tâches.

Nous définissons à présent les termes centraux employés dans ce document et précisons les idées et concepts que nous leur associons. Nous allons établir la correspondance entre les propriétés de la configuration (à la fois matérielle, logicielle et applicative) et les propriétés recherchées au niveau des systèmes de communication pour prendre en compte ces caractéristiques. Dans les sections suivantes, nous allons préciser les notions de hiérarchie, d'hétérogénéité et de dynamicité.

1.1.3.2 La notion de hiérarchie dans les communications

La première notion que nous souhaitons préciser est celle de la hiérarchie. Nous allons donner plusieurs définitions puis indiquer celle retenue et utilisée dans ce document ainsi qu'une classification permettant de mieux en appréhender le sens. Pour finir, nous ferons le lien entre la propriété retenue et les fonctionnalités recherchées au sein du système de communication.

Différents aspects de la hiérarchie La notion de hiérarchie est fluctuante mais peut être définie comme l'ensemble des niveaux de communication départagés par différentes métriques basées sur la distance entre les deux entités communiquantes, ou encore la vitesse des communications. Ces niveaux se superposent et le support d'un niveau donné implique le support de ceux qui lui sont inférieurs. Nous considérons des *configurations* hiérarchiques, mais nos communications ne le sont *pas*, ce qui signifie qu'un message transitant à niveau donné de l'échelle ne passe pas obligatoirement par d'autres que le sien.

La hiérarchie dans la configuration Dans notre cas, nous allons adopter la définition suivante :

Définition 8 (Hiérarchique) *: Une configuration sera déclarée* **hiérarchique** *s'il est possible de lui faire correspondre une échelle de classification des communications. Cette échelle sera basée sur la distance existante entre les nœuds participant à cette communication ainsi que sur leur appartenance à des mêmes entités physiques (machines, grappes, etc.)*

Les différents niveaux de la hiérarchie Jusqu'ici, nous avons mentionné les niveaux de hiérarchie sans les définir précisément. À la lumière de ce qui précède, nous pouvons établir un classement des différents schémas de communication au sein d'une configuration matérielle. La classification pourra être basée sur une métrique de type *vitesse* ou *distance* et cette hiérarchie inclue également la hiérarchie mémoire. Pour les grappes de grappes de machines multi-processeurs, nous allons finalement considérer les communications suivantes :

- les communications de type intra-nœud : dans ce cas, la hiérarchie mémoire se confond avec la hiérarchie des communications. Il s'agit d'un cas particulier concernant la gestion des machines multi-processeurs ;

– les communications de type inter-nœud : il s'agit de tous les niveaux pour gérer *effectivement* les configurations de type grappe de grappes. Cela regroupe essentiellement les communications intra- et inter-grappes.

Les niveaux correspondent bien à la portée des communications et vont du plus local au plus global. Un système de communication destiné à l'exploitation des grappes de grappes supportera toujours les niveaux inter-nœud et intra-grappe (cf 1.2.3.1).

Propriétés correspondantes du système de communication Finalement, les propriétés correspondantes pour le système de communication seront les suivantes :

Définition 9 (Multi-échelle) *: Un système de communication sera qualifié de* **multi-échelles** *s'il possède la capacité de gérer la hiérarchie de la configuration qu'il exploite.*

Définition 10 (Multi-grappes) *: Un système de communication sera déclaré* **multi-grappes** *s'il est multi-échelles et qu'il supporte le niveau de communication inter-grappes.*

1.1.3.3 La notion d'hétérogénéité

Nous procédons de la même façon pour traiter la notion d'hétérogénéité. Nous allons cependant constater que cette dernière possède une éventail plus large d'acceptions.

Différents aspects de l'hétérogénéité La notion d'hétérogénéité est également variable et nous avons isolé les sens suivants :

– l'hétérogénéité peut être celle des processeurs des machines. Dans ce cas, il faut faire la distinction entre une hétérogénéité logique et une hétérogénéité technologique. Dans le premier cas, deux processeurs avec des jeux d'instructions différents seront hétérogènes tandis que dans le second cas, des processeurs cadencés à des vitesses différentes seront considérés comme hétérogènes ;
– l'hétérogénéité peut faire référence aux réseaux utilisés. Là encore, il faut distinguer le cas logique du cas technologique. Dans le second cas, des générations différentes d'un même matériel, des interfaces distinctes ou des modèles de programmation dissemblables confèrent un caractère hétérogène aux réseaux concernés ;

Outre ces définitions, nous insistons sur la distinction nécessaire à établir entre le *support* de l'hétérogénéité et son *exploitation*, en général effectuée par des systèmes qui la masquent au niveau supérieur. C'est par exemple le cas des systèmes multirails qui mettent en place des services comme de la fragmentation ou de la tolérance aux pannes. Dans un tel cas, le *support* de l'hétérogénéité permet son *exploitation* effective.

Enfin, l'hétérogénéité aussi possède ses niveaux d'action. Par exemple, si une couche de communication permet de gérer des réseaux hétérogènes en établissant un réseau virtuellement homogène (avec des mécanismes de retransmission par exemple), alors cette couche est aussi considérée comme hétérogène. Mais si un système de communication est bâtit au-dessus d'une telle couche, alors il devient homogène, car il ne "voit" qu'un réseau virtuellement homogène, quand bien même ce dernier est physiquement hétérogène.

L'hétérogénéité dans la configuration Dans notre cas, nous nous restreignons à une hétérogénéité des réseaux. D'où la définition suivante :

Définition 11 (Hétérogène) *: Une configuration sera déclarée* **hétérogène** *si elle présente un ensemble de réseaux différents. Ces différences peuvent s'exprimer du point de vue de la technologie, de la génération du matériel ou encore de l'interface de programmation employée pour son exploitation.*

Propriétés correspondantes du système de communication Les propriétés recherchées seront donc les suivantes :

Définition 12 (Multi-protocoles) *: Un système de communication est* **multi-protocoles** *s'il possède la capacité de gérer simultanément plusieurs protocoles de communication. Ces protocoles peuvent être distincts ou bien différentes instances d'un unique protocole .*

Il est possible d'affiner cette définition :

Définition 13 (Multi-réseaux) *: Un système de communication multi-protocoles peut être qualifié de* **multi-réseaux** *s'il possède la capacité de gérer simultanément plusieurs protocoles réseaux au sein d'un même niveau de la hiérarchie des communications.*

1.1.3.4 La notion de dynamicité

Le dernier aspect que nous traiterons sera la dynamicité. Là encore la multiplicité des définitions nous conduit à restreindre le sens du mot.

Différents aspects de la dynamicité Quels sont les sens les plus fréquents pour le terme "dynamicité" ?

- le premier sera celui associé à l'exécution des programmes et quand le nombre de tâches varie ;
- le second sens est une extension du premier, mais avec une échelle plus large. Nous allons parler de sessions dynamiques où l'on cherche à rajouter ou à enlever un ensemble de tâches (i.e. une session) à une autre pré-existante. Deux sessions indépendantes peuvent ainsi fusionner pour ne former qu'une unique session pendant un certain temps puis se séparer à nouveau avant de terminer leur exécution ;
- le troisième sens sera à prendre dans un contexte de déploiement de services en fonction de la configuration et pendant l'exécution (eg. support des machines SMP, tolérance aux pannes, etc). Un tel déploiement pourra être qualifié aussi de dynamique ;
- le dernier sens sera lié au terme de volatilité. Ce terme est employé dans le cas où un nombre de tâches disparaissant/réintégrées est important dans un quantum de temps limité. Par exemple, un système gérant des pannes en rafales sera qualifié de volatile. De ce point de vue, la volatilité peut être considérée comme un stade supérieur de la dynamicité.

Les deux définitions de base associées à la gestion dynamique des processus seront les suivantes :

Définition 14 (statique) *: Un ensemble de tâches est dit statique si ces dernières sont créées uniquement au lancement du programme*

Définition 15 (dynamique) *: Un ensemble de tâches est dit dynamique si ces dernières sont créées ou détruites à n'importe quel moment de l'exécution*

Propriété correspondante du système de communication Enfin, La propriété correspondante pour le système de communication est la suivante :

Définition 16 (Gestion dynamique des processus) *: Un système de communication est* **compatible avec la gestion dynamique des processus** *s'il est capable de gérer une modification du nombre de processus participant à l'application pendant l'exécution de cette dernière.*

Plus précisément :

Définition 17 (Multi-sessions) *: Un système de communication sera declaré* **multi-sessions** *s'il est capable de gérer des sessions (fusion et séparation des groupes de processus les constituant) et ce au cours de l'exécution des applications concernées.*

1.1.3.5 Cadre de la thèse

À la lumière des définitions données dans cette première partie, nous pouvons enfin préciser le cadre dans lequel s'inscrivent nos recherches. Nos cibles matérielles seront donc les grappes de grappes, avec les variantes mentionnées. Ces configurations sont intrinsèquement *hiérarchiques* (propriété recherchée pour le système de communication : *multi-échelle*), et possèdent des réseaux d'interconnexion multiples et *hétérogènes* (propriété recherchée : *multi-réseau*). Les applications visés pour l'exploitation de ces grappes de grappes seront potentiellement *dynamiques* (propriété recherchée : *multi-session*)

1.2 Les nouveaux défis des bibliothèques de communication hautes-performances

Lorsque nous avons cadré notre travail, nous avons établi en fait toute une série de défis auxquels sont confrontées les bibliothèques de communication.

1.2.1 Introduction

Les défis que les bibliothèques de communication doivent relever sont multiples et concernent des aspects aussi bien matériels que logiciels. Ainsi, nous avons isolé les points suivants comme essentiels pour les bibliothèques de communication :

- le support des différents types de grappes, (cf. 1.1.1.4) ;
- le support de l'échelle des communications dans les grappes et les grappes de grappes (cf. 1.1.3.2) ;
- le support de l'hétérogénéité dans les grappes et les grappes de grappes ;

– le support des configurations et applications dynamiques dans les grappes et grappes de grappes ;

Pistes actuellement suivies Ces défis, en termes de fonctionnalités ou de support, sont bien entendu au cœur de nombreuses recherches mais nous pouvons toutefois distinguer deux grandes tendances.

La première tendance est celle consistant à offrir un *support simple* pour ces nouvelles fonctionnalités. Les outils résultants doivent permettre une exploitation efficace mais simple des topologies hétérogènes par des applications dynamiques. Dans ce cas, l'utilisateur conserve un contrôle total sur ce qu'il fait et le système entérine ses choix.

La seconde tendance est celle de l'*exploitation* des configurations hétérogènes et dans ce cas, les services offerts sont plus perfectionnés que dans le cas précédent : il pourra s'agir par exemple de détection et de tolérance aux pannes avec changement dynamique et transparent du réseau utilisé, de fragmentation des données sur plusieurs liens parallèles (homogènes comme hétérogènes), ou encore de la répartition des données selon la charge de ces différents liens. Dans un tel cas de figure, la topologie est masquée à l'utilisateur mais exploitée pour assurer une haute qualité de service. C'est par exemple le cas des systèmes dits *multirails* ([PBH99], [CFP$^+$01]), dont le but est de permettre l'exécution d'une application coûte-que-coûte et pour lesquels les multiples réseaux sont considérés comme un moyen d'assurer une certaine fiabilité de fonctionnement.

Enfin, nous remarquons qu'en ce qui concerne les problèmes de l'hétérogénéité et de la dynamicité, beaucoup de solutions proposées utilisent des schémas de type inter-opérables. Si cela est une solution au problème, il est néanmoins possible de se passer de tels mécanismes, généralement coûteux et pas suffisament appropriés au contexte des grappes de grappes.

1.2.2 L'exploitation des grappes de PC

Le premier des défis pour les bibliothèques de communication est d'être capable d'exploiter efficacement les architectures de type grappes de PC.

1.2.2.1 Rappel : les raisons du succès des architectures de type grappes

Ces architectures sont couramment employées pour de nombreuses raisons que nous rappelons ici et qui sont bien souvent d'ordre économique.

La principale raison est le faible coût de ces machines en comparaison de la puissance délivrée : le rapport est plus favorable aux grappes qu'aux supercalculateurs[4]. Ce coût a permis l'implantation de grappes dans de nombreux laboratoires et cette large diffusion explique pour partie l'intérêt de la communauté scientifique.

Ensuite, il est indéniable que la mise en service d'une grappe, même de taille respectable est plus aisée en termes d'espace requis ou d'entretien car le matériel est standard. Les grappes sont de plus très facilement extensibles, à la différence des supercalculateurs.

[4]certains constructeurs de supercalculateurs contestent ceci avec des arguments comme les coûts de programmation ...

Enfin, ce succès prend également sa source dans la diversité des technologies disponibles pour les architectures des processeurs et des réseaux (arrivée du haut-débit).

1.2.2.2 Enjeux et difficultés

Cependant, en dépit de ces avantages, les grappes présentent des aspects plus problématiques dont le premier d'entre eux est la difficulté intrinsèque de programmation. En effet, une exploitation efficace du matériel disponible est souvent une gageure car les nœuds peuvent être multi-processeurs, voire intégrer du multithreading. De plus, les outils logiciels disponibles se veulent souvent génériques et portables (à cause de la diversité du matériel, justement), et le niveau de performances est inférieur à celui d'un supercalculateur qui peut se permettre d'offir des outils très spécialisés et non portables.

Ensuite, la notion d'échelle commence à intervenir avec ces architectures car la taille des grappes est très variable et les logiciels doivent être capables de répondre à ce problème, notamment au niveau de la gestion des communications. En effet, la prise en compte de la hiérarchie permet une meilleure exploitation de l'infrastructure sous-jacente.

Enfin, un dernier point difficile est celui du couplage des grappes et des applications. L'extensiblité des architectures de type grappe conduit naturellement à vouloir passer à une échelle supérieure en interconnectant plusieurs de ces entités ce qui permet de fédérer des ressources de calcul mais pose de nouveaux problèmes. Au niveau applicatif, si des applications s'exécutent sur des grappes différentes, il serait naturel de pouvoir les faire fusionner et les systèmes de communication doivent une fois encore s'adapter au caractère hétérogène et dynamique d'une telle configuration.

1.2.3 La gestion des configurations hiérarchiques

De même que l'introduction des grappes a nécessité une adaptation des bibliothèques de communication, l'arrivée des grappes de grappes a entraîné un changement dans la façon d'aborder leur exploitation. En effet, le caractère hiérarchique s'est accru de paire avec les types de communications possibles. Cela constitue le second défi que ces bibliothèques doivent relever.

1.2.3.1 Retour sur la hiérarchie des communications

Nous reprécisons maintenant la classification vue en 1.1.3.2 pour l' affiner. Les catégories de communications considérées sont donc les suivantes.

Les communications intra-nœuds Les premières communications considérées sont celles se produisant à l'intérieur d'un même nœud de calcul. Nous avons donc :

- les communications intra-processus, qui sont un cas très particulier. Cela ne constitue pas un réel niveau de communication, mais doit plutôt être vu comme un optimisation (en pratique, ces communications reposent souvent sur des opérations mémoire de type `memcpy`) ;

– les communications intra-nœuds dans le cas de machines multi-processeurs, qui sont optionnelles dans la mesure où toutes les grappes ne sont pas construites avec de telles entités ;
– d'autres communications qui pourraient voir le jour, comme les communications survenant à l'intérieur d'une machine de type NUMA (*Non Uniform Memory Access*).

Les communications inter-nœuds Viennent ensuite les communications se produisant entre deux nœuds distincts. Plus de niveaux sont à considérer que précédemment :
– le premier niveau est celui des communications intra-grappe. C'est à ce niveau que nous cherchons à obtenir des performances optimales. Dans le cas d'une grappe multiplement câblée avec présence d'un réseau global et de réseaux locaux d'interconnexion, les communications survenant dans les partitions formées par ces réseaux sont encore considérées comme faisant partie de de niveau ;
– le deuxième niveau est celui des communications se produisant à l'intérieur d'une grappe de grappes. Ce niveau inclue évidemment le précédent et nous cherchons une fois de plus à obtenir des communications optimales pour ce type de communications ;
– les communications d'un niveau supérieur, c'est-à-dire à longue distance, entre plusieurs grappes de grappes par exemple. Ce sont les communications rencontrées dans les grilles de calcul et que nous ne traiterons pas. Notre souci sera tout de même d'assurer une compatibilité entre notre travail et les efforts existants à ce niveau.

Nous remarquons que les niveaux se superposent, sauf dans le cas des communications inter-nœuds : il est ainsi tout-à-fait possible qu'une bibliothèque gère les communications inter-nœuds sans s'occuper des communications intra-nœuds (cas des systèmes destinés aux grappes homogènes de machines uniprocesseurs par exemple).

Nous répétons que nous cherchons à avoir un support optimal du niveau grappes de grappes sans aller au-delà, mais nous gardons à l'esprit qu'un niveau supérieur existe (ou pourra exister au sein d'une configuration donnée). Enfin, cette liste est ouverte et pourra être étendue dans le cas d'évolutions technologiques et/ou architecturales (par exemple, passage de nœuds multi-processeurs à des nœuds NUMA).

Une caractérisation des communications inter-grappes Les communications inter-grappes peuvent à leur tour êtres classées en deux catégories selon la méthode choisie par la bibliothèque de communication pour les réaliser. Nous aurons donc :
– les communications inter-grappes *directes* : dans un tel cas de figure, l'émetteur d'un message l'envoie directement au récepteur, sans passer par un processus intermédiaire, comme un routeur par exemple. Un tel mode de communication empêche donc d'exploiter les liens haut-débit dans le cas de grappes faiblement couplées ;
– les communications inter-grappes avec *retransmission(s)* : elles reposent sur des processus-routeurs s'exécutant sur des machines jouant le rôle de passerelles et équipées de plusieurs technologies réseaux différentes. Ce type de communications est utilisé lorsque la taille des grappes empêche d'avoir des connexions entre tous les processus (limitation de ressources). Les routeurs et passerelles constituent des goulots d'étranglement si les communications inter-grappes sont nombreuses et de plus, les

mécanismes de retransmission sont coûteux. Enfin, une question non triviale est celle du rôle de ces processus-routeurs : doivent-ils être exclus du calcul ou bien doivent-ils assurer le service de retransmission en plus de l'exécution de l'application ?

1.2.3.2 Enjeux et difficultés

La difficulté majeure réside dans la capacité de pouvoir rajouter arbitrairement des niveaux dans la hiérarchie de sorte que l'édifice architectural ne s'effondre pas, c'est-à-dire que le niveau des performances pour un niveau donné soit encore élevé.

Un second enjeu est celui du contrôle de la hiérarchie par l'utilisateur et deux choix sont possibles : d'un côté, le système peut cacher à l'application la topologie sous-jacente avec les différentes grappes, de façon à présenter une vision unifiée et homogène. Ainsi, les applications développées au-dessus de telles bibliothèques n'ont aucun moyen de savoir qu'elles vont être exécutées sur une configuration très hiérarchique et ne peuvent pas adapter leurs schémas de communication dans une perspective d'optimisation. C'est à la bibliothèque de communication de faire de telles optimisations. L'autre solution procède inversement en offrant à l'application une vision totale de la hiérarchie sous-jacente. Dans ce cas, le développement d'applications doit explicitement intégrer cette caractéristique, ce qui complique la tâche mais permet une réelle adaptation de l'application à son environnement.

1.2.4 Le support des réseaux hétérogènes

Le second défi des bibliothèques de communication est d'offrir un support pour les réseaux hétérogènes pour permettre une exploitation optimale du matériel disponible.

1.2.4.1 Problématique

L'interconnexion de plusieurs grappes équipées de matériels distincts conduit à la création d'une entité intrinsèquement hétérogène. Il est donc nécessaire d'arriver à exploiter les réseaux haut-débit locaux pour conserver les performances des grappes individuelles. Cette situation se complexifie d'autant plus que les liens connectant ces grappes peuvent eux-mêmes reposer sur des technologies différentes. De plus, les interconnexions peuvent suivre des schémas complexes comme dans le cas où les grappes forment une composante connexe et non un graphe complet. Il devient alors essentiel de pouvoir retransmettre les messages avec des passerelles haut-débit.

1.2.4.2 Enjeux et difficultés

La première difficulté est celle du surcoût induit par l'introduction du support de l'hétérogénéité dans la bibliothèque de communication : si cette dernière est capable d'exploiter des configurations complexes, le support d'environnements matériels plus simples (par exemple, les grappes homogènes de machines uniprocesseurs) devrait demeurer efficace.

L'exploitation efficace des machines jouant le rôle de passerelles est également un enjeu important pour conserver un haut débit pour les communications inter-grappes. Les

bibliothèques de communication doivent être capables d'offrir un service de routage et de retransmission des messages d'un réseau vers un autre. Les politiques de routage doivent également faire l'objet d'un soin tout particulier, et être modifiée en fonction de la répartition des communications. Elle pourra également être adaptative, c'est-à-dire que les routes pourront être régulièrement recalculées selon des critères de charge des liens ou des passerelles. Un tel recalcul étant potentiellement long et donc source de dégradation des performances, la fréquence doit être judicieusement choisie.

Enfin, les bibliothèques de communication doivent éventuellement offrir en plus du support pour les réseaux hétérogènes une exploitation des différents liens disponibles entre deux nœuds avec une fragmentation des messages par exemple. Ainsi, si deux processus communiquent et que deux réseaux sont disponibles (e.g Myrinet et SCI), alors les messages devraient être divisés en morceaux, qui seraient alors émis en parallèle sur les différents réseaux physiques avant d'être recomposés à l'arrivée. Cette technique (*channel bonding*) permet d'agréger le débit des ressources réseaux disponibles.

1.2.5 Le support des configurations et applications dynamiques

Le troisième défi que nous abordons est celui des environnements dynamiques, ce qui recouvre à la fois des aspects matériels mais aussi logiciels.

1.2.5.1 Problématique

L'interconnexion de différentes grappes peut s'effectuer au niveau matériel (on rajoute un câble) mais également logiciel, avec les applications exploitant ces grappes fusionnant pour ne former plus qu'une application. Il est également très probable que cette dernière souhaite revenir à la situation de départ ce qui revient à scinder une application en plusieurs morceaux. La bibliothèque de communication doit donc être capable de gérer l'intégration de processus supplémentaires dans la configuration d'origine. La dynamicité intervient également dans le cas des grappes de grande taille car la multiplicité du nombre de nœuds de calcul implique que la probabilité d'une défaillance de leur part augmente conséquemment.

1.2.5.2 Enjeux et difficultés

Le problème principal est encore lié à l'intégration de nouvelles fonctionnalités dans les bibliothèques de communication. De même que le passage d'un support uniprotocole à multiprotocole ne doit pas avoir des répercussions trop importantes pour les performances, la migration d'une configuration statique vers un modèle dynamique doit se faire en tenant compte des fonctionnalités déjà introduites. Donc, les performances en environnement dynamique doivent être très peu éloignées de celle obtenues en environnement statique. Également, comme nous nous situons dans un cadre potentiellement hétérogène, la dynamicité a des conséquences sur les mécanismes mis en place comme le routage : les routes déterminées au départ peuvent devenir obsolètes et doivent être recalculées.

Le fait d'avoir une gestion dynamique des tâches doit permettre de mettre en place des mécanismes de tolérance aux pannes (si toutefois la détection des pannes existe). Ceci implique de décider d'une politique de gestion de la numérotation : autorise t'on des numéros

non consécutifs ou non ?, la défaillance d'un processus est-elle un danger pour le comportement de la bibliothèque de communication ?, etc. De plus la bibliothèque de communication doit-elle juste détecter ces défaillances et les reporter aux applications ou bien va-t'elle continuer son exécution et offrir ainsi aux couches supérieures l'illusion d'un comportement régulier et sans défaillances ?

1.3 Nos objectifs

Quels sont finalement les objectifs que nous poursuivons ? En effet, il y a énormément d'aspects à prendre en compte, à un tel point qu'il devient légitime de se demander si tout cela demeure encore compatible avec la haute-performance. C'est justement le but de notre travail, c'est-à-dire d'intégrer les divers mécanismes évoqués sans pour autant sacrifier les performances par rapport à la multiplicité des services. Cette intégration doit conduire à des solutions à la fois simples, flexibles et complètes. L'autre question fondamentale est celle de l'adéquation des systèmes actuels vis-à-vis de ces défis : y a-t'il des bibliothèques de communication répondant à tous ces besoins ?

1.3.1 Définitions des services recherchés

les caractéristiques pour lesquelles nous voulons obtenir un support sont les suivantes :

- premier besoin : un support de type multi-échelle et en particulier du niveau multi-grappe ;
- deuxième besoin : un support de type multi-réseaux et en particulier dans une approche pour l'hétérogénéité ;
- troisième besoin : un support de type multi-sessions pour la gestion dynamique des processus mais sans impact pour les performances si possible ;
- quatrième besoin : des capacités d'extensibilité car il y a nécessité de prévoir un cadre de travail plus large que celui d'origine ;
- cinquième besoin : nous voulons obtenir une portabilité des performances, et ce quel que soit le niveau considéré de l'échelle.

Récapitulatif : Nous récapitulons maintenant les correspondances que nous avons établies en ce qui concerne les propriétés des configurations, celles du système de communication et celles que nous cherchons à intégrer. Ces correspondances sont résumées dans le tableau ci-dessous :

Propriété de la configuration	Propriété correspondante du système	Propriété recherchée
Hiérarchique	Multi-échelles	**Multi-grappes**
Hétérogène	Multi-protocoles	**Multi-réseaux**
Dynamique	Gestion dynamique des processus	**Multi-sessions**

Enfin, notre démarche de travail sera la suivante :

Nous allons adapter une implémentation du standard MPI pour obtenir un système de communication offrant les services recherchés. Le choix de MPI est motivé par le fait qu'il s'agit d'un standard et que les performances sont déterminantes. De plus, nous aurons une panoplie d'applications déjà disponibles car MPI est l'outil le plus utilisé pour la programmation d'application parallèles.

1.3.2 Cahier des charges

Nous précisons enfin le cahier des charges pour la réalisation de ce travail. Du point de vue des performances, il est essentiel pour nous de ne pas sacrifier un niveau de la hiérarchie et donc nous voulons les meilleures performances possibles pour les communications intra-grappes et inter-grappes. Ces communications doivent utiliser les ressources technologiques disponibles du mieux possible.

Ensuite, il est indispensable que la puissance de calcul soit conservée (ie la moins dégradée) pour que le système soit utilisable en pratique et ne reste pas au stade de prototype ou d'étude de faisabilité pour les idées développées dans la cadre de cette thèse.

Enfin, nous devons rester dans le cadre du standard c'est-à-dire introduire ces nouvelles fonctionnalités sans mettre à mal les efforts de standardisation déjà consentis auparavant.

Chapitre 2

Un état de l'art

Depuis la publication du standard MPI (au début des années 90), le nombre d'articles et de livres consacrés au sujet est tel qu'il semble très difficile d'effectuer un recensement exhaustif des projets de recherche ou des systèmes commerciaux liés à MPI. Dans cet état de l'art, nous nous sommes surtout concentrés sur les solutions du domaine libre, pour lesquelles de la documentation existe et où les sources des logiciels sont disponibles et évaluables. La conséquence de cette décision est que nous avons écarté un certain nombre de solutions commerciales destinées à des machines particulières. Nous nous sommes efforcés de les citer néanmoins, mais l'absence de documentation qui ne se réduise pas à de la simple réclame rend le travail critique indispensable très difficile, voire impossible.

L'organisation de ce chapitre est la suivante : nous allons aborder un ensemble de systèmes existants (aussi bien actuels qu'un peu plus anciens) et qui s'attachent à proposer une solution au double problème de la gestion de la hiérarchie et de l'hétérogénéité. Le choix de lier ces deux aspects est délibéré car ils constituent selon nous deux instances d'un problème plus global, à savoir la gestion de multiples protocoles de communication dans MPI. Nous expliciterons ensuite le fonctionnement des solutions autorisant une gestion dynamique des processus. Nous avons opté pour un tel schéma organisationnel car il apparaît qu'en règle générale les implémentations de MPI solutionnent soit un problème (l'hétérogénéité), soit l'autre (la dynamicité), exceptionnellement les deux. Cependant, il convient de noter d'ores et déjà qu'existent des mises en œuvre du standard prenant en compte conjointement ces deux aspects.

La dernière section de ce chapitre dresse un bilan de l'existant et présente un tableau récapitulatif synthétique des solutions exposées dans cet état de l'art.

2.1 Architectures supportant hiérarchie et hétérogénéité

Cette section aborde les architectures des solutions offrant un support pour les machines hiérarchiques et hétérogènes dans MPI.

2.1.1 Deux propriétés pour une unique problématique

Le support des systèmes hiérarchiques dans MPI est un problème qui n'est pas récent. En effet, quand les architectures de type grappes sont apparues, l'idée de substituer les nœuds constitués de simples PC uniprocesseurs par un matériel plus sophistiqué – notamment par des machines multiprocesseurs – a rapidement fait surface. L'emploi de ce type de machines a eu une conséquence importante sur les logiciels parce qu'introduisant une certaine asymétrie dans les communications dont il fallait bien tenir compte pour conserver de bonnes performances. Ceci est en particulier vrai pour les implémentations de MPI, qui doivent alors gérer à la fois un protocole réseau (pour les communications inter-nœuds) de concert avec un second protocole (pour les communications intra-nœuds) en général basé sur des systèmes de mémoire partagée.

L'apparition des architectures de type «grappes de grappes» n'a fait que redonner de l'actualité et de l'ampleur à ces problèmes : la notion de hiérarchie refait surface (communications intra-grappes et communications inter-grappes), tout comme la gestion des multiples protocoles (les différentes grappes possèdent peut-être des réseaux d'interconnexions différents). Par conséquent, les deux problèmes de la gestion de la hiérarchie et de l'hétérogénéité des réseaux sont regroupés parce qu'ils peuvent être résolus avec des techniques et architectures logicielles identiques. Conceptuellement, il n'y a aucune différence entre une implémentation de MPI destinée à des grappes homogènes de machines multiprocesseurs et une implémentation de MPI destinée à une grappe de grappes hétérogènes de machines uniprocesseurs : les contraintes étant rigoureusement identiques, les solutions peuvent alors être similaires.

Cette partie de l'état de l'art reflète la récurrence de ce problème ainsi que cette évolution chronologico-technologique des architectures de type "grappes". Nous avons essayé d'éviter de faire un catalogue des solutions existantes, l'approche retenue étant plutôt celle de la classification des solutions fondée sur les architectures mises en place. Ces dernières présentent de subtiles différences et constituent des variations d'un même patron architectural. Nous avons ainsi isolé trois «grandes» familles de solutions :

- en premier lieu les solutions destinées aux cas particuliers que constituent les grappes de machines multi-processeurs ;
- ensuite les solutions plus génériques qui tentent de résoudre les problèmes de la gestion de hiérarchie et de l'hétérogénéité en utilisant des mécanismes d'inter-opérabilité (à un niveau assez haut, donc) ;
- enfin, la famille des solutions qui se proposent de traiter le problème avec une intégration des mécanismes dans MPI directement (à un niveau plus bas que dans l'approche précédente).

Ces différentes familles seront détaillées dans le reste de cette section.

2.1.2 Le cas particulier des grappes de machines multiprocesseurs

Ainsi que nous l'avons évoqué en introduction (c.f 2.1.1), la difficulté rencontrée lors de l'exploitation des grappes de machines multi-processeurs est la gestion la plus efficace possible des communications intra- et inter-nœuds. Il est nécessaire de prendre en compte la multiplicité de ces schémas ainsi que leurs différences (un protocole réseau et un protocole

spécifique aux communications intra-nœuds par exemple). Pour ce faire, deux approches ont été retenues :

- la première approche peut être qualifiée d'*hybride*, car le problème de la gestion de la hiérarchie est solutionné par l'emploi de deux outils de programmation. MPI est utilisé pour les communications inter-nœuds et un second outil (jugé plus adéquat que MPI) est employé pour exploiter les nœuds multi-processeurs. Cette exploitation passe par le biais d'optimisations des communications intra-nœuds ou par la réorganisation des tâches de calcul sur ce nœud. Cette alternative possède un caractère moins générique que la solution fondée sur les communications mais mérite d'être signalée : nous l'évoquerons succintement par la suite ;
- la seconde approche, plus triviale, repose sur l'emploi exclusif d'une bibliothèque MPI. C'est donc au sein même de l'implémentation que la gestion de la hiérarchie est mise en place.

Nous détaillons à présent ces deux types d'approches.

2.1.2.1 Les solutions hybrides

Dans un premier temps, nous examinons les solutions hybrides, en détaillant leur principe de fonctionnement, avant de discuter des avantages et limitations d'une telle méthode. Pour finir nous donnons quelques exemples de réalisations instanciant cette approche.

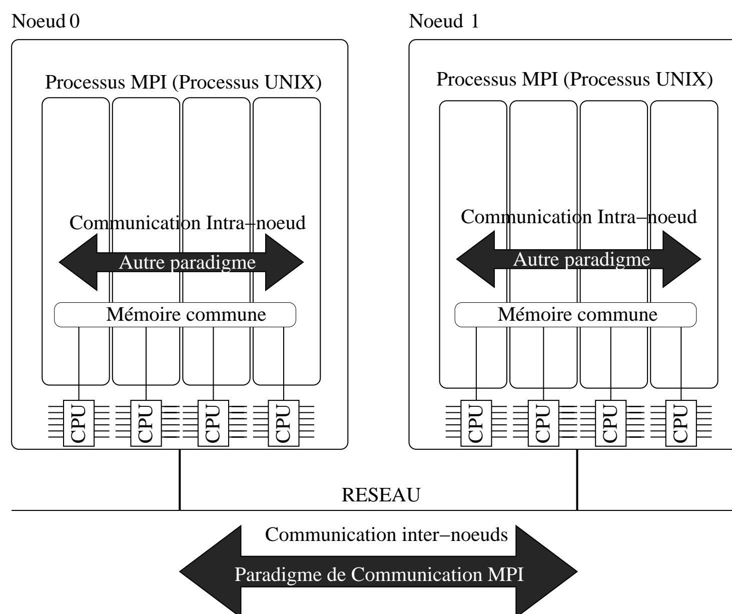


FIG. 5 – Approche hybride

Principes de fonctionnement La figure 5 décrit schématiquement le principe de fonctionnement d'une approche de type *hybride*. Les différents processus MPI (des processus UNIX traditionnels) s'exécutent sur les différents processeurs de la machine. On considérera

que l'application y est seule et que l'on ne lance pas plus de processus que de processeurs physiquement disponibles. Un processus MPI est donc attaché à un processeur particulier. Lorsque deux processus MPI veulent communiquer, deux choix leur sont offerts :

- les deux processeurs utilisés par les deux processus communiquant (émetteur et récepteur) sont localisés dans deux machines différentes, auquel cas la bibliothèque MPI est utilisée et les capacités de communication du réseau d'interconnexion sont exploitées ;
- les deux processeurs sont localisés à l'intérieur de la même machine. Dans ce cas, un autre système de communication, plus performant que le réseau est exploité (mémoire partagée généralement). Cette exploitation est réalisée avec un second outil de programmation utilisant un paradigme différent (i.e variables partagées) de celui de MPI (i.e le passage de message).

Le fait d'utiliser un second outil de programmation pose un problème du point de vue de la portabilité des programmes MPI déjà existants. En effet, il devient nécessaire de les réécrire en introduisant l'emploi du second paradigme avec des appels à l'outil adéquat. Cette situation n'étant pas viable – MPI devant demeurer un standard – le recours à un préprocesseur est monnaie courante : le programme MPI original est modifié par une insertion automatique des appels au second outil dans le code originel.

Deux outils sont principalement employés pour les communications intra-nœuds : les processus légers et la bibliothèque OpenMP ([DM98]). Tout deux sont conçus pour exploiter la mémoire partagée et constituent *a priori* de bons candidats pour une telle utilisation.

Dans le cas des processus légers, le fonctionnement diffère selon que l'on cherche à travailler sur les communications intra-nœuds ou l'organisation des tâches de calcul :

- dans le premier cas de figure, les processus légers se doivent d'être des entités reconnues par la bibliothèque MPI. Il est alors possible de les utiliser comme support d'un processus MPI en lieu et place d'un processus lourd UNIX traditionnel. On fait s'exécuter un *unique* processus UNIX contenant un *ensemble* de processus MPI sur chaque nœud de la grappe (cf. figure 6) ;
- dans le second cas de figure, un unique processus lourd UNIX sert de support à un unique processus MPI. Ce dernier s'exécute sur l'ensemble des processeurs du nœud et contient des processus légers, non reconnus par la bibliothèque MPI et utilisés pour paralléliser les phases de calcul dans le processus (cf. la figure 7).

Dans les deux cas, la mémoire partagée est utilisée, les processus légers exploitant de façon transparente pour le programmeur l'unique espace d'adressage commun à l'ensemble des processeurs du nœud. Un tel dispositif suppose néanmoins que la bibliothèque de processus légers soit capable de fonctionner avec des machines multi-processeurs. Elle devra être soit de niveau noyau, soit de niveau utilisateur mais disposer alors d'un ordonnanceur hybride.

Discussion Cette approche est intéressante du point de vue des performances, car la mémoire partagée est mieux exploitée par les processus légers. Le recours à de la mémoire partagée ou à des sockets entre processus lourds n'évite pas de devoir effectuer une recopie intermédiaire (dans le segment de mémoire, justement), tandis qu'avec le partage de l'espace d'adressage, cette recopie est remplacée par de simples opérations de synchronisation entre

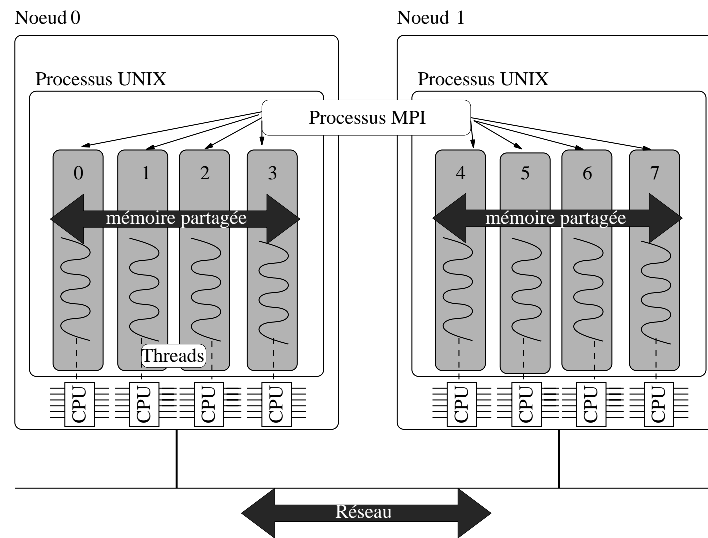


FIG. 6 – Approche hybride – Optimisation des communications intra-nœuds

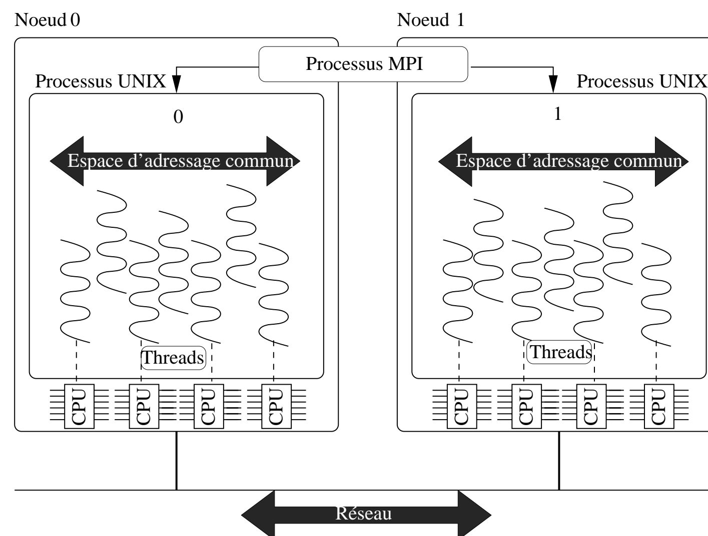


FIG. 7 – Approche hybride – Exploitation du parallélisme intra-processus

processus légers, moins coûteuses que dans le cas des processus lourds. De plus, la quantité de mémoire partagée utilisable entre des processus est limitée, mais cette limitation est moindre dans le cas d'un espace d'adressage commun. Les performances accessibles avec de telles solutions devraient être optimales pour le niveau intra-nœud.

Mais cette approche présente des limitations. Tout d'abord, elle se concentre exclusivement sur l'optimisation d'un niveau bien particulier de la hiérarchie des communications et délaisse les autres. Une solution fondée sur l'approche hybride ne peut offrir suffisamment de garanties concernant l'extensibilité (rajout de niveaux dans la hiérarchie). Comme nous nous situons dans le cadre de grappes homogènes de machines multi-processeurs, il est évident que les communications inter-nœuds doivent être à leur tour être gérées par un

élément logiciel dédié : le problème du support multi-protocole reste alors entier (comment gérer efficacement la scrutation, comment éviter une approche simple de type tourniquet, etc).

Cette approche, fondée sur des éléments logiciels divers et spécialement conçus pour un usage particulier suppose que cette collaboration se déroulera sans heurts. Or, et ceci est particulièrement vrai dans le cas de MPI et des processus légers, cela implique un important travail d'intégration des deux outils, sous peine de voir les performances n'atteindre que des niveaux décevants. L'utilisation conjointe de deux outils n'offre aucune garantie de performances correctes.

Enfin, l'utilisation des processus légers pose également le problème de la gestion des appels non-réentrants : le code MPI applicatif est souvent manipulé par un préprocesseur et il n'est pas avéré que ce mécanisme puisse détecter correctement de tels appels. De plus, le fait d'utiliser des processus légers en tant que support pour les processus MPI leur en interdit l'utilisation, ce qui pourtant devient de plus en plus courant.

Instances Les solutions implémentant une démarche de ce type sont nombreuses :

– **thread MPI** ([TY01]) est une implémentation de MPI utilisant et offrant un support pour les processus légers. Le niveau de support actuel est `MPI_Thread_serialized` (c'est-à-dire que tous les appels aux fonctions de la bibliothèque MPI doivent être sérialisés). Cette solution constitue un parfait archétype de l'approche avec une correspondance Processus MPI – Processus léger telle que décrite par la figure 6.

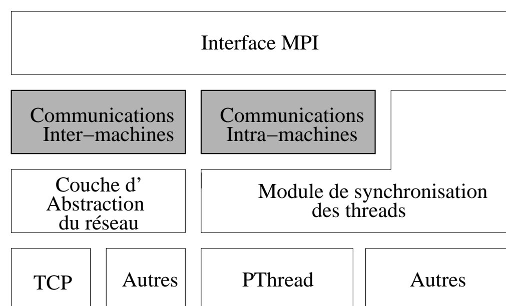


FIG. 8 – Architecture de *thread MPI*

L'architecture est schématisée par la figure 8. On remarquera d'emblée que cette solution, bien qu'elle se focalise sur l'optimisation des communications intra-nœuds, ne néglige pas le reste de la hiérarchie. L'architecture l'atteste, avec ses modules dédiés aux communications intra-nœuds mais aussi inter-nœuds. La présence d'une couche d'abstraction du réseau d'interconnexion sous-jacent (une légère anticipation : c'est aussi l'approche suivie par la bibliothèque de communication Madeleine) est aussi un signe tangible que les concepteurs n'ont pas mis de côté les aspects multi-réseaux, même si aucun support ne semble actuellement disponible.

Cette solution étant destinée à des grappes homogènes de machines multi-processeurs elle ne propose pas non plus de mécanismes pour gérer les configurations de type

grappes de grappes. La bibliothèque de processus légers utilisée est les *Pthreads* classiques, mais une autre bibliothèque pourrait la remplacer (à condition de pouvoir exploiter les machines multi-processeurs, bien entendu). Le défaut majeur réside dans l'absence de mélange entre communications et processus légers et l'ensemble du standard MPI 1 n'est pas totalement disponible ;

- **MultiThreaded Device** ([Rad97]) n'est pas une implémentation complète de MPI. Il s'agit en fait d'un module logiciel conçu pour fonctionner au sein d'une implémentation déjà existante : MPI *Chameleon* ([GLDA96], MPICH dans la suite du document). Un tel module logiciel est dénommé *device* dans la terminologie courante des développeurs MPICH. Dans le cas précis de *MultiThreaded Device*, le module est uniquement destiné à l'exploitation des communications intra-nœuds. Des modules annexes, dédiés aux communications inter-nœuds doivent lui être adjoints afin de disposer d'une version de MPI capable de gérer des configurations de type grappes. De fait, ce module n'aborde pas le problème de la gestion du multi-protocole ;
- **Thread-only MPI** ([Dem97]) est une implémentation partielle de MPI utilisant des processus légers qui servent de support aux processus MPI (cf. la figure 6). Cette implémentation vise une catégorie bien particulière de matériels : les machines multi-processeurs de taille importante, mais qui ne sont pas des grappes ;
- **Multithread Implementation of MPI** ([GCC99]) est une implémentation de MPI qui utilise les processus légers en tant qu'outil de programmation, mais aussi le paradigme associé (espace d'adressage commun). Dans cette optique, les processus légers servent de support aux processus MPI. Cette implémentation offre de plus des primitives de communications collectives optimisées pour tenir compte de la hiérarchie. Cependant, cette version de MPI n'est disponible que pour des plate-formes utilisant Windows comme système d'exploitation, ce qui réduit singulièrement son utilisation dans un contexte de calcul haute-performance ;
- **Adaptative MPI** ([LBK02]) est une implémentation de MPI qui utilise les processus légers comme des processeurs virtuels. Un processus MPI est déployé sur un processus virtuel (cf. la figure 6), qui peut être migré d'un nœud vers un autre pour réaliser un équilibrage dynamique de la charge ou redéployer des processus sur des machines nouvellement disponibles (AMPI ne met pas en œuvre une gestion dynamique ; on se contente de lancer à l'initialisation un nombre de processus virtuels plus important que le nombre de processeurs physiques disponibles à ce moment). La bibliothèque de processus légers utilisée est de niveau utilisateur et s'appelle CHARM++ ([CHAb]). L'interface du standard MPI a donc été étendue pour y intégrer ces services de migration, dont le mécanisme repose sur un principe d'allocation de piles iso-adresses, repris de l'environnement de programmation PM2 [NM95] ;
- **MPI-Lite** ([PB98]) est une implémentation de MPI utilisant les processus légers comme support pour les processus MPI. Similairement à *Thread-Only MPI*, elle n'est en principe destinée qu'aux machines multi-processeurs, pas aux grappes ;
- **MPI + OpenMP** est une solution utilisant MPI pour les communications inter-nœuds et OpenMP ([DM98]) (directives ou processus légers) pour procéder à une parallélisation du code MPI s'exécutant sur un nœud multi-processeur. Cette approche est celle décrite par la figure 7 et s'adresse prioritairement aux grappes homogènes de machines multi-processeurs : elle n'aborde pas la gestion des grappes de grappes ou du support

multi-réseau. Le problème de la collaboration d'outils différents se pose ici et le gain en performance n'est pas avéré, seules certaines classes d'applications pouvant en tirer un avantage ([CE00]). La portabilité des performances semble donc ne pas être assurée par une telle approche.

2.1.2.2 Les solutions multi-protocoles

La seconde approche retenue pour la getion de la hiérarchie repose sur l'emploi exclusif de la bibliothèque de communication MPI. Le principe de fonctionnement est explicité puis discuté dans le reste de cette partie. Des exemples de réalisations logicielles sont également fournis pour illustrer le propos.

Principes de fonctionnement La figure 9 schématise une architecture de MPI qui a été fréquemment adoptée et mise en œuvre pour exploiter les grappes homogènes de machines multi-processeurs. Elle repose sur la présence de deux modules de communication :

- le premier module est responsable des communications intra-nœud et utilise de la mémoire partagée pour les réaliser ;
- le second module est responsable quant à lui des communications inter-nœuds et s'appuie sur le réseau d'interconnexion présent dans la grappe.

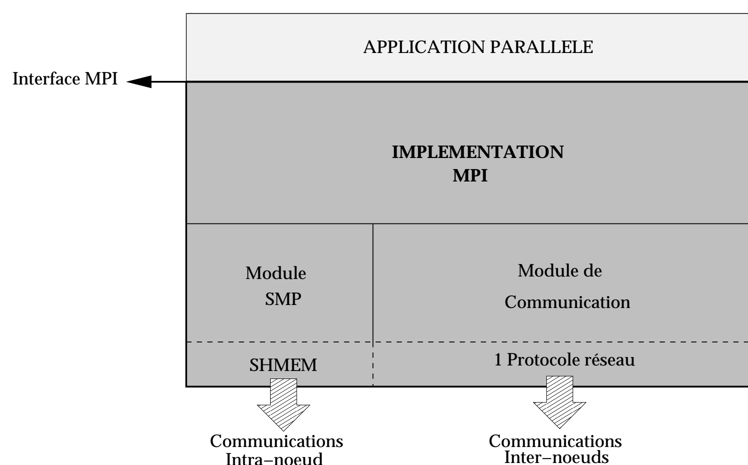


FIG. 9 – Architecture bimodulaire

Cette architecture multi-protocole a été déjà évoquée dans la section 2.1.2.1, car nous y avions indiqué que plusieurs modules de communication pouvaient coexister afin d'assurer un meilleur support des communications au sein d'une grappe (cas de *thread MPI* notamment). Cette architecture bi-modulaire est surtout une extension d'une solution ne possédant à la base qu'un unique module dédié à un support très optimisé d'un protocole réseau particulier (contexte des grappes homogènes de machines uniprocesseurs). L'arrivée des grappes de machines multi-processeurs a changé cette donne et des adaptations sont devenues indispensables. La tendance naturelle a été d'introduire un module supplémentaire pour les communications intra-nœuds. Cependant, ce support n'est pas la cible privilégiée de ce type de solutions, à la différence des approches hybrides vues précédemment.

Discussion Cette architecture ne prend pas en compte plus deux niveaux de la hiérarchie des communications, mais son principe est aisément extensible. Elle peut donc légitimement être considérée comme un substrat possible pour les configurations de type grappes de grappes (cf. 2.1.4.2). L'efficacité d'une mise en œuvre de MPI fondée sur cette architecture repose essentiellement sur la manière dont est organisée la progression des communications. Or la technique communément employée est une scrutation séquentielle de chacun des modules (algorithme de type tourniquet). Lors de cette scrutation, on s'efforce de traiter le maximum de communications en cours pour le module. L'effet induit est que le temps moyen de transfert pour l'ensemble des protocoles supportés augmente.

Instances Cette architecture est très répandue pour les implémentations de MPI destinées à exploiter des grappes homogènes de machines multi-processeurs interconnectées par un réseau haut-débit spécifique, tel que *Myrinet* ([MYR]) ou encore *Scalable Coherent Interface* ([HR99]). Voici des exemples récents :

- **MPICH-P4 w/ SHMEM** ([GLDA96]) est une des implémentations les plus répandues du standard MPI. Il s'agit de MPICH utilisant une bibliothèque de communication appelée P4 destinée aux réseaux de type Ethernet avec le protocole TCP. MPICH-P4 se destine aux grappes homogènes de machines uniprocesseurs, mais l'ajout d'un module gérant les communications en mémoire partagée permet d'étendre son champ d'action aux grappes homogènes de machines multi-processeurs. La gestion simultanée des deux modules suit le schéma classique décrit précédemment, c'est-à-dire un algorithme de type tourniquet pour la scrutation ;
- **MPICH-GM** ([MPIb]) est une implémentation de MPI fondée également sur MPICH mais le module gérant les communications inter-nœuds a pour cible le protocole GM ([GM]) destiné au réseau haut-débit Myrinet ([MYR]). Ici encore, deux modules gèrent les deux types de communications, avec les mêmes inconvénients que dans la solution précédente. Les communications par mémoire partagée, bien qu'utilisant un protocole simple avec une recopie intermédiaire, affichent d'excellentes performances (cf. 5.3.2.1, Figures 58 et 59) ;
- **MPI-BIP/SMP** ([PTW99]) est une seconde implémentation de MPI pour le réseau Myrinet, mais fondée sur le protocole BIP (*Basic Interface for Parallelism*, [PT98]). Le support des communications en mémoire partagée est assuré par un module spécial, appelé smp_plug ([GPT99]), et qui s'insère dans le noyau de GNU/Linux pour être utilisé ;
- **MPICH-PM** ([TSH^{+}00]) est une troisième implémentation de MPI pour le réseau Myrinet. Elle est basée sur la bibliothèque de communication PM développée au RWCP (Japon). Cette implémentation est discutée en 2.1.4.3 (paragraphe consacré à MPICH-SCore) ;
- **SCI-MPICH** ([SCI]) est une implémentation de MPI, fondée sur MPICH et destinée à un réseau haut-débit de type SCI (*Scalable Coherent Interface*, [HR99]). Une bibliothèque particulière, appelée SMI, fait l'interface entre MPICH et le protocole SISCI exploitant le réseau SCI. Le module de communication (*device*) réalisant cette intégration s'appelle `ch_smi`. Le support des communications intra-nœuds est lui aussi assuré. Une particularité de cette version est qu'il est possible de la configurer à l'installation pour lui permettre d'utiliser un processus léger effectuant la réception des messages. L'utilisation d'un tel processus léger à pour effet d'augmenter le temps de transfert pour les

messages de faible taille (cf. 3.1.2.2) ;

- **MPI-StarT** ([HH98]) est une autre implémentation de MPI pour les grappes de machines multi-processeurs utilisant le réseau StarT-X ([STA]). Cette version dérive également de MPICH. La topologie sous-jacente est prise en considération car les opérations collectives sont optimisées selon la répartition des processus et leur appartenance ou non à un même nœud.

2.1.3 Les architectures inter-opérables

2.1.3.1 Idées générales

Nous venons de voir que de nombreuses solutions logicielles existent pour les grappes homogènes. Une grappe de grappes étant une interconnexion de telles machines, une exploitation satisfaisante devrait s'appuyer sur l'utilisation des outils pré-existants, très optimisés pour les composantes de la grappe de grappes. C'est l'idée-force sur laquelle s'appuient les solutions inter-opérables, où sont réutilisés autant que possible les logiciels précédemment développés.

Dans ce type d'architectures, plusieurs implémentations de MPI indépendantes (souvent même différentes) se côtoient. Chacune de ces implémentations exploite une grappe locale et des éléments logiciels annexes sont rajoutés pour permettre les communications inter-grappes. En pratique, la nature de ces éléments logiciels est assez restreinte puisqu'il pourra s'agir :

- soit d'une version de MPI dont la cible sera le réseau interconnectant les nœuds d'une des grappes locales ;
- soit d'un module logiciel spécialement développé à cette fin.

Conceptuellement, les deux solutions sont très proches.

Dans le reste de cette partie, nous allons donc détailler la série suivante d'architectures répondant à notre cahier des charges :

- l'architecture fédérative multi-MPI ;
- l'architecture fédérative avec module extérieur de communication ;
- l'architecture récursive.

Nous ne citerons plus le cas des communications intra-nœuds pour plus de lisibilité (le cas des communications intra-nœuds ayant été traité en 2.1.2), mais il est bien évident qu'elles sont concernées et traitées aussi par ces solutions. Nous recentrons notre discours sur les communications intra- et inter-grappes.

2.1.3.2 L'architecture fédérative multi-MPI

Principe de fonctionnement L'architecture fédérative multi-MPI, décrite par la figure 10, n'utilise que des bibliothèques de communication MPI : une première version de MPI sert pour les communications intra-grappes et une seconde gère les communications inter-grappes. Cette architecture est donc intrinsèquement multi-grappes et hérite les propriétés des versions de MPI sur lesquelles elle s'appuye :

– si le MPI employé pour les communications inter-grappes est multi-réseaux, alors cette solution le devient également ;
– si le MPI employé pour les communications intra-grappes supporte les grappes de machines multi-processeurs, alors ce sera également le cas pour cette solution ;
– si le MPI employé pour les communications intra-grappes est multi-réseau alors ce sera également le cas pour cette solution ;

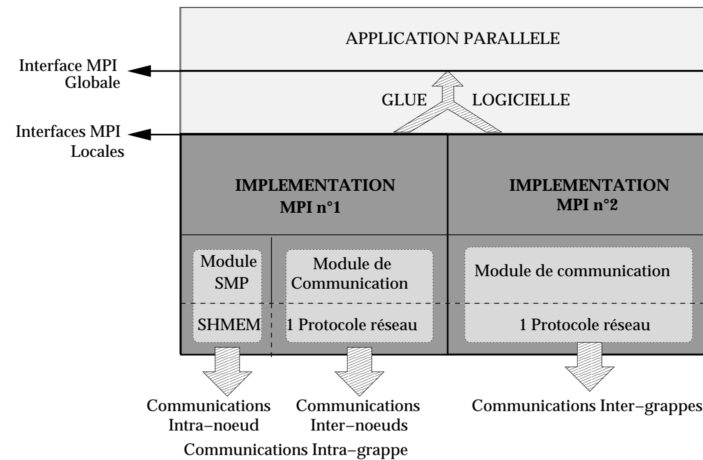


FIG. 10 – L'architecture fédérative multi-MPI

La présence de deux versions de MPI nécessite l'introduction d'une glue logicielle permettant de mettre en commun les deux interfaces et d'offrir ainsi une vision unifiée de l'ensemble à l'application.

Discussion Cette architecture semble à première vue séduisante : la réutilisation d'éléments logiciels pré-existants permet une simplification du travail de développement. Ces mêmes élements étant généralement très optimisés pour les réseaux sous-jacents, les performances devraient être au rendez-vous. Enfin, comme les communications inter-grappes sont basées sur MPI, l'ensemble du standard est disponible pour le programmeur d'applications.

Toutefois, les inconvénients sont multiples : comme dans le cas de la solution hybride (cf. 2.1.2.1), le fait d'utiliser des composants logiciels spécialisés devrait conduire *a priori* à des bonnes performances. Cependant, il est possible que les implémentations de MPI mettent en œuvre des techniques qui soient incompatibles, conduisant à une dégradation des performances. Par exemple, une première implémentation de MPI pourrait utiliser des mécanismes de scrutation gênants pour un fonctionnement efficace de la seconde. Un autre problème réside au niveau des configurations supportées. S'il est possible d'exploiter des configurations de type grappes de grappes (avec des grappes locales possédant des réseaux différents, plus un troisième réseau les interconnectant), la nature des réseaux reliant ces multiples éléments se doit d'être homogène. La seule possibilité de supporter des réseaux hétérogènes interconnectant ces grappes réside dans le support multi-réseaux de la version de MPI utilisée pour ce type de communications. Le problème demeure donc entier, puisqu'on se retrouve avec la question du support des réseaux multiples dans MPI.

Instances Il n'existe en pratique qu'une unique instance de cette architecture : **MPI-GLUE** ([Rab98]). Il s'agit d'une des toutes premières solutions pour exploiter les architectures de type grappes de grappes. MPI-GLUE résoud en partie le problème de l'intégration car la scrutation des différents réseaux peut être adaptative, selon leur caractéristiques (vitesses de transmission, débit).

Du côté des communications inter-grappes, seul le protocole TCP est supporté, ce qui interdit l'exploitation de liens hauts-débits entre les différentes grappes lorsqu'ils existent. MPI-GLUE n'utilise pas de processus-démons et tous les processus s'exécutant sur des grappes distinctes sont connectés les uns aux autres par des sockets TCP.

2.1.3.3 L'architecture fédérative avec module extérieur de communication

Principe de fonctionnement Ce type d'architecture, schématisé par la figure 11 est une variation de l'architecture vue précédemment (cf. 2.1.3.2). La différence réside dans la gestion des communications inter-grappes : plutôt que d'utiliser une autre implémentation de MPI, un module de communication dédié est employé. La gestion des communications intra-grappes ne change pas, une version de MPI optimisée pour le réseau d'interconnexion sous-jacent étant de mise. C'est l'interface de cette version de MPI qui est exportée vers l'application dont les appels aux routines MPI sont répartis par une glue logicielle :

- dans le cas d'une communication intra-grappe il s'agit d'un vrai appel MPI. Dans ce cas, c'est la version installée sur la grappe locale qui est directement utilisée ;
- dans le cas d'une communication inter-grappe, l'appel MPI est intercepté et redirigé vers le module extérieur de communication qui retransmet cet appel vers le module de communication sur la grappe distante. Ce dernier se charge à son tour d'acheminer le message vers le processus destinataire.

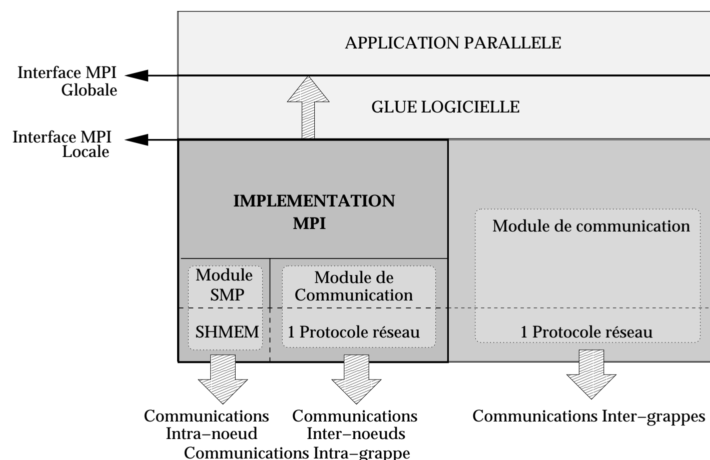


FIG. 11 – L'architecture fédérative avec module exterieur de communication

Le second point constitue une deuxième différence avec l'architecture précédente qui utilise un schéma de communication que l'on peut qualifier de *direct* où chaque processus présent dans la grappe de grappes communique directement avec les autres. Dans le cas de

l'architecture avec module extérieur, les communications sont retransmises par le module. En pratique, des processus-démons sont utilisés pour en exécuter le code.

Enfin, cette solution qui est intrinsèquement multi-grappes, peut être multi-réseaux si le MPI utilisé pour les communications intra-grappes l'est aussi et si le module externe le permet.

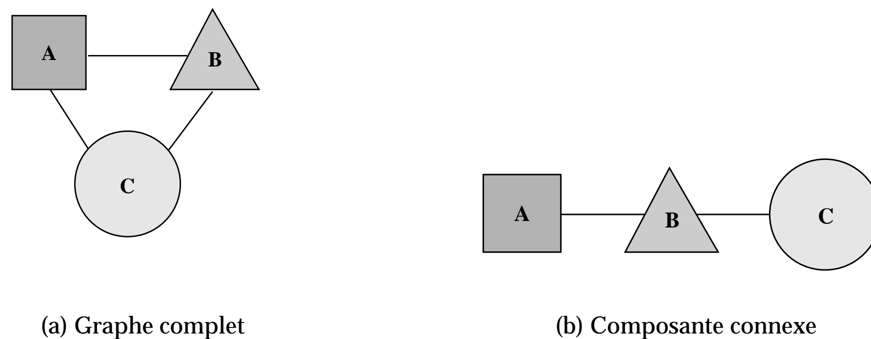
Discussion Cette architecture présente l'avantage – tout comme la précédente – de réutiliser des versions de MPI pré-existantes. La différence par rapport à l'architecture fédérative multi-MPI réside dans la meilleure intégration des divers éléments logiciels : en particulier, le module extérieur de communication est spécialement conçu pour fonctionner de concert avec une bibliothèque de communication MPI, ce qui permet de procéder à des optimisations, ou du moins de prendre en compte le mode de fonctionnement de cette dernière. Mais il est évident qu'il n'est pas possible d'écrire un module *entièrement* générique : les implémentations de MPI présentent des caractéristiques de fonctionnement différentes qui ne sont pas forcément compatibles avec celles de ce module. Un autre avantage de l'utilisation d'un tel élément logiciel est la possibilité de mettre en place des services pour améliorer les performances des communications inter-grappes, comme de la compression, du routage, etc. Enfin, cette architecture possède les propriétés des éléments logiciels sous-jacents : elle sera multi-réseaux du moment que la version de MPI ainsi que le module de communication le sont.

Cependant, si la gestion de la scrutation pour les différents types de communications n'est plus effectuée dans MPI, il faut tout de même l'implémenter dans la glue logicielle. De plus, l'ensemble du standard MPI n'est pas forcément entièrement disponible pour développer des applications : le programmeur sera limité par le sous-ensemble de fonctionnalités supporté par le module extérieur de communication. Il existe une limitation similaire à celle de l'architecture multi-MPI, à savoir que dans le cas d'interconnexions de plus de deux grappes, l'ensemble des réseaux d'interconnexion doit être homogène. Sauf à développer des capacités de multi-réseaux dans le module extérieur de communication, ce qui revient à retomber sur le problème de départ (car si l'on cherche à introduire ce genre de fonctionnalité dans un tel module, alors autant le faire directement dans une implémentation de MPI !).

Instances Les implémentations de MPI utilisant une telle architecture sont assez nombreuses :

- **PArallel Computer eXtension-MPI (PACX-MPI)** ([GRBK98]) est sans doute la version la plus connue de MPI illustrant ce principe. PACX utilise deux processus-démons par grappe, pour la retransmission des messages vers les autres grappes. Ces démons peuvent être deployés sur le même nœud, ou sur deux nœuds distincts et ne participent pas au calcul. Cela induit une diminution des ressources de calcul, qui peut être plus ou moins sensible selon la taille des grappes. Les processus-démons fonctionnent toujours par paire :
  - le premier processus-démon gérant toutes les communications sortant de la grappes ;
  - le second processus-démon gérant toutes les communications entrantes.

Les communications inter-grappes utilisent actuellement le protocole TCP (ATM est également supporté) et il est possible d'étendre le nombre de protocoles utilisés pour ces communications. Au niveau des topologies supportées, la limitation suivante mérite d'être soulignée : l'ensemble des grappes doit former un graphe complet (cf. la Figure 12(a)) et non simplement une composante connexe, si bien qu'en dépit de la présence d'un mécanisme de retransmission, PACX est dans l'impossibilité d'exploiter une interconnexion de grappes simplement "alignées" (cf. la figure 12(b)). Une étude ([Tra02]) dresse un bilan plus complet des capacités de PACX pour la gestion des configurations de type grappes de grappes ;



(a) Graphe complet (b) Composante connexe

FIG. 12 – Topologies d'interconnexion

– **toute implémentation de IMPI** (cf. 1.1.2.3) met en place une architecture de ce type. Il convient de remarquer que les implémentations de IMPI sont peu nombreuses, et que cet autre standard n'adresse que le problème de l'inter-opérabilité des implémentations de MPI-1, pas de MPI-2. Le problème de IMPI réside au niveau des performances, puisque pour connecter des processus dépendants d'implémentations différentes, IMPI utilise des processus de retransmission (au moins un par implémentation de MPI dans la configuration concernée, les démons *impid*), qui sont reliés les uns aux autres par des sockets TCP/IP. Au final, le design intrinsèque de IMPI limite les communications inter-grappes au seul protocole TCP et impose des retransmissions (cf. Figure 13), même dans le cas où l'on chercherait à faire communiquer deux versions identiques de MPI exploitant des architectures similaires ;

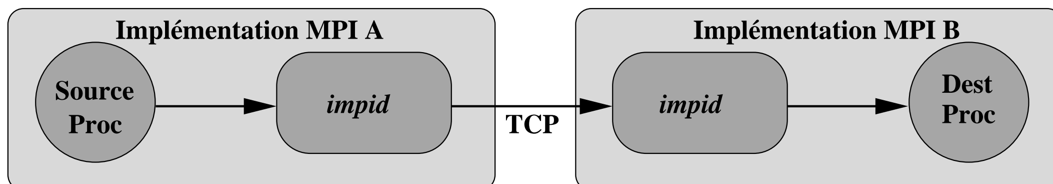


FIG. 13 – Retransmissions dans IMPI lors d'un appel à `MPI_Send`

– **Unify** ([VSRC95]) est une tentative de fusion des outils MPI et PVM, l'objectif étant de pouvoir faire cohabiter des appels à chacun de ces outils au sein d'un même code

applicatif. cette approche a été plébiscitée avant la publication de MPI-2, car l'utilisation de PVM, en particulier de ses capacités dynamiques, permettait de contourner les limitations du modèle statique de MPI-1. Dans le cas de Unify, ces fonctionnalités n'étaient toutefois pas utilisables, PVM servant uniquement de module de communication extérieur entre les différentes implémentations de MPI. Unify peut être considéré comme un précurseur de PVMPI (cf. le paragraphe ci-dessous), mais l'ensemble des fonctionnalités de MPI n'a jamais été complètement supporté et ce projet n'a pas réussi à atteindre une maturité suffisante pour que son usage se généralise ;

- **PVMPI** ([ED96], [ED97]) procède d'une façon analogue, en utilisant PVM comme module extérieur de communication. Cet emploi entraîne une sévère limitation puisque seul le protocole TCP est utilisable pour les communications inter-grappes. Or, l'un des problèmes de PVM étant les performances, le cumul TCP + PVM pour ces communications laisse dubitatif quant au niveau que celles-ci pourraient atteindre. Le multi-réseau n'est pas supporté mais PVMPI possède en revanche une gestion dynamique des processus, cette fonctionnalité étant héritée du substrat PVM. PVMPI utilise deux schémas pour les communications inter-grappes : soit un schéma direct, soit un schéma avec retransmission si le logiciel est configuré pour utiliser des processus-démons ;
- **MPI-Connect** ([MPIa]) est un projet actuellement en sommeil. Son principe est basé sur une interception des appels à la bibliothèque MPI et dans le cas de messages concernant des grappes distinctes, des routines PVM sont utilisées. Dans le cas contraire, un appel MPI régulier est réalisé. Les limitations sont celles habituelles dans le cas de l'emploi de PVM : performances non satisfaisantes et utilisation exclusive de TCP pour les communications inter-grappes. Ce projet propose également un support pour MPI-2 (et donc pour la gestion dynamique des processus).

2.1.3.4 L'architecture récursive

L'architecture récursive constitue la dernière solution inter-opérable. Il s'agit d'un mélange des deux architectures précédentes, avec un module extérieur de communication et une glue logicielle, cette dernière étant réalisée dans MPI au lieu de lui être extérieure.

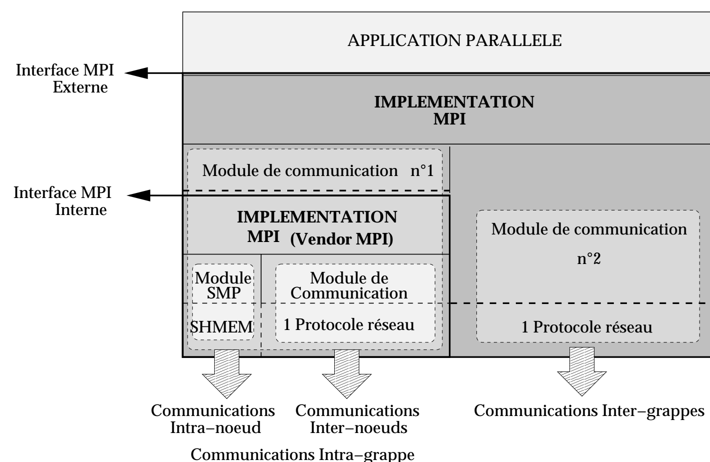


FIG. 14 – L'architecture récursive

Principe de fonctionnement Le nom "récursif" se justifie car cette architecture possède une particularité : l'implémentation de MPI est bâtie autour de deux modules de communication : le premier est un module extérieur et les communications intra-grappes s'appuient également sur une seconde version de MPI, locale à la grappe (habituellement dénommée *Vendor MPI*, cf. la figure 14). Cette architecture est bien entendu multi-grappe (car interopérable) et peut être multi-réseaux si le *Vendor MPI* et le module extérieur de communications le sont également.

Discussion Comme dans les deux cas précédents, la réutilisation d'éléments logiciels préexistants permet de tirer parti d'implémentations de MPI très optimisées pour un réseau donné. En particulier, les performances des communications intra-grappes devraient être du même niveau que celles du *Vendor MPI* sous-jacent. L'emploi d'un module extérieur de communication permet un accès à des services variés (si tant est qu'ils existent) et l'ensemble du standard est disponible pour le programmeur d'applications parallèles.

Mais une fois de plus – et c'est bien là le travers fondamental des approches interopérables – la propriété de support des réseaux multiples s'obtient uniquement si les logiciels sous-jacents la possèdent déjà. Le problème de départ reste donc entier.

Instances Les instances de cette architecture ne sont pas nombreuses, mais l'une des solutions les plus répandue à l'heure actuelle pour exploiter les grilles de calcul en fait partie :

- **MPICH-G2** ([FK98], [KTF03]) est une implémentation de MPI fondée sur MPICH. Plus précisément, cette version de MPI est développée avec le concours des créateurs de MPICH. Elle utilise un module de communication appelé `ch_g2`, qui repose sur une version de MPI (*Vendor MPI*) exploitant une grappe locale (communications intra-grappes). Les communications inter-grappes sont réalisées par un module extérieur de communication utilisant TCP/IP exclusivement. MPICH-G2 n'emploie pas de mécanismes de retransmission (comme PACX, cf. ci-dessus), et les processus s'exécutant sur la grappe de grappes sont tous interconnectés par des sockets. Un service de compression est assuré pour améliorer les performances des communications inter-grappes.

  MPICH-G2 possède une hiérarchie à quatre niveaux :

  - le premier niveau est celui du *Vendor MPI* (censé exploiter un réseau haut-débit) : c'est le niveau *vendor-supplied MPI* ;
  - le deuxième est celui du réseau TCP/Ethernet interconnectant les nœuds de la grappe : c'est le niveau *intra-machine TCP* ;
  - le troisième est celui du réseau local TCP : c'est le niveau *LAN-TCP* ;
  - le dernier est celui du réseau global TCP : c'est le niveau *WAN-TCP*.

  Chaque niveau est supposé moins performant que celui qui le précède (i.e WAN-TCP < LAN-TCP < intra-machine TCP < vendor-supplied MPI). Les opérations collectives dans G2 sont optimisées pour tenir compte de ces niveaux ([Lac01]).

  Soulignons que le démarrage des applications repose sur la bibliothèque Globus ([FGG+98], [ADF+01])) plutôt destinée au *metacomputing* et aux grilles de calcul. MPICH-G2 est donc conçu pour gérer de telles configurations à large échelle, mais peut être employé pour l'exploitation des grappes de grappes. MPICH-G2 implémente

également quelques fonctionnalités de MPI-2 au niveau de la gestion dynamique des processus. Ces fonctionnalités sont orientées client-serveur :

- côté serveur : `MPI_Open_port`, `MPI_Close_port` et `MPI_Comm_accept` ;
- côté client : `MPI_Comm_connect` ;

Outre le fait que ce système vise d'abord les grilles et non les grappes de grappes, l'emploi exclusif de TCP pour les communications inter-grappes élimine l'utilisation des liens à haut débit existants. De plus, la hiérarchie est arbitrairement fixée à quatre niveaux et n'est pas extensible. Enfin, l'intégration dans MPICH-G2 d'une version de MPI en tant que *Vendor MPI* suppose que cette dernière soit relativement récente, ce qui élimine d'emblée un certain nombre de systèmes ;

- **GridMPI** ([IMK+03]) est le pendant nippon de MPICH-G2. Il s'agit d'une implémentation de MPI spécialement destinée aux grappes de grande taille et aux grilles de calcul et prenant en compte les longues latences inhérentes à ce type de configurations. GridMPI dérive de YAMPII (*Yet Another MPI Implementation*, [YAM]), implémentation créée *ex-nihilo* par des équipes de recherche japonaises (ce qui est plutôt rare, les développements de nouvelles implémentations étant longs et ardus, le recours à MPICH ou LAM/MPI étant plutôt la règle). GridMPI possède néanmoins une architecture similaire à celle de MPICH-G2 (cf. Figure 15) avec toutefois deux différences notables :
  - d'une part la présence d'une couche logicielle supplémentaire, située entre la couche responsable des communications point-à-point et l'*Abstract Device Interface* (cf. 4.1.2). Cette couche est destinée à prendre en compte les latences des différents liens de la topologie sous-jacente pour améliorer les performances des communications ;
  - d'autre part GridMPI peut être configuré pour utiliser un *VendorMPI* ou bien une bibliothèque de communication autre que MPI.

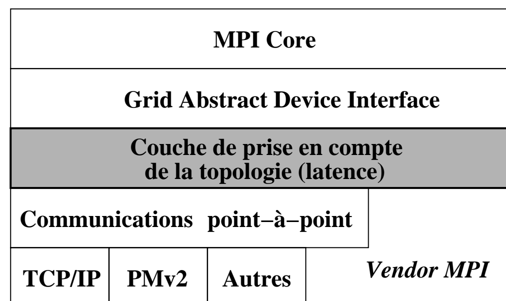


FIG. 15 – Architecture de GridMPI

GridMPI ne possède donc pas une hiérarchie à quatre niveaux, comme dans le cas de MPICH-G2, mais se base sur les latences. Il est intéressant de noter que les concepteurs de GridMPI pointent le problème des passerelles : s'il est peu probable que tous les nœuds des grappes de grande taille possèdent une adresse IP globale, en revanche la grappe peut posséder un espace d'adressage privé. Il faut donc que certains nœuds exécutent un service de traduction (*Network Address Translation*) et c'est là un risque d'engorgement potentiel ([IMK+03]).

GridMPI est un projet récent, la documentation disponible est parcimonieuse et des évaluations ne sont pas encore disponibles, mais les problèmes de MPICH-G2 risquent de se retrouver dans GridMPI, en particulier l'utilisation exclusive de TCP pour les communications inter-grappes. De plus, en ce qui concerne les communications intra-grappes, GridMPI n'autorise pas de support multi-réseau (cf. 2.1.4.3) ;

- **Seamless Thinking Aid MPI** ([ITKT00]) est une autre solution japonaise, plus ancienne que la précédente et destinée prioritairement à l'interconnexion de supercalcuteurs hétérogènes. Cette solution opère une distinction entre les communications intra- et inter-machines, permettant l'utilisation du *Vendor MPI* le cas echéant. Dans le cas de communications inter-machines, le protocole utilisé est TCP/IP comme dans le cas de MPICH-G2. Une différence notable est que StaMPI propose un système relativement souple pour les communications inter-machines car les schémas directs ou avec retransmissions sont possibles. La différence avec PACX réside dans le nombre de processus-routeurs qui n'est pas arbitrairement fixé, mais varie selon la configuration, ce qui permet d'éviter des retransmissions coûteuses. StaMPI est également implémenté au-dessus de MPICH-SCore (cf. 2.1.4.3), ce qui lui permet de supporter les grappes homogènes de machines multi-processeurs.

  Un autre point très intéressant est que cette solution autorise la gestion dynamique des processus, mais avec une orientation plutôt «session» car la fonction `MPI_Comm_spawn` est implémentée (en plus des opérations `MPI_Open_port`, `MPI_Close_port`, `MPI_Comm_accept` et `MPI_Comm_connect`). Ceci est valable plus particulièrement pour les supercalculateurs (cible de base de StaMPI), mais des fonctionnalités de MPI-2 ont été introduites dans MPICH-SCore, ce qui fait que cette solution permet également une gestion dynamique des processus dans le cas de grappes de grappes ([TIYT03]). Au final cette solution est insuffisante car non multi-réseaux (pour les communications inter-machines) en se contentant de TCP.

2.1.4 Les architectures intrinsèquement hétérogènes

2.1.4.1 Idées générales

Cette section aborde maintenant la dernière famille d'architectures : il s'agit des solutions intrinsèquement hétérogènes et qui intègrent directement dans l'implémentation de MPI les mécanismes permettant l'exploitation des configurations hétérogènes (et hiérarchiques). C'est là une différence fondamentale avec la famille des architectures inter-opérables où les propriétés sont héritées des fondations logicielles utilisées pour leur implémentation.

Nous présentons deux architectures :

- l'architecture multi-modulaire et

- l'architecture unimodulaire.

Comme dans le cas des architectures inter-opérables, il s'agit plus de variations que de changements radicaux.

2.1.4.2 L'architecture multi-modulaire

Principe de fonctionnement La figure 16 schématise cette architecture. Une unique implémentation de MPI s'appuie sur un ensemble de modules de communication où chacun est dédié au support d'une technologie réseau particulière. Cette architecture dérive de celle employée pour l'implémentation de versions de MPI destinées aux grappes homogènes de machines multi-processeurs (cf. 2.1.2.2). En fait, il s'agit d'une extension de cette architecture où la gestion de la scrutation des différents protocoles est effectuée dans les couches hautes de MPI. Cette gestion est relativement basique puisqu'il s'agit d'un parcours de la liste des modules. La fonction de progression des communications associée à chacun d'entre eux est exécutée séquentiellement (c.f 4.1.2.2).

Une telle architecture permet une prise en compte simple de la hiérarchie : si l'on associe un protocole à un niveau donné, la gestion des multiples protocoles devient synonyme de gestion de la hiérarchie. Cependant, cette dernière a tendance à s'estomper car les modules sont strictement symétriques du point de vue des couches supérieures de l'implémentation. Le support multi-réseau est également possible avec une solution de ce genre.

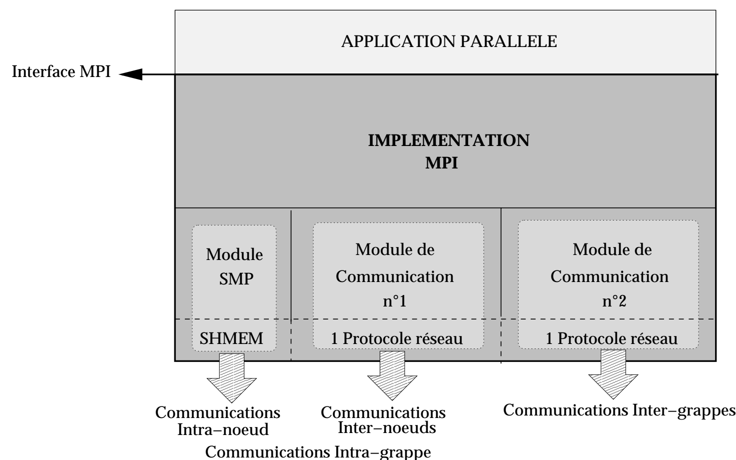


FIG. 16 – L'architecture multi-modulaire

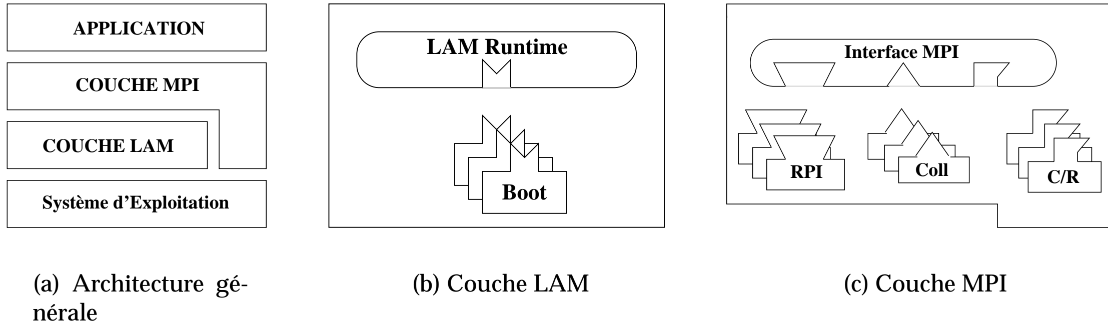
Discussion Cette architecture présente l'avantage de solutionner le problème du support des multiples protocoles de communication. De plus, la gestion des communications intergrappes est plus intéressante que dans le cas des architectures inter-opérables : l'implémentation de MPI résultante a la possibilité d'utiliser n'importe lequel des protocoles de communication disponibles, et pas uniquement le protocole TCP. Cette solution est donc plus flexible.

Cependant le problème majeur concerne la gestion de la scrutation. Cette dernière étant effectuée au niveau de MPI, la progression des communications n'intervient que lorsque l'utilisateur fait un appel à la bibliothèque. Dans le cas d'un entrelacement de longues phases de calcul et de phases de communications (ce qui correspond peu ou prou à la structure d'une application écrite avec MPI), ces dernières ne progresseront pas durant les phases de calcul. De plus, le fait d'utiliser une scrutation séquentielle pour les différents protocoles

n'est pas un procédé très extensible ; plus l'on rajoute de modules et plus le temps de transfert moyen pour l'ensemble des modules augmente.

Instances Des instances de cette architecture sont des implémentations de MPI très intéressantes :

- **toutes les solutions non-hybrides pour grappes homogènes de machines multiprocesseurs** (cf. 2.1.2.2) sont des versions simplifiées de cette architecture (ou plutôt cette architecture est une extension de ces solutions) ;
- **LAM/MPI** ([Squ03]) est l'autre implémentation libre de MPI la plus répandue à l'heure actuelle. Historiquement, il s'agit même d'une des toutes première implémentation libre du standard, car LAM (*Local Area Multicomputer*) était un environnement de programmation qui existait avant MPI. Son architecture est résolument modulaire (dans sa dernière version, la 7, cf. la figure 17(a)), que ce soit pour les mécanismes de démarrage des applications (cf. la figure 17(b)), pour les protocoles réseaux ou pour les opérations collectives (cf. la figure 17(c)). En particulier, la progression des communications est assurée par les modules appelés des RPI (*Request Progression Interface*). Plusieurs RPI peuvent cohabiter au sein de la même installation de LAM/MPI qui posséde un support pour Globus. Une implémentation de IMPI est également disponible. Les technologies réseaux supportées dans LAM/MPI sont GigaBitEthernet, Myrinet (avec GM) et Infiniband.



(a) Architecture générale (b) Couche LAM (c) Couche MPI

FIG. 17 – Structure modulaire de LAM/MPI

Des processus-démons sont utilisés sur chacun des nœuds de la grappe et servent au démarrage des processus MPI. La présence de processus-démons permet de faire interagir des applications différentes utilisant LAM/MPI ainsi qu'un support (partiel) des fonctionnalités de MPI-2 en ce qui concerne la gestion dynamique des processus. LAM/MPI autorise de plus deux niveaux de multithreading : `MPI_THREAD_SINGLE` et `MPI_THREAD_SERIALIZED` (les deux plus faibles). Bien que des processus-démons existent, LAM/MPI utilise un schéma de communication direct pour les communications inter-grappes (pas de retransmissions). C'est un point faible car sans un tel mécanisme toutes les communications inter-grappes se font sur un réseau homogène (i.e TCP même quand des liens haut-débits existent). LAM/MPI propose également un support pour la tolérance aux pannes utilisant des mécanismes de prise de points de contrôle (*checkpoints*) et de redémarrage des applications. Les points de contrôle sont

coordonnés et leur prise repose sur un RPI spécialement réécrit (actuellement, seulement au-dessus de TCP). Plusieurs politiques de prises de points de contrôle et de redémarrage (C/R) sont possibles (cf. la figure 17(c)).

Enfin, LAM/MPI est en passe de fusionner avec de deux autres projets : FT-MPI (cf. 2.2.3.2) et LA-MPI ([LAM]) pour donner naissance à une nouvelle implémentation libre de MPI : Open MPI ([OPE]). Participent à ce projet les équipes de PACX-MPI (cf. 2.1.3.3) et de MVAPICH, implémentation de MPI pour grappes ayant Infiniband comme réseau d'interconnexion ([MVA]) ;

- **chaMPIon/Pro** ([CHAa]) est une implémentation commerciale du standard MPI dans sa version 2 (2.1), multithreadée et réentrante (au moins en ce qui concerne MPI 1). Le niveau offert est `MPI_THREAD_MULTIPLE` (i.e le plus haut). Cette implémentation supporte un panel varié de technologies réseaux : Myrinet (avec le protocole GM), VIA, Infiniband, TCP/Ethernet, Quadrics ainsi que les communications par mémoire partagée. Il semble que la progression des communications soit assurée par deux modules logiciels : un module *multi-device* et un module dénommé *progress*. Nous n'avons pas réussi à déterminer la manière dont ces composants interagissent ni s'il faut y voir le signe d'une présence d'un moteur de progression des communications (cf. 3.2.1.3), l'information à ce sujet nous faisant défaut. De plus, chaMPIon/Pro serait capable de prendre en compte la topologie sous-jacente, en offrant notamment des opérations collectives optimisées. Enfin, ChaMPI/Pro possède également une gestion dynamique des processus : les tâches supplémentaires sont lancées avec la commande *mpirun* uniquement, il n'y a pas de processus-démons présents sur les nœuds de calcul pour procéder à la mise en route ou établir les connexions entre les différents processus.

2.1.4.3 L'architecture unimodulaire

Principe de fonctionnement Cette solution reprend le principe de l'architecture précédente mais pousse sa logique d'intégration encore un cran au-dessus : l'implémentation de MPI est basée sur un unique module de communication (cf. la figure 18). C'est ce module qui gère l'ensemble des procotoles de communication, en particulier la scrutation. Là encore, la hiérarchie s'efface car l'utilisation d'un module unique gomme tous ces aspects. Cette architecture est potentiellement multi-grappes.

Discussion Cette solution est la plus extensible car le support pour de nouvelles technologies réseaux s'effectue sans avoir à modifier l'implémentation de MPI. Tout le travail se concentre dans le module de communication. MPI gagne de plus un accès transparent et direct à tous les services réalisés par ce module : routage et retransmission, détection et tolérance aux pannes, etc. L'ensemble du standard est également disponible. Cette solution résoud aussi le problème de la scrutation rencontré dans le cas de l'architecture multimodulaire, car étant effectuée directement au sein du module, il est possible de tenir compte de la présence des différents protocoles.

Cependant, le revers de la médaille réside justement dans l'utilisation d'une couche de communication opaque, ce qui conduit à un manque de transparence pour les applications qui ne sont plus capables de connaître la nature de la topologie sous-jacente afin d'en tirer parti. De plus, si les performances de ce module de communication sont faibles, alors aussi

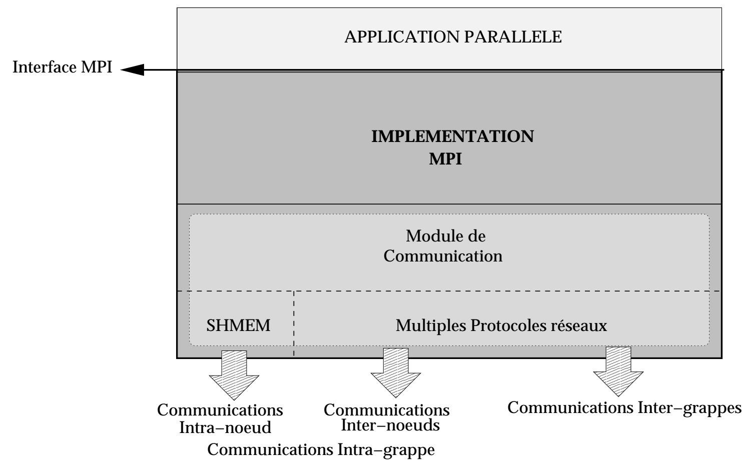


FIG. 18 – L'architecture unimodulaire

bien les communications intra-grappes qu'inter-grappes en pâtiront.

Instances On recense quelques instances de cette architecture :

- **MPI/I-WAY** [FGT] est historiquement une des plus anciennes solutions pour les grilles de calcul et les grappes de grappes. Il s'agit (s'agissait) d'une implémentation de MPI fondée sur MPICH et destinée à fédérer des centres de ressources de calcul. D'un point de vue de l'implémentation, c'est la bibliothèque de communication Nexus qui était employée. Cette bibliothèque a été utilisée dans Globus, mais son usage à par la suite été abandonné car les performances n'étaient pas satisfaisantes. Elle bénéficiait d'une sélection automatique des protocoles les plus performants selon les communications (cf. la figure 19). En pratique, les protocoles supportés étaient MPL et TCP est c'est ce dernier qui était dédié aux communications inter-grappes. Cette implémentation utilisait également des processus légers (les *Pthreads*). MPI/I-WAY peut donc aisément être qualifié d'ancêtre de MPICH-G. Cette solution n'affichait pas de performances satisfaisantes et l'introduction des processus légers de niveau noyau (cas de l'implémentation des *Pthreads* sous UNIX) s'est soldée par un certain discrédit de cette approche ;
- **MPICH-SCore** est une autre implémentation de MPI fondée sur MPICH, et originellement destinée à exploiter des grappes homogènes de machines multi-processeurs interconnectées par un réseau de type Myrinet. SCore est en fait un système d'exploitation pour grappes, avec des fonctionnalités de d'ordonnancement de groupe (*gang scheduling*) pour un partage d'une grappe aussi bien spatial que temporel. Cette version de MPI utilise une bibliothèque de communication de bas-niveau, appelée PM, actuellement dans sa deuxième version (PMv2, [TSH+00]). Les technologies supportées à ce jour sont Myrinet et GigaBitEthernet. Cette bibliothèque a vu ses mécanismes étendus dans le but de passer à une gestion de multiples protocoles réseaux.

  Ainsi, MPICH-SCore devrait être multi-réseaux, mais la lecture de [TSH+00] nous a laissé dubitatifs quant à cette assertion. En effet, le problème de la gestion de protocoles multiples est un sujet non trivial et l'article passe allègrement dessus. Quant

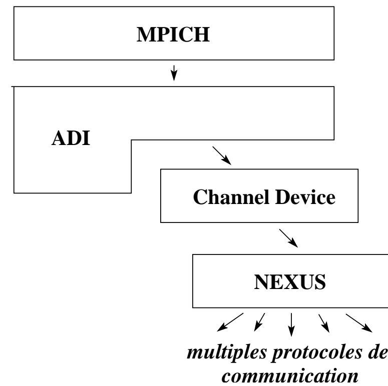


FIG. 19 – Architecture de MPI/I-WAY

aux mesures de performances, elles se concentent sur des évaluations de grappes homogènes de machines multi-processeurs (communications intra-grappes, donc) et font l'impasse sur les communications inter-grappes. [TIYT03] vient confirmer cette analyse car StaMPI a été porté au-dessus de MPICH-SCore afin de réaliser de telles communications. Cette opération eût été inutile si MPICH-SCore fût capable de le faire nativement ;

– **GridMPI** pourrait également suivre un fonctionnement de ce type, quand est utilisée la bibliothèque de communication PMv2 ([TSH+00]) plutôt qu'un *Vendor MPI*. Cependant, le paragraphe précédent nous permet d'affirmer que cela ne peut être le cas en l'état actuel. Signalons que le principal concepteur de YAMPII, utilisé comme fondation par GridMPI, a dirigé l'équipe qui a mis au point MPICH-SCore (cf. 2.1.2.2) ;

– **MPICH-VMI** ([PP]) est l' archétype de l'approche de type «boîte noire». Il s'agit d'une implémentation de MPI qui utilise la bibliothèque VMI (*Virtual Machine Interface*), dont le but premier est de permettre une portabilité des codes binaires d'applications MPI sur un ensemble de grappes. MPI profite des services offerts par cette bibliothèque, et ce de façon transparente. [PP] cite les services qui devraient, à terme, être disponibles dans VMI. On trouve entre autres :

  – de la fragmentation de messages sur plusieurs liens potentiellement hétérogènes ;
  – un changement dynamique de réseau quand une panne est détectée.

  Les réseaux actuellement supportés dans VMI sont Myrinet/GM, Infiniband/VIA et GigaBitEthernet/TCP. En particulier, on remarquera que l'utilisateur ne sait pas quel réseau il utilise. Ce choix est laissé à l'appréciation de VMI ce qui est dommageable pour le contrôle de la topologie par l'application ;

– **MPIConnect** [SCA] est une implémentation commerciale de MPI réalisée par la société Scali. Peu d'information est disponible à son sujet, mais l'architecture est celle décrite par la figure 20. Une couche de portabilité, appelée *Direct Access Transport* ([DAT]) s'intercale entre les couches hautes de MPI et l'ensemble des protocoles de communication. MPIConnect peut fonctionner au-dessus de réseaux de type Ethernet, GigaBitEthernet, Myrinet, SCI et Infiniband. Les communications par mémoire partagée sont aussi supportées. Des processus légers sont utilisés pour l'implémentation, ce qui rend cette dernière réentrante et permet d'exporter cet emploi au niveau applicatif (niveau

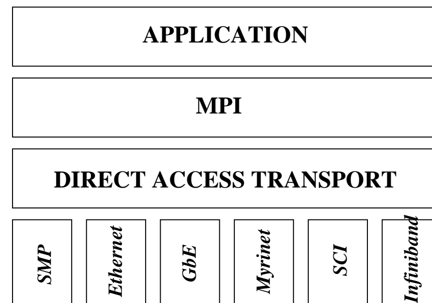


FIG. 20 – Architecture de MPIConnect

supporté : `MPI_THREAD_MULTIPLE`). MPIConnect est capable, tout comme MPICH-VMI de changer dynamiquement de réseau d'interconnection en cas de panne. Cette solution semble donc – comme la précédente – souffrir d'un manque de transparence pour l'utilisateur qui peut donner des indications sur les priorités des réseaux à utiliser, mais ne peut diriger finement ceci depuis l'application. Enfin, les communications inter-grappes n'utilisent pas de retransmissions, si bien que dans le cas des grappes de grappes, les communications point-à-point se font sur des réseaux homogènes. En pratique, cela veut dire utiliser TCP même en présence de liens haut-débits interconnectant partiellement les grappes.

2.1.5 Autres approches

Nous abordons dans cette section un ensemble de solutions qui échappent à la classification que nous avons établie.

2.1.5.1 L'architecture homogène unificatrice

Cette solution est sans conteste la plus simple pour apporter une solution au problème de la gestion des configurations hiérarchiques et hétérogènes dans MPI.

Principe de fonctionnement Il s'agit d'une solution par défaut (de repli serait plus juste peut-être) dont le principe est extrêmement simple, tout comme la réalisation : si l'on considère que la technologie Ethernet (ainsi que ses déclinaisons : Fast, GigaBit) de paire avec le protocole TCP/IP sont disponibles universellement, alors il suffit de les utiliser. Les grappes de grappes deviennent en quelque sorte une sorte de grosse grappe homogène avec Ethernet en tant que réseau d'interconnexion.

Discussion Cette approche, du «plus petit dénominateur commun», n'est pas vraiment hiérarchique car les communications intra- et inter-grappes se confondent. De plus, l'utilisation exclusive d'Ethernet/TCP n'est pas satisfaisante car non seulement les liens à haut-débit reliant les grappes ne sont pas employés, mais en plus les réseaux rapides dans les grappes locales sont également inutilisés. D'un point de vue des performances, il n'y a rien a attendre

d'extraordinaire d'une telle solution, même en ayant recours à un certain attentisme (Ethernet a été remplacé par FastEthernet, qui sera bien un jour supplanté par GigaBitEthernet ...). L'attente d'une évolution technologique comme seul facteur d'amélioration n'est définitivement pas satisfaisant, ni intellectuellement, ni opérationnellement, mais cette approche fonctionne et offre une solution minimale aux problèmes posés. Enfin aussi criticable qu'elle soit, cette approche n'est pas tellement éloignée des solutions inter-opérables (cf. 2.1.3), qui ont définitivement fait leur deuil de l'utilisation des liens haut-débit inter-grappes, pour ne recourir en pratique qu'à TCP.

2.1.5.2 MetaMPICH

Principe de fonctionnement MetaMPICH ([PSB03]) est une implémentation de MPI fondée sur MPICH et qui vise à l'exploitation de grappes de grappes (appelées dans la terminologie MetaMPICH des *metacomputer*). Les concepteurs de MetaMPICH ont bien réalisé que l'emploi de TCP pour les communications inter-grappes pouvait constituer un goulot d'étranglement pour les performances et que l'utilisation des liens à haut-débit entre les grappes – quand ils existent – devait être privilégiée.

Ce constat justifie la présence de processus-routeurs, chargés de retransmettre les messages entre deux processus n'appartenant pas à la même grappe. Ces routeurs logiciels sont implémentés à l'aide de processus légers : un couple de processus légers se charge de l'émission et de la réception des messages depuis/vers une grappe locale, tandis qu'un ensemble de processus légers est chargé du transfert de ces messages vers la grappe distante (cf. la figure 21).

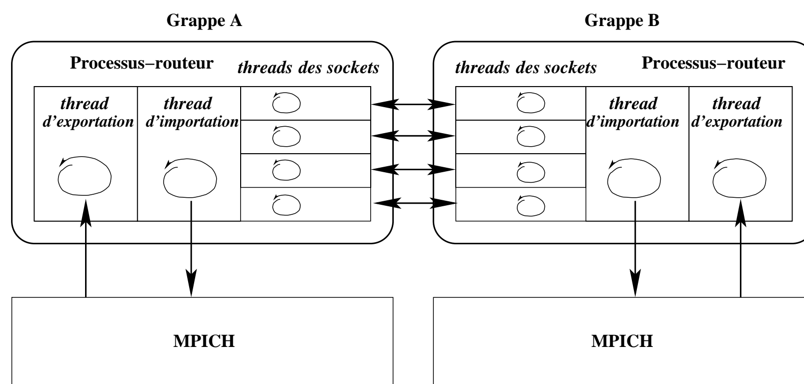


FIG. 21 – Structure d'un processus-routeur

Les routeurs peuvent procéder à une fragmentation des messages s'il y a plusieurs cartes réseaux disponibles afin d'augmenter le débit et d'optimiser l'utilisation des ressources. La particularité concerne la nature de ces processus-routeurs, qui sont des processus MPI réguliers, contrairement à l'approche avec des démons (comme PACX, cf. 2.1.3.3) où ces derniers ne sont pas des membres de la session MPI. MetaMPICH utilise donc un langage de description permettant de définir le rôle des divers processus dans la session MPI : processus applicatifs réguliers ou bien processus-routeurs.

MetaMPICH utilise trois modules logiciels (*ADI devices*) pour la gestion des communications (cf. la figure 22) :

- un module régulier exploite le réseau d'interconnexion de la grappe locale ;
- le module *Passerelle* (`ch_gateway`) est un *pseudo-module* dont la tâche est de retransmettre les messages *vers* le processus-routeur ;
- le module *Tunnel* (`ch_tunnel`) est un *pseudo-module* dont la tâche est de distribuer les messages dans la grappe locale *depuis* le processus-routeur.

Il convient de noter que les deux derniers modules ont besoin d'un module régulier pour fonctionner et qu'il «usurpent» en fait l'identité du processus réel (émetteur du message) appartenant à la grappe distante. MetaMPICH est une solution tout particulièrement destinée aux grappes de grappes.

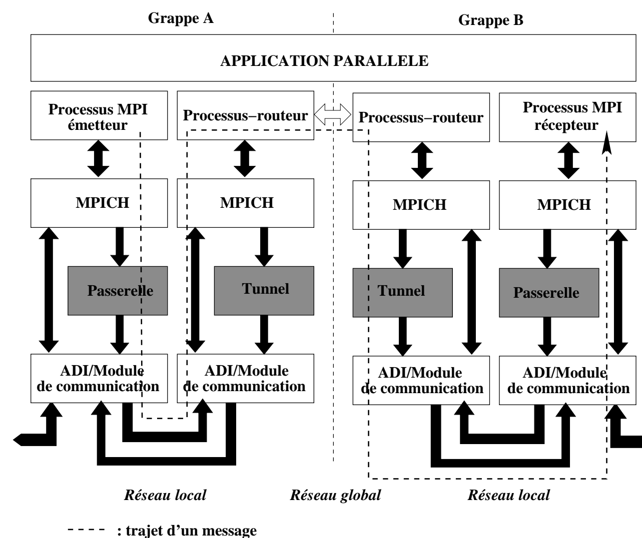


FIG. 22 – Principe du routage/retransmission dans MetaMPICH

Discussion Il s'agit de la seule solution refusant une utilisation exclusive de TCP pour les communications inter-grappes. La possibilité de pouvoir utiliser des liens hauts-débits est très intéressante, même si cela peut devenir à son tour un goulot d'étranglement potentiel (dans le cas où le *ratio* quantité de communications/nombre de liens serait faible). Cependant, si cette solution est orientée «grappes de grappes», la gestion des grappes multiplement câblées ne semble pas optimale car à ce moment, l'utilisateur sera dans l'obligation de créer des passerelles entre les sous-grappes afin de les faire communiquer. Un schéma simple où le réseau haut-débit est utilisé à l'intérieur de chaque partition et où le réseau global d'interconnexion est employé entre les sous-grappes (pas d'intervention de routeurs) n'est pas prévu dans le fonctionnement de MetaMPICH. Ce manque de souplesse d'utilisation est regrettable mais il ne faut pas perdre de vue que MetaMPICH se concentre uniquement sur les communications inter-grappes et ne s'intéresse pas à la gestion multi-protocole dans MPI. Par conséquent, le support multi-réseaux, aussi bien pour les communications intra- qu'inter-grappes n'est pas mis en place. Au final, MetaMPICH pâtit d'un manque de généricité.

2.1.5.3 HMPI

Principe de fonctionnement *Heterogeneous MPI* ([AL03]) n'est pas une implémentation de MPI. Il s'agit en fait d'une extension permettant de gérer l'hétérogénéité *avec* MPI (plutôt que *dans*). HMPI travaille essentiellement sur les communicateurs et opère une correspondance entre des derniers et un découpage des processus applicatifs selon des critères d'appartenance à une partition dont le réseau possède certaines propriétés (latence, débit). L'hétérogénéité dont il est ici question n'est donc pas celle traitée par les solutions vues jusqu'à présent, mais concerne les propriétés du matériel du point de vue des performances (cf. 1.1.3.3). Cet outil est un préprocesseur et s'emploie pour écrire des algorithmes. Il vise donc le niveau applicatif, pas l'implémentation de MPI en soi.

Discussion HMPI présente un intérêt car s'il ne règle pas les problèmes du support de la hiérarchie et de l'hétérogénéité dans une implémentation de MPI, il donne des pistes sur l'utilisation de MPI dans un tel contexte. Soulignons que [AL03] pointe trois manques dans les implémentations actuelles de MPI :

- le support pour l'hétérogénéité des réseaux (hétérogénéité au sens "niveau des performances") ;
- le support pour la tolérance aux pannes ;
- la support pour l'écriture de programmes tenant compte de ces aspects.

Concrètement, seul ce dernier point est traité, et curieusement, [AL03] affirme que pour les deux autres points, presque rien n'a été fait dans ces domaines ! Étonnant pour un article datant de 2003.

2.2 Systèmes supportant des applications dynamiques

Après avoir abordé le problème de la gestion de la hiérarchie et de l'hétérogénéité, nous examinons celui du support des applications dynamiques.

2.2.1 Introduction

Ainsi que nous l'avons évoqué en 1.1.3.4, la notion de dynamicité est assez souple. Si l'on prend comme définition le fait que l'implémentation de MPI est capable de gérer une modification du nombre de processus au cours de l'exécution, alors on se retrouve avec deux familles de solutions correspondant à ces critères :

- d'une part les solutions implémentant une gestion dynamique des processus basique (sur le principe) : des opérations comme le rajout ou la suppression d'un (ou d'un ensemble de) processus sont disponibles ;
- d'autre part les solutions mettant en œuvre des mécanismes de prise en compte du phénomène de défaillance des processus applicatifs. C'est le cas en particulier des implémentations de MPI tolérantes aux pannes qui doivent assurer un bon fonctionnement de l'application même (et surtout) en un tel cas. Les mécanismes mis en place peuvent bien entendu reposer sur ceux, plus simples, du point précédent mais ceci, ainsi que nous le verrons par la suite, n'a rien d'obligatoire.

Le reste de cette section est donc consacré à la description de solutions appartenant à ces deux familles.

2.2.2 Solutions avec simple gestion dynamique des processus

La gestion dynamique des processus dans MPI se pose depuis longtemps (cf. [LG94]), avant même la publication de la seconde mouture du standard. La comparaison avec PVM ([LG98], [GKP98]) ayant mis en exergue le manque de souplesse de MPI, ces problèmes ont été résolus avec MPI-2. Cependant, le nombre d'implémentations de MPI-2 est restreint et ces dernières sont souvent partielles, seules des portions bien délimitées du standard étant mises en œuvre, comme les opérations unilatérales (*one-sided operations*), les fonctions client-serveur, etc. Des implémentations de MPI avec gestion dynamique des processus ont déjà été examinées dans le cadre du support multi-protocole, notamment :

- **chaMPIon/Pro** (cf. 2.1.4.2) qui est la seule implémentation disponible et totalement opérationnelle (à ce jour) de MPI-2 que nous ayons recensée[1]. Elle est commerciale et implémente donc le modèle client-serveur du standard ;
- **LAM/MPI** (cf. 2.1.4.2) implémente un support partiel de MPI-2 et en particulier la gestion dynamique des processus ;
- **MPICH-G2** procède de même (cf. 2.1.3.4 pour la liste des fonctions implémentées) ;
- **StaMPI** implémente aussi une fonction permettant plutôt la fusion de session (cf. 2.1.3.4) ;
- **PVMPI** (cf. 2.1.3.3) en revanche, hérite sa gestion dynamique des processus du substrat PVM, ce support est découplé des fonctionnalités de MPI-2.

La plupart de ces solutions ont une approche orientée «client-serveur», chose attendue car dans le droit-fil du standard MPI-2, sauf pour StaMPI qui a choisi une approche session, ce qui permet plus facilement de traiter le cas d'applications voulant se connecter les unes aux autres.

Enfin, on remarquera que le nombre de solutions est relativement peu important en comparaison du nombre d'implémentations de MPI disponibles s'efforcant d'apporter des réponses au double problème de la hiérarchie et de l'hétérogénéité.

2.2.3 Solutions intégrant des mécanismes de tolérance aux pannes

Cette partie va nous permettre de dresser une liste de solutions disposant de mécanismes de détection des erreurs et de tolérance aux pannes. Cette liste n'étant pas exhaustive, le lecteur pourra se reporter sur [BBC+02] pour une classification plus aboutie des systèmes mettant en œuvre de la tolérance aux pannes. Il s'agit de la seconde famille évoquée en introduction de cette section. La tolérance aux pannes implique une variation du nombre de processus au cours du déroulement d'une application, une panne supposant que l'implémentation possède des fonctionnalités de déconnexion (ou du moins de conservation de l'intégrité de la session lors d'une déconnexion) et une éventuelle réintégration du processus défaillant suppose en revanche des fonctionnalités d'ajout.

[1]Même si nous sommes persuadés que d'autres versions existent

MPI et la tolérance aux pannes Quels sont les rapports entre le standard MPI est la propriété de tolérance aux pannes ? Une assertion courante est que MPI n'est pas tolérant au pannes. Ceci est une interprètation erronnée de ce que stipule le standard et n'est que le comportement d'une implémentation particulière dudit standard. En l'occurence ce comportement, dans la plupart des implémentations de MPI, est d'arrêter l'application en cas de défaillance d'un processus mais ceci n'est pas spécifié par MPI : il s'agit d'un *choix d'implémentation.* [LG02] est très clair à ce sujet et clarifie les rapports entre le standard et la propriété :

> *« La tolérance aux pannes n'est ni une propriété de MPI (car c'est un ensemble de spécifications), ni une propriété d'une implémentation de MPI, car aucune implémentation de MPI ne pourra jamais garantir qu'un programme MPI quelconque sera à l'abri de défaillances. La tolérance aux pannes est une propriété d'un programme MPI couplé avec une implémentation spécifique de MPI »*

Dans notre cas, nous allons considérer les implémentations dont le comportement est de permettre à l'application de poursuivre sa bonne exécution. Pratiquement, il n'y a pas unicité du comportement, car plusieurs politiques de base sont possibles :

- l'implémentation MPI se remet automatiquement de cette panne et permet la poursuite de l'exécution de l'application en cours ;
- l'application est avertie de la panne et l'implémentation lui délègue le soin d'entreprendre les actions idoines ;
- l'implémentation rend certaines opérations invalides ou inutilisables ;
- l'application termine son exécution et reprend automatiquement (*restart*) à partir d'un point de contrôle (*checkpoint*).

D'autres politiques dérivant de celles décrites ci-dessus sont également possibles, par exemple l'implémentation MPI peut redémarrer à partir d'un point de contrôle, permettant à l'application de continuer son exécution. Parce qu'il nous a paru réducteur de classifier les solutions selon la politique choisie (comme dans la section 2.1), notre examen des solutions suit l'ordre alphabétique.

2.2.3.1 CoCheck MPI

Principe de fonctionnement La première version de CoCheck MPI ([Ste96]) était construite à partir du système Condor ([Con]), utilisé pour réaliser des prises de points de contrôle dans une application MPI. La seconde version (l'actuelle) est fondée sur une implémentation de MPI appelée tuMPI et dérivant de MPICH (pour les fonctionnalités de haut-niveau). C'est la bibliothèque NXLib qui est chargée de la partie communication de l'ensemble. CoCheck est un élément logiciel *a priori* indépendant de l'implémentation de MPI, mais en pratique il a été intégré dans le code de tuMPI. Le principe de fonctionnement de CoCheck MPI est basé sur la prise de points de contrôle et le redémarrage de l'application. Cette prise de points de contrôle est coordonnée (i.e l'ensemble des processus procède simultanément à ce relevé) et est réalisée au niveau de l'implémentation de MPI, pas de l'application. De plus, tuMPI autorise la migration des processus de façon transparente pour l'application. Ces mécanismes reposent sur la présence d'un *coordinateur central,* un processus-démon composé de plusieurs modules :

- le *starter* est utilisé pour démarrer des processus ;
- le *locator* est un serveur d'adresses. Chaque processus MPI possède une adresse (réseau) virtuelle qui est utilisée pour communiquer avec lui. Cette correspondance entre l'adresse virtuelle et l'adresse réelle est indispensable car les processus sont migrables. Chaque processus MPI possède une telle table de correspondance qui doit être mise à jour à chaque migration. Ce module aide de plus les différents processus MPI à se connecter les uns aux autres ;
- le *grouper* est utilisé pour faire la gestion des groupes et contextes MPI.

La figure 23 schématise une configuration avec deux processus tuMPI et le coordinateur central. C'est ce dernier qui s'occupe de relancer des processus applicatifs en cas de défaillance.

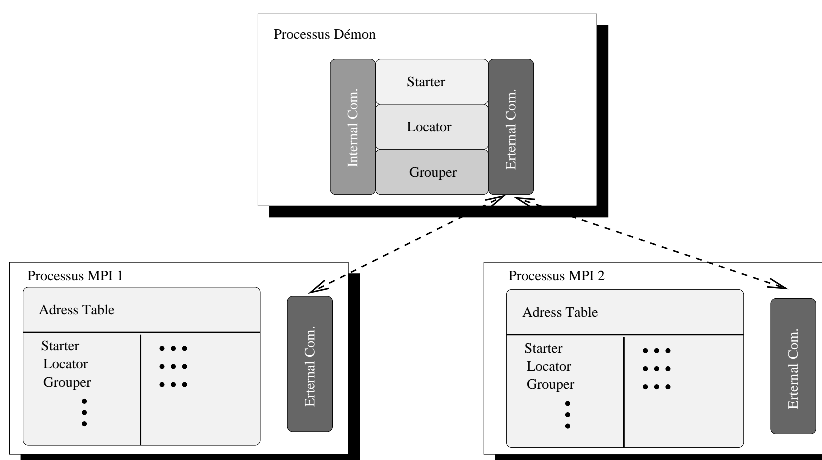


FIG. 23 – Structure des processus tuMPI et du coordinateur central

Discussion Le problème de cette approche, dans une optique de tolérance aux pannes se situe au niveau de l'extensibilité (à cause du coordinateur central unique). Notre souci étant la gestion dynamique des processus (support multi-sessions), cette solution n'est pas satisfaisante non plus car on ne peut ajouter des processus supplémentaires en cours d'exécution, même quand il n'y a pas de défaillances. En fait, le nombre de processus applicatifs lancés originellement constitue une limite : ce nombre peut varier, certes, mais jamais le dépasser.

2.2.3.2 FT-MPI

Principe de fonctionnement FT-MPI ([FD00],[FBD01]) est une implémentation de MPI élaborée au-dessus du système Harness, conçu pour le *metacomputing* plus que pour les grappes. Il s'agit en fait de l'interface MPI pour ce projet. Architecturalement, les communications point-à-point s'appuient sur une bibliothèque de processus légers (SNIPE_Lite) de façon à augmenter la réactivité et le recouvrement des communications par du calcul. Les protocoles supportés sont : TCP, GM (pour Myrinet), BIP (pour Myrinet), VIA et la mémoire partagée (cf Figure 24).

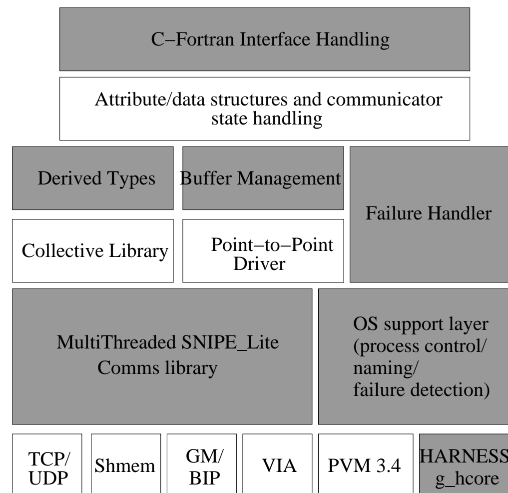


FIG. 24 – Architecture de FT-MPI

Le principe de fonctionnement de FT-MPI repose sur des modifications de la sémantique de MPI en ce qui concerne les communicateurs. En effet, le standard précise que les communicateurs connaissent deux états, le couple {*valid, invalid*}. FT-MPI rajoute des états intermédiaires, censés refléter les périodes d'instabilité où une défaillance est détectée et où le système y répond. Les communicateurs possèdent désormais des tailles variables et la présence de processus vides y est même tolérée. Plusieurs opérations ont donc été rajoutées dans ce but :

- `SHRINK` : si un processus dans un communicateur tombe en panne, alors la taille du communicateur se réduit et les processus sont renumérotés. Un appel à `MPI_Comm_size` est donc nécessaire pour prendre en compte la modification au niveau de l'application ;
- `BLANK` : comme `SHRINK`, sauf que le communicateur peut contenir des «trous» (processus invalides pour le moment) qui seront comblés plus tard. L'opération `MPI_Comm_size` retourne la taille du communicateur, pas le nombre de processus valides qu'il contient ;
- `REBUILD` : on créé de nouveaux processus pour combler les trous dans un communicateur. Soit les nouveaux processus prennent la place des trous, soit on fait une opération `SHRINK` puis les nouveaux processus sont placés «à la fin» ;
- `ABORT` : l'application termine son exécution immédiatement.

La tolérance aux pannes est gérée par l'utilisateur à deux niveaux : soit au niveau des communicateurs, soit au niveau des communications elles-mêmes. L'utilisateur a le choix lors de la détection de la défaillance : retour à un point de contrôle ou autres solutions. FT-MPI n'effectue pas lui-même le travail de prise de points de contrôles, mais permet l'utilisation d'outils le réalisant.

Discussion Cette solution a une approche intéressante en ce qui concerne la gestion des communicateurs, qui sont utilisés pour exporter au niveau de l'application les nouvelles fonctionnalités introduites. Les performances sont également satisfaisantes, notamment en

raison de l'absence de mécanismes de prise de points de contrôle ou d'enregistrement des processus MPI.

Cependant, aussi intéressante soit-elle, cette approche semble dangereuse, puisqu'elle se permet de changer la sémantique du standard. L'étendre aurait été une meilleure solution car dans ce cas, on conserve le noyau initial des fonctionnalités de MPI tel quel et on lui adjoint ce que l'on désire. Les deux n'interfèrent pas et cela garantit qu'une application MPI quelconque fonctionnera avec cette implémentation. De plus, comme dans le cas de CoCheck MPI, il n'est pas possible d'ajouter des processus supplémentaires, la variation du nombre étant uniquement dans un sens décroissant, ce qui ne correspond pas au cahier des charges fixé.

2.2.3.3 MPICH-V

Nous abordons maintenant MPICH-V (V pour *Volatile*) ou plutôt la famille de solutions car il existe plusieurs variations : MPICH-V, MPICH-V2, MPICH-V CL et MPICH-V3. Nous allons nous concentrer sur MPICH-V et MPICH-V2, car la version CL est anecdotique (c'est l'algorithme qui change : il s'agit ici d'un algorithme de prise de points de contrôle coordonnés fondé sur celui de Chandy et Lamport) et MPICH-V3 est très récent, peu d'information est disponible.

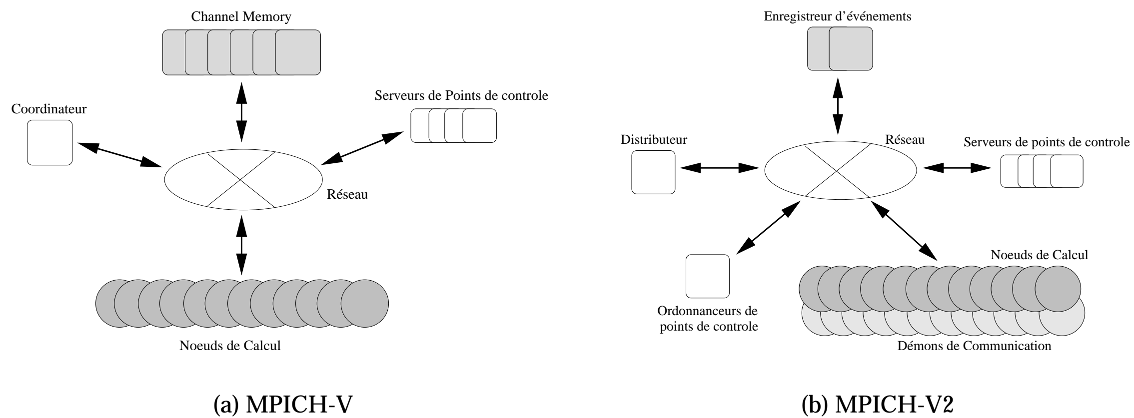
MPICH-V ([BBC+02]) est une implémentation de MPI tolérante aux pannes, fondée sur la version MPICH-CM elle-même dérivant de MPICH. Il s'agit d'un module spécifique de communication dénommé `ch_cm` et utilisant de façon sous-jacente le protocole TCP pour les communications. Le principe de fonctionnement repose sur des prises de points de contrôle et des enregistrements des messages échangés entre les processus. Une originalité de MPICH-V réside dans l'absence de contrôle central ou de «photographie» globale de la situation, qui limitent généralement l'extensibilité (cf. nombre d'approches précédentes). MPICH-V recours à divers éléments pour fonctionner (cf. Figure 25(a)) :

- un coordinateur qui gère les processus MPI. Il les lance au démarrage de l'application et coordonne également les ressources dont cette dernière aura besoin (nœuds pour exécuter des services (comme les *Channel memories* ou les serveurs de points de contrôle) ou des processus MPI réguliers). Il connecte également ces services entre eux ;
- des *Channel Memories* qui sont des entités conservant les messages en transit. Elles conservent le contexte de communication et leur implémentation utilise des processus légers. Ces entités sont en fait un reliquat de MPICH-CM ;
- des serveurs de points de contrôle *Checkpoint servers* ;
- les nœuds de communication.

MPICH-V suppose que le système de prise de points de contrôle est fiable, tout comme le coordinateur.

Une conséquence de l'utilisation des *Channel Memories* est que lors d'une communication entre deux processus, le message se voit dans l'obligation de transiter par une de ces entités pour y être stocké. Un tel mécanisme est très coûteux car le temps de communication double mécaniquement. Enfin MPICH-V ne permet pas d'ajouter de nouveaux nœuds à une application ; en cas de défaillance de l'un d'entre eux, des nœuds supplémentaires, réservés dès

le lancement de l'application sont utilisés pour les remplacer.



(a) MPICH-V (b) MPICH-V2

FIG. 25 – Architectures comparées de MPICH-V et MPICH-V2

MPICH-V2 ([BCH+03]) est le successeur de MPICH-V. Cette version a été mise au point dans le but d'éliminer le problème inhérent à l'utilisation des *Channel Memories* (i.e le doublement du temps de communication). Il s'agit toujours d'un système de prise de points de contrôle non-coordonnés (fondé sur Condor), couplé avec un enregistrement des messages. À la différence de MPICH-V, cet enregistrement est scindé en deux phases :

Phase 1 les données du messages sont stockées sur le nœud d'émission

Phase 2 l'évenement correspondant (date et indentité du récepteur) est stocké sur un serveur d'événements (supposé fiable) par un processus-démon s'exécutant au coté de chaque processus MPI

L'architecture a également subi des modifications (cf. Figure 25(b)) : le Coordinateur a été remplacé par un Distributeur (*dispatcher*) et une entité supplémentaire a été ajoutée : l'ordonnanceur de points de contrôle. Les prises des points de contrôle ne sont toujours pas coordonnées, mais sont désormais ordonnancées pour obtenir de meilleures performances. MPICH-V2 fonctionne encore au-dessus de TCP et utilise des processus-démons réalisant les connexions entre les processus MPI. Les entités supposées fiables pour avoir un fonctionnement correct sont : le Distributeur, les unités de stockage et les enregistreurs d'événements. Une dernière différence notable avec MPICH-V est que MPICH-V2 déclenche automatiquement le redémarrage de l'application en cas de défaillance d'un nœud.

Un avantage de MPICH-V2 est son indépendance vis-à-vis de l'implémentation de MPI employée : il serait donc en théorie possible d'obtenir un LAM/MPI-V2. Les performances sont nettement meilleures que celles de MPICH-V, mais cette version de MPI, qui cible des applications à large échelle de type pair-à-pair est plutôt destinée aux applications de longue durée, avec de gros messages, car le temps de transfert est sensiblement plus élevé que dans le cas d'une version classique de MPICH. Enfin, comme son prédécesseur, il n'est pas possible de rajouter des processus supplémentaires, ce qui fait que MPICH-V2 ne répond pas non plus à notre cahier des charges.

2.2.3.4 MPI-FT

Principe de fonctionnement MPI-FT ([LNLE00]) est une implémentation de MPI fondée sur LAM/MPI. Son principe de fonctionnement est similaire à celui de FT-MPI (mais sa mise en œuvre diffère), c'est-à-dire une modification de la sémantique MPI pour les communicateurs. Les pannes sont détectées à l'aide d'un script UNIX et un processus appelé l'*observer* se charge de la surveillance des processus MPI. Cet *observer* est une entité supposée fiable et relance de nouveaux processus MPI quand d'autres ont disparu. Les messages échangés entre les processus sont enregistrés (*message logging*) selon deux approches :

- avec l'approche optimiste, les processus MPI doivent enregister leurs messages de traffic seulement ;
- avec l'approche pessimiste, *tous* les messages sont enregistés dans l'*observer*.

Discussion Cette solution diffère de la précédente car les communicateurs conservent une taille fixe. Cependant, les réserves concernant la modification de la sémantique sont toujours d'actualité. Ces nouveaux processus relancés par l'*observer* reprennent leur exécution depuis le début donc le coût pour se remettre d'une défaillance est élévé. L'enregistrement des messages de traffic suppose également une grosse capacité de stockage. Enfin, MPI-FT ne permet pas de rajouter de nouveaux processus participant à l'application, les nouveaux introduits ne sont que des remplaçants de disparus.

2.2.3.5 MPI/FT(TM)

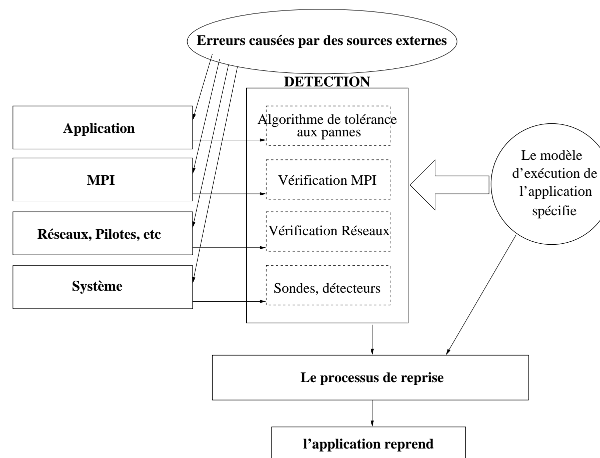


FIG. 26 – Principe de fonctionnement de MPI/FT(TM)

Principe de fonctionnement MPI/FT(TM) ([BSC⁺01]) est une extension de MPI qui a été implémentée dans MPICH. Son principe de fonctionnement est décrit par la figure 26 : la détection des fautes se fait au niveau de MPI dont l'implémentation elle-même vérifie son propre état pour garantir sa cohérence. Cette tâche est accomplie par un ensemble de processus légers, appelés les *Self Cheking Threads*. En fait, la détection des pannes peut intervenir

à quatre niveaux différents, MPI/FT(TM) travaillant à l'un de ces niveaux. D'autres mécanismes peuvent donc être utilisés complémentairement.

Discussion Cette solution est peu extensible, car elle emploie un coordinateur central qui ne peut gérer, d'après ses concepteurs, qu'une dizaine de nœuds. De plus, cette tolérance aux pannes ne permet pas une véritable gestion dynamique des processus, à savoir l'ajout de nouvelles tâches est impossible.

2.2.3.6 Starfish MPI

Principe de fonctionnement Starfish MPI ([AF99]) est une extension de MPI, dont l'implémentation est fondée sur le système Starfish qui met en œuvre la tolérance aux pannes aussi bien de façon interne que pour les applications développées au-dessus. Ainsi, Starfish intègre un système de communications collectives de groupe pour des ensembles de processus (*Ensemble*) et des mécanismes de prises de points de contrôle (coordonnés ou non). La conséquence est que MPI gagne de façon transparente ces capacités. Starfish MPI permet en effet d'exécuter des programmes MPI classiques sans aucune modification, mais pour tirer le meilleur parti de toutes les fonctionnalités disponibles (e.g prises de points de contrôle décidées par l'utilisateur, reconfiguration dynamique) des altérations deviennent nécessaires. Starfish MPI est écrit en OCaml et utilise un processus léger pour effectuer des scrutations. Les fonctionnalités de MPI-2 relatives à la gestion dynamique des processus sont disponibles et les technologies réseaux supportées sont Ethernet et Myrinet.

Quand un nœud tombe en panne, on peut alternativement en ajouter un nouveau qui reprend ce que faisait le défaillant ou adapter l'application pour qu'elle exploite cette configuration réduite. Chaque nœud possède un processus-démon et ces derniers forment un groupe appelé le *Starfish Group* qui utilise le système de communication de groupe *Ensemble*. Tous les messages de contrôle et de reconfiguration transitent via *Ensemble*, mais pas les messages de données. Ces processus-démons vont créer les nouveaux processus MPI, vérifier la présence de pannes, effectuer les opérations de reprise d'après-panne et enfin maintenir l'intégrité de la configuration du système. Les communications entre ces processus-démons et les processus MPI se font avec TCP.

La structure d'un processus MPI est modulaire, avec les entités suivantes (cf. Figure 27) :

- un *group handler*, responsable de la communication avec le processus-démon ;
- un partie application, qui contient le code de l'application MPI ;
- un module de prise de points de contrôle et de redémarrage, ce qui confère la possibilité de changer facilement de politique de tolérance aux pannes ;
- un module MPI qui correspond à l'implémentation ;
- un module *Virtual Network Interface* est qui est concrètement la seule partie à changer pour obtenir le support d'une nouvelle technologie réseau.

Ces modules communiquent les uns avec les autres par l'intermédiaire d'un bus d'objets. Pour des raisons de performances, ce dernier peut être contourné dans le cas de communications de messages de données entre les parties «application», «MPI» et «*Virtual Network Interface*» (utilisation d'un *Fast Path*).

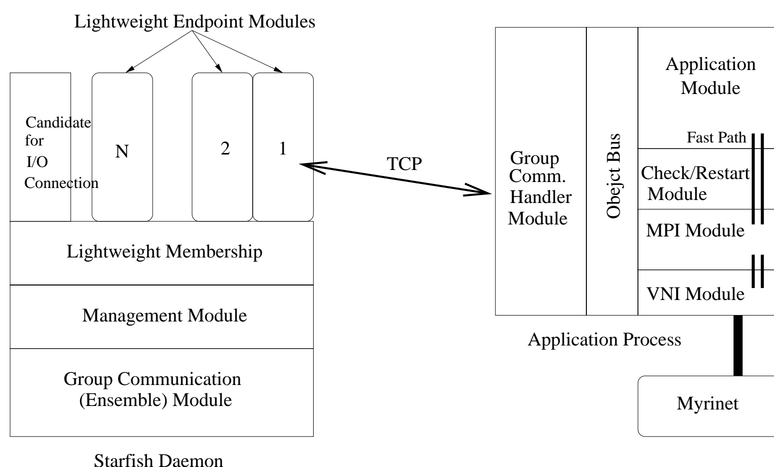


FIG. 27 – Structure des processus Starfish MPI

Discussion Cette solution lève la limitation de CoCheck MPI puisque la présence des fonctionnalités de MPI-2 permet donc d'introduire des processus MPI supplémentaires dans la configuration d'origine. Cependant, le problème majeur réside dans les performances puisque le langage choisi pour développer ce système (OCaml) entraîne l'emploi de *bytecode* ce qui grève les performances de l'ensemble. En ce sens, cette solution ne répond pas au cahier des charges de MPI car c'est la performance qui est au centre des préoccupations, plutôt que l'intégration de fonctionnalités poussées et pas obligatoirement nécessaires.

2.3 Bilan de l'existant

2.3.1 Gestion de la hiérarchie et de l'hétérogénéité

En ce qui concerne le problème de la gestion des configurations hiérarchiques et hétérogènes, le panel de solutions est large et varié. Une tendance nette se dessine, à savoir que la conception est souvent guidée par la réutilisation de logiciels pré-existants. Une refondation totale de l'architecture prenant dès l'origine la mesure du problème demeure rare. Les solutions se décomposent en deux grandes familles : les approches inter-opérables, dont le défaut majeur est la délégation de la résolution du problème aux couches plus basses et les approches intégrant directement les mécanismes recherchés. Même si une attention toute particulière est requise pour obtenir une scrutation efficace, c'est définitivement de ce côté que nous trouverons une architecture susceptible de répondre à nos besoins. Il faut cependant rester vigilant à conserver un contrôle de la topologie par l'application sans négliger les services (comme le routage et la retransmission) qui peuvent apporter d'importants bénéfices pour les performances des communications inter-grappes.

C'est d'ailleurs le point noir des approches quelles qu'elles soient : à part la notable exception de MetaMPICH, la seule solution envisagée pour les communications inter-grappes repose sur l'emploi du protocole TCP, car il est universel. Ceci n'est pas forcément une mauvaise chose, mais ne saurait être totalement satisfaisant.

Enfin, et ceci concerne au premier chef la gestion de la hiérarchie, ces deux familles de solutions, loin d'être opposées, sont complémentaires. Si les solutions inter-opérables se destinent plus à une exploitation de configurations à large échelle comme les grilles de calculs et si le problème des grappes de grappes est mieux appréhendé par les solutions intégrant directement les mécanismes, alors il serait possible de construire des systèmes destinés aux grilles et reposant sur des implémentations de MPI ciblant les grappes de grappes.

2.3.2 Gestion de la dynamicité

Du côté de la gestion de la dynamicité, le constat est plus mitigé : peu de solutions répondent à nos besoins. Les solutions implémentant des systèmes de tolérance aux pannes ne permettent pas en pratique d'ajouter des processus supplémentaires, *a fortiori* pas de fusionner deux sessions. Quant aux solutions hétérogènes, peu proposent également une gestion dynamique des processus, les deux candidats les plus sérieux étant LAM/MPI et MPIConnect. Cependant, dans les deux cas, des limitations apparaissent quant aux configurations supportées et aux protocoles utilisables pour les communications inter-grappes.

Au niveau de la réalisation, on remarquera que généralement, les systèmes proposant une gestion dynamique des processus utilisent des démons ou tout autre entité externe pour mettre en œuvre les mécanismes recherchés.

2.3.3 Conclusion

Il y a donc un manque criant de plate-forme capable de remplir entièrement le cahier des charges, c'est à dire un support efficace et peu coûteux des nouvelles fonctionnalités voulues. À nos yeux, la seule implémentation susceptible de convenir serait MPIConnect, qui est commerciale et dont la gestion des communications inter-grappes n'est pas optimale.

On peut noter qu'un nombre important d'implémentations de MPI permettent un transfert de technologie entre des systèmes de plus bas niveau et les applications. Ces implémentations possèdent des propriétés particulières en termes de fonctionnalités, qui correspondent en fait à celles réalisées dans ces systèmes (tolérance aux pannes, systèmes multirails, etc). Les développeurs essayent donc de trouver le moyen d'exploiter ces fonctionnalités et/ou de les faire remonter au niveau de MPI. En ce sens, MPI remplit bien son rôle d'interface standard, et les gens préfèrent l'adopter car il est peu probable que les développeurs d'applications changent leurs habitudes, quand bien même ils remplaceraient leur outil habituel pour un autre plus performant.

2.3.4 Tableau récapitulatif des fonctionnalités supportées

Ce tableau synthétise pour chaque système étudié dans cet état de l'art les fonctionnalités supportées quant à la hiérarchie, l'hétérogénéité et la dynamicité.

Implémentation de MPI		Gestion du Multi-échelle ?	Gestion du Multi-réseaux ?	Gestion dynamique des processus ?
TMPI	[TY01]	SMP	Non	Non
MT	[Rad97]	SMP	Non	Non
TOMPI	[Dem97]	SMP uniquement	Non	Non
MiMPI	[GCC99]	SMP	Non	Non
AMPI	[LBK02]	SMP	Non	Non
MPI-Lite	[PB98]	SMP uniquement	Non	Non
MPICH-P4 w/shmem	[GLDA96]	SMP	Non	Non
MPICH-GM w/shmem	[MPIb]	SMP	Non	Non
MPI-BIP/SMP	[PTW99]	SMP	Non	Non
SCI-MPICH w/shmem	[SCI]	SMP	Non	Non
MPI-StarT	[HH98]	SMP	Non	Non
MPI-Glue	[Rab98]	**Multi-grappes**	Dépend du MPI	Dépend du MPI
PACX-MPI	[GRBK98]	**Multi-grappes**	Dépend du MPI	Non
IMPI	[IMP00]	**Multi-grappes**	Dépend du MPI	Non
Unify	[VSRC95]	**Multi-grappes**	Dépend du MPI	Non
PVMPI	[ED96]	**Multi-grappes**	Non	**Oui**
MPI-Connect	[MPIa]	**Multi-grappes**	Non	**Oui**
MetaMPICH	[PSB03]	**Multi-grappes**	Non	Non
MPICH-G2	[KTF03]	**Multi-grappes**	Dépend du MPI	**Oui**
GridMPI	[IMK+03]	**Multi-grappes**	Non	Non
StaMPI	[ITKT00]	**Multi-grappes**	Non	**Oui**
LAM/MPI	[Squ03]	**Multi-grappes**	Non	**Oui**
chaMPIon/Pro	[CHAa]	**Multi-grappes**	Non	**Oui**
MPICH-VMI	[PP]	**Multi-grappes**	**Oui**	Non
MPICH-SCore	[TSH+00]	SMP	Non	**Oui**
MPIConnect	[SCA]	**Multi-grappes**	**Oui**	**Oui**
MPI/I-WAY	[FGT]	**Multi-grappes**	Non	Non
HMPI	[AL03]	Non	Non	Non
CoCheck MPI	[Ste96]	Non	Non	Partielle
FT-MPI	[FD00]	Non	Non	Partielle
MPICH-V	[BCH+03]	Non	Non	Partielle
MPI-FT	[LNLE00]	Non	Non	Partielle
MPI/FT(TM)	[BSC+01]	Non	Non	Partielle
StarFish MPI	[AF99]	Non	Non	**Oui**

Chapitre 3

Proposition : une architecture extensible pour un support efficace des environnements hiérarchiques, hétérogènes et dynamiques dans MPI

L'état de l'art nous a convaincu de la nécessité de proposer une architecture logicielle originale, résolument extensible (capable de s'adapter aux futures configurations matérielles), intégrant de façon native les différents mécanismes destinés à résoudre les défis du matériel et des applications. Cette architecture est l'objet de ce chapitre : nous allons tout d'abord procéder à un rappel des exigences des configurations visées ainsi que des limitations des propositions actuelles. Nous exposons ensuite les leçons tirées de l'analyse de ces limitations avant de mettre en évidence les principales caractéristiques de notre proposition. Nous détaillons ensuite comment notre architecture peut être utilisée pour l'exploitation de configurations à plus large échelle avant de dresser un bilan.

3.1 Analyse et démarche

Notre démarche est double : d'une part nous exhibons les besoins actuels de fonctionnalités et d'autre part nous pointons les manques dans les solutions proposées aussi bien d'un point de vue fonctionnel que des performances obtenues. Nous explicitons enfin la démarche adoptée.

3.1.1 Rappels des exigences

3.1.1.1 Cas des architectures de type «grappe»

Dans le cas des grappes, les défis se situent à plusieurs niveaux : outre les caractéristiques des machines-cibles, il faut également prendre en considération les particularités des applications destinées à les exploiter ainsi que les attentes des développeurs de systèmes ou

d'applications et des utilisateurs finaux.

La nature des grappes dont il est question ici a déjà été exposée précédemment : ce sont essentiellement les grappes de grappes et les grappes de grande taille (cf. 1.1.1.4). Il s'agit donc de configurations à la fois hiérarchiques, hétérogènes mais aussi évolutives, car la nature des composants peut changer au cours du temps (par exemple, migration de machines multi-processeurs vers des machines à accès mémoire non-uniformes (*Non-Uniform Memory Access* ou NUMA)).

Quant aux applications, elles ont besoin d'un support pour la dynamicité mais aussi de généricité. Nous souhaitons pouvoir utiliser un schéma de communication inter-grappes variable (direct ou avec retransmissions) afin que les applications puissent s'adapter et exploiter le mieux possible la configuration matérielle. La tolérance aux pannes se justifie avec le contexte de l'arrivée des grappes de grande taille et des applications à large échelle (aussi bien en temps comme en espace, cf. l'approche HTC en 1.1.1.2). De plus, l'objectif principal de MPI étant les performances, la façon dont sont gérées les communications ne doit pas avoir des répercussions négatives sur le temps de transfert et le débit.

Enfin, les utilisateurs veulent des systèmes à la fois riches (au niveau des configurations supportées et des fonctionnalités disponibles), flexibles (permettant une manipulation aisée des grappes) mais néanmoins simples pour le développement (ajout de nouvelles technologies de communication) et la configuration (quelques fichiers tout au plus).

3.1.1.2 Hiérarchisation de nos priorités

Nous avons procédé à un regroupement de ces exigences en diverses catégories selon la priorité que nous leur attachons. L'architecture proposée doit être en adéquation avec les points suivants :

1. **les objectifs du standard** : la contrainte se situe au niveau des performances de l'implémentation résultante qui doivent être à la fois de tout premier plan et portables sur une large palette de configurations. Ceci constitue ce que l'on attend prioritairement de MPI.
2. **les services recherchés** : nos objectifs sont triples en matière de fonctionnalités puisque l'implémentation instanciant l'architecture proposée doit permettre l'exploitation des configurations hiérarchiques, hétérogènes et dynamiques. Ceci représente le cœur de notre travail et doit permettre d'être en accord avec le point précédent pour de nouvelles catégories de configurations.
3. **les attentes des utilisateurs** : l'implémentation obtenue doit être extensible aisément tout en demeurant simple dans son utilisation mais complète au niveau des configurations supportées.

Nous n'avons pas la prétention de vouloir considérer l'ensemble de ces aspects dans le cadre de cette thèse, et avons choisi de traiter prioritairement quelques uns d'entre eux comme la hiérarchie, l' hétérogénéité et la dynamicité. Nous en avons intégré d'autres dans la réflexion, par exemple l'extensibilité de l'architecture, sa souplesse ou encore sa simplicité. Certains points, en revanche, n'ont pas été abordés pour le moment, notamment la tolérance aux pannes.

3.1.2 Les raisons de l'absence actuelle de solutions

L'état de l'art nous a fourni une liste de solutions que nous avons décortiquées afin de retenir les points forts des implémentations actuelles et d'en éliminer les points plus faibles. Cette liste constitue un gisement d'idées à exploiter. Nous allons d'abord expliquer pourquoi nombre de solutions n'ont pas conduit à des résultats satisfaisants, pour voir ensuite les leçons que nous pouvons en tirer.

3.1.2.1 Des fonctionnalités imprévues

En premier lieu, il est évident que les problèmes que nous cherchons à résoudre (i.e gestion de la hiérarchie, de l'hétérogénéité et de la dynamicité) n'ont pas été pris en compte dès l'origine (cf. 1.1.2.3, le *MPI Paradox*). De nombreuses solutions consistent à ajouter les fonctionnalités recherchées à des implémentations génériques pré-existantes et le support de ces multiples aspects revient à procéder par ajouts successifs.

Notre démarche est différente car nous avons décidé de mettre en place une architecture dans laquelle il est possible d'intégrer de nouvelles fonctionnalités simplement et sans remettre en cause les performances de celles déjà présentes.

3.1.2.2 «Une inadaptation des processus légers pour les communications ?»

D'autres raisons expliquent le manque de solutions répondant à notre cahier des charges. Notamment une certaine suspicion à l'égard d'outils logiciels considérés comme inefficaces. Le cas des processus légers est éloquent puisque en dépit des services potentiels qu'ils pourraient rendre dans la gestion des communications, peu de projets y ont recours. MPI/I-WAY (cf. 2.1.4.3) a sans doute contribué au discrédit de l'approche mêlant communications et processus légers.

Les processus légers sont souvent utilisés pour effectuer des scrutations ou des réceptions de messages. Le problème qui se pose est que pour recevoir effectivement des données, le processus léger récepteur doit être ordonnancé. Comme cet ordonnancement n'intervient qu'à la fin d'un quantum de temps, alors la durée de ce quantum se répercute sur le temps de transfert minimal. De plus, dans le cas de processus légers de niveau noyau, le coût d'un changement de contexte n'est pas négligeable car faisant appel au système d'exploitation. Cependant, un travail sur l'ordonnancement des processus légers pourrait permettre de limiter ces désagréments et l'emploi d'une bibliothèque de niveau utilisateur diminuera le coût d'un changement de contexte.

3.1.2.3 Un empilement trop opaque de couches logicielles

La volonté de réutiliser des éléments logiciels pré-existants entraîne mécaniquement des empilements afin d'arriver aux objectifs désirés. L'exemple typique est celui des approches inter-opérables (cf. 2.1.3) qui exploitent des versions spécialisées de MPI afin d'obtenir de bonnes performances et y rajoutent une surcouche.

Si cette démarche basée sur l'empilement existe, elle connaît tout de même des limites car les développeurs sont réticents à l'utilisation de couches qu'ils ne peuvent totalement

maîtriser, en particulier en ce qui concerne les communications. En effet, l'empilement de couches ajoute des surcoûts de traitement pour les données à cause de l'encapsulation des messages de la couche inférieure dans ceux de la couche supérieure (en général, rajout d'un en-tête). Ce surcoût était sensible dans le cas des bibliothèques de communication construites au-dessus de protocoles bas-niveau et l'utilisation de ces bibliothèques avait pour effet de ralonger le chemin critique d'émission des messages. De plus, l'introduction d'une couche de portabilité au-dessus de protocoles distincts pouvait entraîner une perte potentielle de fonctionnalités.

Cependant, les progrès acccomplis par les bibliothèques actuelles de communication comme Madeleine ([Aum02b]) permettent de réviser ce jugement : il devient possible d'avoir une interface générique masquant la couche basse mais permettant une exploitation optimale du matériel sous-jacent. Ceci est rendu possible par un travail sur l'interface et l'exportation des fonctionnalités à un niveau supérieur.

Nous pensons donc que l'empilement de couches génériques cachant les interfaces bas-niveau n'est plus actuellement un facteur de ralentissement – comme c'était le cas auparavant – et que cette approche permet d'obtenir des résultats satisfaisants.

3.1.2.4 MPI comme unique outil

Enfin, une dernière erreur a peut-être été de croire que MPI était une solution universelle, pour ne pas dire *la* solution, à un panel trop large de problèmes, aussi bien au niveau des configurations matérielles que des applications. Ces problèmes dépassent peut-être le champ d'action pour lequel MPI a été conçu à l'origine. Par exemple, le passage d'une configuration de type «supercalculateur» vers une autre de type «grille de calcul» n'est peut être pas très adapté pour une bibliothèque de communication comme MPI. En effet, des mécanismes d'authentification, de sécurité ou encore de gestion et d'allocation de ressources ne sont pas spécifiés et pourtant ils sont incontournables lorsqu'on travaille à cette échelle.

3.1.3 Enseignements retenus

L'analyse des erreurs commises ainsi que des solutions existantes nous permet de tirer les enseignements suivants.

3.1.3.1 Une nécessaire adaptation au contexte

Il est indispensable de concevoir des outils spécialisés dans l'exploitation de configurations bien déterminées. Ces outils doivent néanmoins être conçus pour une utilisation dans un cadre plus large, mais cela implique une intégration dans un système englobant et non une extension de la solution d'origine. De façon symétrique, il n'est pas non plus souhaitable d'utiliser dans un contexte réduit des outils conçus pour une large échelle.

Nous pensons une fois encore aux approches de type inter-opérables, conçues pour l'exploitation des grilles de calcul, et utilisées pour les grappes hétérogènes et les grappes de grappes. Il est nécessaire de conserver les outils pour l'utilisation prévue.

3.1.3.2 Une intégration poussée des composants

Le travail pour tout repenser et réécrire étant très important, la réutilisabilité des logiciels est donc nécessaire. Ce principe est au cœur des solutions de type inter-opérables (cf. 2.1.3). Cette réutilisation n'est pas critiquable en soi, mais doit s'accompagner de précautions quant aux éléments empruntés. Le principe consistant à rajouter des surcouches ne nous paraît pas souhaitable car cela manque d'extensiblité.

Par exemple, dans le cas de la hiérarchie, les solutions inter-opérables permettent de gagner un nouveau niveau mais que faire quand vient le temps d'un nouvel ajout ? Si l'approche inter-opérable n'est pas convenable, l'approche intégrée, en revanche, est celle qui permettra de résoudre nos problèmes convenablement car prenant en compte les mécanismes intrinsèquement[1].

C'est en ayant une telle approche à l'esprit que l'intégration doit être réalisée (par exemple, les communications et le multithreading). De plus, il s'agit de la seule voie possible si l'on désire introduire ce travail dans un système plus large (cf. 3.1.3.1).

3.1.3.3 Une prise en compte de problèmes indépendants

Enfin, il est nécessaire de mettre en place des mécanismes suffisamment flexibles et adaptables pour permettre l'intégration de nouvelles fonctionnalités même si la version de départ n'était pas explicitement prévue pour.

Ainsi, la mise en place de techniques pour gérer l'hétérogénéité ne devrait pas constituer un frein pour la réalisation de support pour la gestion dynamique des processus ou encore la tolérance aux pannes. Ces fonctionnalités doivent être complémentaires et ne pas s'excluent mutuellement.

Nous en profitons pour affirmer que les aspects d'hétérogénéité et de dynamicité sont découplés. Le problème difficile est de réaliser des communications en environnement hétérogène et si l'on en est capable avec un mode de fonctionnement statique alors la migration vers un mode dynamique ne pose pas de difficulté pour le cœur du travail, mais plutôt à la marge, au niveau du lancement, des connexions et de la numérotation des processus. Ces aspects ne sont pas négligeables et soulèvent de nombreuses questions mais ils peuvent être traités de façon quasi-indépendante des problèmes de hiérarchie ou d'hétérogénéité. D'ailleurs, aucun des systèmes examinés dans l'état de l'art n'impose un modèle statique pour apporter des solutions au double problème de la gestion de la hiérarchie et de l'hétérogénéité.

3.1.4 Démarche de recherche

Les deux sections précédentes nous permettent de préciser la démarche de recherche qui est la nôtre. Nous avons travaillé sur les points de blocage constatés dans les solutions

[1]Ceci ne constitue pas un problème, car nous ne nous sommes pas placés dans un cadre inter-opérable. Une de nos hypothèses de travail est que l'ensemble des applications lancées sur les grappes le sera avec notre version de la bibliothèque. On ne cherche donc pas à pouvoir faire communiquer deux implémentations distinctes de MPI.

actuelles et empêchant la réalisation d'une implémentation de MPI pour les configurations hiérarchiques, hétérogènes et dynamiques.

3.1.4.1 Deux axes fondamentaux

Nous prenons ainsi le contre-pied de bien des solutions existantes en établissant les deux principes suivants comme axes directeurs de notre démarche :

Principe 1 :

> **le multithreading n'est pas incompatible avec des communications efficaces**, bien au contraire il peut permettre une amélioration de leur traitement.

Principe 2 :

> **l'empilement de couches logicielles à bas-niveau n'est pas foncièrement un facteur de dégradation des performances** si les interactions entre les couches logicielles sont bien pensées.

Nous verrons par la suite que cela a des conséquences importantes au niveau de l'architecture et du fonctionnement de la solution. Il est donc essentiel pour nous de pouvoir disposer des outils adéquats pour mener à bien ce travail et de demeurer vigilants quant à leur choix (cf. 4.2).

3.1.4.2 Des aspects importants mais souvent négligés

Mis à part ces deux axes fondamentaux, d'autres points nous semblent essentiels et l'analyse des solutions existantes montre qu'ils ne sont pas systématiquement pris en compte dans les implémentations actuelles.

La portabilité des performances Nos cibles prioritaires sont les grappes hétérogènes et les grappes de grappes, mais nous voulons aussi obtenir des résultats satisfaisants sur des configurations plus classiques telles que les grappes homogènes de machines uniprocesseurs comme multi-processeurs. Cela démontrera le bien-fondé de notre démarche qui ne sera pas cantonnée à une classe trop restreinte de situations. La portabilité des performances montrera en outre que notre solution est souple et adaptable.

La réactivité des applications La réactivité des applications face aux évenements d'entrées-sorties est trop souvent laissée de côté dans les implémentations actuelles. Nous estimons que ceci est dommageable et qu'un gain de performances pourrait être obtenu avec des systèmes de communications plus réactifs. L'introduction du multithreading et l'utilisation des processus légers nous permettra d'arriver à éliminer les problèmes constatés dans une majorité de solutions existantes.

Des appels non-bloquants asynchrones Également, l'implémentation des appels non-bloquants dans MPI révèle quelques surprises puisqu'une fraction non-négligeable des programmeurs d'application confond le fait d'être non-bloquant avec l'asynchronisme. D'un point de vue opérationnel, cela ne change rien, mais du côté des performances, les choses sont tout autres. C'est l'incapacité des implémentations actuelles à pouvoir effectuer un recouvrement des communications par du calcul qui est à l'origine de cet état de fait. L'utilisation de processus légers va une fois encore nous permettre d'améliorer l'existant (cf. 5.3.3)

3.1.4.3 Les compromis résultants

Notre démarche entraîne de devoir effectuer des choix, car certaines fonctionnalités introduites ont fatalement un impact négatif sur les performances que nous voulons bien mesurer et limiter.

Limites de la réutilisabilité Le problème principal avec une démarche telle que la nôtre réside dans l'utilisation d'éléments pré-existants car l'introduction du multithreading ne permet pas de reprendre tels quels les travaux actuels : des modifications souvent profondes – car touchant directement à l'architecture – se révèlent nécessaires. Si le principe de l'introduction des processus légers est un choix pour lequel il est facile d'opter, la transition vers une réalisation concrète est plus ardue à négocier. Il existe en effet une abondante littérature ([Pro96],[PSH96], [PS98], [PD96]) se focalisant plus particulièrement sur le développement d'une implémentation de MPI à base de processus légers et ces articles sont très précis et constituent une base de départ fort intéressante pour entreprendre un tel travail. Mais en dépit de constats souvent pertinents et argumentés, les auteurs n'ont semble-t'il pas réussi à concrétiser leurs observations[2], ce qui montre que le problème est techniquement difficile.

Latence contre réactivité Un autre point méritant d'être signalé est que l'utilisation de processus légers pour la gestion des communications ne permet pas en théorie d'obtenir des temps de transfert très bas (cf. 3.1.2.2), notamment en raison du coût des changements de contexte et de l'ordonnancement. Si l'utilisation d'une bibliothèque de niveau utilisateur permet de diminuer ce phénomène, il ne disparaîtra pas totalement pour autant. Il s'agit là d'un choix : celui du support de fonctionnalités très intéressantes contre une perte minime de performances dans le cas de messages de faible taille. Nous verrons néanmoins qu'il est possible de se situer à des niveaux similaires à ceux d'implémentations de MPI très optimisées pour une technologie réseau particulière (cf le chapitre 5). Enfin, il convient de préciser que ceci est contrebalancé par le fait que la réactivité est largement améliorée.

3.1.4.4 De l'interaction avec les utilisateurs

Nous intégrons aussi à notre démarche des aspects moins techniques mais qui nous apparaissent capitaux pour la réussite : notre philosophie générale de l'interaction entre le système et l'utilisateur s'articulera donc autour des deux points suivants.

[2] Nous subodorons néanmoins que les auteurs participent au développement de chaMPIon/Pro.

L'utilisateur conserve un contrôle total le système s'efforçe d'offrir des fonctionnalités et des services à l'utilisateur tout en assurant les meilleures performances possibles et ce de façon portable. Pour cela, il est amené à effectuer des choix, mais ces derniers doivent *toujours* pouvoir être remis en cause par l'utilisateur directement (par exemple au lancement, par le truchement de données de configuration) ou indirectement par le biais de l'application qu'il aura écrite (par exemple des communications différentes selon les partitions).

Le respect des standards Enfin l'introduction de nouvelles fonctionnalités doit se faire en respectant totalement le standard. Ce respect pourra s'obtenir en utilisant au maximum les fonctionnalités et objets pré-existants. Pour ce faire, nous sommes résolument partisans de l'approche de type «extension de l'interface», moins préjudiciable pour la portabilité que le changement de sémantique. L'avantage d'une telle approche est souligné dans [LG94] pour la gestion dynamique des processus par exemple.

3.2 Proposition d'architecture

Notre proposition d'architecture permettant de répondre aux exigences des architectures de type «grappes» est décrite dans cette partie. Cette architecture est le prolongement de notre démarche et tient compte des points formulés précédemment.

Signalons d'ores et déjà que cette architecture est strictement indépendante d'une implémentation particulière de MPI car les concepts exposés ici ne pré-supposent rien quant à cette dernière. Ainsi, ce travail pourra être appliqué à toute mise en œuvre (libre) du standard, bien qu'en pratique une telle tâche puisse se révéler quantitativement très importante. De même, cette architecture est utilisable *a priori* avec toute bibliothèque de communication de bas-niveau même s'il nous semble important qu'une réelle intégration existe entre cette bibliothèque et les couches supérieures l'utilisant pour l'implémentation de MPI.

3.2.1 Exigences architecturales

Les propriétés de notre architecture sont décrites dans les parties suivantes. D'après ce que nous avons dit précédemment, il apparaît que certains éléments sont incontournables et notre solution s'articule au niveau architectural autour des points suivants. Nous voulons :

1. de la flexibilité pour la mise en œuvre et l'utilisation ;
2. des processus légers pour améliorer la réactivité des applications ;
3. d'une progression des communications qui soit indépendante du déroulement des applications ;
4. d'une bibliothèque de communication de bas-niveau exploitant efficacement les différents réseaux sous-jacents.

3.2.1.1 Une architecture modulaire

Ayant écarté les solutions inter-opérables (cf. 3.1.3.2), nous avons donc opté pour une architecture modulaire pour la gestion des communications. Mais il ressort de l'analyse des

solutions de ce type que l'approche multi-modulaire (cf. 2.1.4.2) souffre d'un manque d'extensibilité car l'ajout de nouveaux protocoles – facile par ailleurs – se ressent sur les performances globales.

Quant à l'approche unimodulaire (cf. 2.1.4.3), elle masque par trop la topologie sous-jacente, que l'on cherche à mettre à la disposition explicite de l'application et de l'utilisateur. Le dilemme est donc d'avoir une architecture conservant une transparence vis-à-vis de la topologie comme dans le cas multi-modulaire tout en ayant une gestion de la scrutation performante comme dans le cas unimodulaire.

En fait, notre solution se situe à mi-chemin entre ces deux approches et se fonde sur les éléments logiciels suivants (cf. Figure 28) :

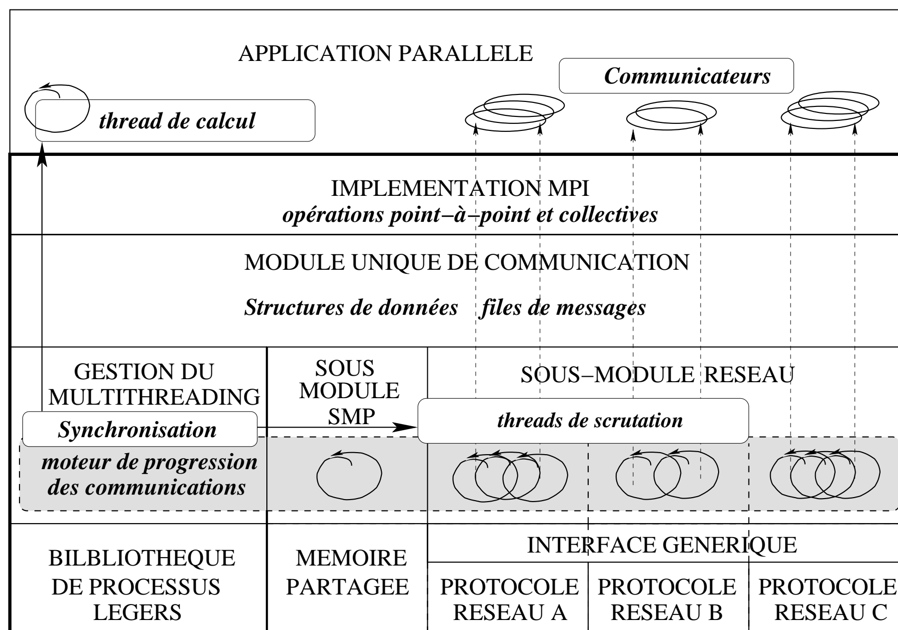


FIG. 28 – Architecture proposée

- une couche de haut-niveau, mettant en place les objets fondamentaux et fonctionnalités de MPI s'appuyant sur d'autres plus simples (par exemple, les opérations collectives vont utiliser les communications point-à-point, tout comme les envois tamponnés) ;
- un *unique* module de communication, exploité par la couche de haut-niveau et qui est responsable de la factorisation de l'accès à certains services pour les sous-modules de communication, comme les files de messages par exemple ;
- des sous-modules de communication, qui co-existent au sein de ce module. Ces différents sous-modules prennent en charge l'ensemble des protocoles de communication sous-jacents. Actuellement n'existe qu'un couple de ces sous-modules : le sous-module responsable des communications par mémoire partagée et celui gérant les communications réseaux proprement dites ;
- un module gérant les processus légers, c'est-à-dire les différentes opérations de synchronisation et la scrutation par le moteur de progession des communications (cf. ci-dessous). Ce module est fondé sur une bibliothèque de processus légers ;

– une interface générique de communication masquant les différents protocoles réseaux (par exemple A pourra être TCP, B SCI et C, GM).

Cette structure est aisément extensible, comme dans le cas multi-modulaire, sans toutefois connaître les désagréments d'une scrutation inefficace, ainsi que nous le verrons en 3.2.1.3. De plus, la présence de sous-modules distincts autorisant une identification des instances des protocoles de communication permet d'exporter facilement la topologie vers l'application. Ces sous-modules utilisent concrètement le système de progression des communications décrit en 3.2.1.3.

3.2.1.2 Une architecture multithreadée

Bien que nous fassions appel à l'outil «processus légers», notre architecture ne s'inscrit pas dans un schéma hybride en ce qui concerne les communications intra-nœuds, car cette approche manque de généricité (trop de dépendance vis-à-vis du couple configuration-application).

Utilisation pratique des processus légers La progression des communications repose sur l'utilisation de processus légers et le principe est le suivant : la scrutation des communications pour une instance d'un protocole est confiée à un processus léger dédié. Cet emploi implique également que ces opérations de scrutation puissent être effectuées non pas dans la couche haute de MPI (défaut des approches multi-modulaires) mais à un niveau plus bas (comme dans l'approche unimodulaire), l'ordonnanceur jouant le rôle de «chef d'orchestre» (cf. ci-après 3.2.1.3).

Ces processus légers sont internes à l'architecture et ne sont donc pas visibles au niveau applicatif. Ils sont créés à l'initialisation par le processus léger principal se chargeant de la progression du calcul pour l'application.

Enfin, les processus légers constituent un élément pratique permettant la prise en compte des événements de modification de sessions (expansion notamment) sans avoir à utiliser un processus-démon devant se synchroniser de façon plus lourde avec les processus applicatifs MPI.

Apports du multithreading dans la progression des communications En dépit du fait que le multithreading et les communications soient difficiles à faire collaborer efficacement, les apports sont fort intéressants pour un système de communication tel que MPI :

– c'est une approche qui est portable. En effet, des techniques courantes utilisent du code embarqué sur les cartes réseaux dans une perspective d'optimisation. Il peut s'agir d'opérations collectives réalisées de façon matérielle ou encore d'éléments permettant une meilleure progression des communications. Ces approches sont intéressantes (cas de Myrinet ou de Quadrics) mais la nôtre, certes moins efficace car basée sur du logiciel, permet d'utiliser une progression motorisée des communications avec tout type de matériel ;
– cela permet de mettre en place facilement une gestion simultanée et efficace de multiples protocoles de communication potentiellement très différents et utilisant des mécanismes divers comme de la scutation ou des interruptions pour signaler l'arrivée

d'un message. Nous offrons une vision unifiée en cachant la véritable nature de ces mécanismes ;

- cela permet enfin, comme déclaré précédemment d'avoir une réactivité accrue de l'application vis-à-vis des communications.

3.2.1.3 Une amélioration du modèle original de progression

Dans MPI, une application est partagée entre deux mondes : les phases de calcul et celles de communication. Ces phases sont entrelaçées et ne se mélangent pas. Concrètement, pour faire progresser les communications, il est indispensable de faire appel à une fonction de la bibliothèque MPI. Une conséquence est que lors d'une phase de calcul très longue, plus aucune communication n'est possible. Il faut donc arriver à dissocier la progression des communications des appels aux fonctions MPI. Tout au plus, ces fonctions ne devraient que les initier. Ce constat justifie le point suivant.

Indépendance des communications vis-à-vis de l'application Nous recherchons à mettre en place des communications dont la progression ne soit pas soumise aux aléas du déroulement de l'application. Une fois encore, les processus légers nous apportent la réponse. La gestion de la progression sera en effet déléguée à un élément logiciel dédié : le **moteur de progression**.

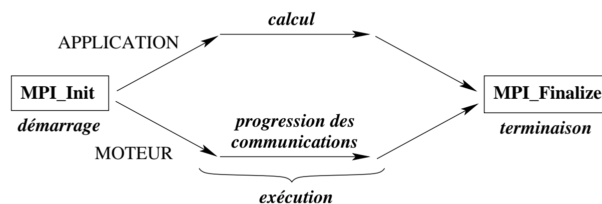


FIG. 29 – Cohabitation moteur/application

Ce moteur est mis en place de façon interne par les sous-modules de communication et son exécution est découplée de celle de l'application, même si – bien entendu – ces deux éléments interagissent constamment. La figure 29 illustre ce principe. Il existe donc dans une application écrite avec notre version de MPI deux entités cohabitant et s'exécutant parallèlement. Le point le plus important étant que même dans le cas où l'application est stoppée, les communications continuent de progresser, ce qui n'était pas possible auparavant. Enfin, comme ceci est effectué à bas niveau, un travail sur l'ordonnanceur permet de faire progresser des messages pour différents communicateurs qui utilisent le même medium physique : les requêtes de scrutation peuvent en effet être «factorisées».

La notion-clef : le moteur de progression Le moteur de progression est donc constitué par l'ensemble de ces processus légers de scrutation et permet une progression découplée du déroulement de l'application. Le code de l'application sera quant à lui exécuté par un autre processus léger et les interactions évoquées au paragraphe précédent seront en pratique des

synchronisations entre ces différents processus. L'ordonnanceur agit directement sur le moteur et joue le rôle d'autorité commune permettant de départager les différents protocoles via les processus qui leur sont affectés. Cette autorité est donc localisée à un niveau relativement bas, si bien que même si les différents protocoles n'ont pas conscience de cohabiter avec d'autres, ils ne risquent pas de les pénaliser avec leur politique de scrutation.

3.2.1.4 Une exportation directe des nouvelles fonctionnalités

Le dernier point concerne l'exportation des nouvelles fonctionnalités vers l'interface MPI. Ce problème se pose plus particulièrement dans le cas de la gestion de l'hétérogénéité, car il faut permettre à l'utilisateur l'emploi d'un protocole particulier de façon simple. L'idée est la suivante : dans MPI, les processus sont regroupés au sein de «conteneurs» appelés les communicateurs (*communicators*) et chaque opération de communication s'effectue à l'échelle d'un tel communicateur. En pratique, ceci permet un multiplexage des communications dans MPI. Ce mécanisme est ici détourné pour permettre la gestion de la topologie, car nous lions chaque communicateur à un protocole particulier. Ainsi, les processus utilisant entre eux une technologie réseau déterminée forment un groupe que nous assimilons à un communicateur. Au niveau applicatif, échanger des messages par le biais d'un communicateur revient à utiliser un protocole particulier. Ainsi, nous ne modifions pas la sémantique de ces objets et restons parfaitement dans le cadre du standard, car n'importe quel programme MPI pourra fonctionner au-dessus de l'implémentation de cette architecture.

3.2.1.5 Discussion

Finalement, cette architecture est le résultat de décisions qui sont discutées dans les paragraphes suivants.

Unicité de l'architecture Cette architecture découle naturellement des choix faits précédemment dans le cadre de notre démarche. Elle tient également compte des enseignements que nous avons tirés de l'analyse des solutions actuelles. L'architecture permet bien la gestion simultanée de protocoles distincts et la progression des communications est effectivement indépendante de celle de l'application. Ces aspects sont intégrés nativement dans la solution que nous proposons. Cependant, des alternatives existent, notamment en ce qui concerne la mise en place du moteur de progression (MPICH2, par exemple). Enfin, l'utilisation des processus légers n'est pas pensable dans toutes les situations[3].

MPI et le multithreading La problèmatique de l'introduction des processus légers dans MPI est assez ancienne et date presque de la publication du standard lui-même. Il n'est donc guère étonnant de trouver dans la littérature un nombre significatif de publications consacrées à ce sujet. Ces articles se divisent en deux catégories :

– soit les processus légers sont utilisés dans une optique d'exploitation des machines multi-processeurs. C'est le cas de MT ([Rad97]), TOMPI ([Dem97]), MiMPI ([GCC99]),

[3]Des machines comme le RedStorm ou l'IBM BlueGene ne possèdent pas de bibliothèque de processus légers par exemple . . .

TMPI ([TY01]) ou encore AMPI ([LBK02]). Ces solutions ont déja été examinées dans l'état de l'art (cf. 2.1.2.1) ;

– soit les processus légers sont employés pour l'implémentation de MPI proprement dite, et dans ce cas, cet emploi n'est pas forcément visible au niveau applicatif. C'est le cas de solutions telles que MPICH/SCore avec MPC++ ([OHT+96]) ou de chaMPIon/Pro [CHAa]. Nous penchons naturellement pour des solutions de ce type.

Il faut de plus signaler l'existence d'une série d'articles ([Pro96],[PSH96], [PS98] et [PD96]) exclusivement consacrés à l'introduction du multithreading dans MPI et plus particulièrement dans l'implémentation MPICH (cf. 3.1.4.3). En l'occurence, la solution décrite est conceptuellement la plus proche de la nôtre : les auteurs ont analysé les défauts de MPICH quant aux processus légers et tirent des conclusions similaires aux nôtres (cf. [PS98] et 4.1.2). De même, ils se proposent d'utiliser le multithreading pour effectuer une scrutation des communications plus performante tout en étant indépendante du déroulement de l'application. Enfin, les processus légers sont présentés comme l'outil de choix pour gérer un ensemble de protocoles réseaux différents.

Opacité et performances Enfin, la dernière question que l'on est en droit se se poser est celle des performances. Puisque l'interface de communication sous-jacente masque les protocoles de communication de bas-niveau, comment serons-nous à même d'obtenir de bonnes performances puisque nous n'aurons plus accès aux interfaces de bas-niveau qui permettent justement d'utiliser les mécanismes particuliers du matériel ? Nous avons déjà répondu à cette interrogation en 3.1.2.3 : un travail sur les interfaces et leur expressivité permet – même dans le cas où elles sont génériques – d'obtenir de bonnes performances.

3.2.2 Des communications performantes sur réseaux hétérogènes

Cette architecture devrait permettre d'obtenir des performances satisfaisantes aussi bien dans un cas homogène qu'hétérogène. Cependant, dans ce dernier cas de figure, certaines questions méritent d'être posées.

3.2.2.1 Un réseau hétérogène peut-il offrir de hautes performances ?

La question qui nous semble centrale est celle des performances possibles dans un tel cas de figure. Nous avons vu dans l'état de l'art qu'une minorité de systèmes n'utilisaient pas le protocole universel TCP/IP pour réaliser leurs communications inter-grappes. Le recours à ce plus petit dénominateur commun est-il le fruit de la résignation ou bien guidé par le pragmatisme ?

Toutefois, il ne nous semble pas judicieux d'attendre une migration totale vers des réseaux de type GigaBitEthernet, migration qui d'ailleurs ne sera sûrement pas complète en raison du coût d'une telle opération. De plus, il existera toujours des protocoles exhibant de meilleures performances que celles de TCP et il serait contre-productif de s'en priver.

L'utilisation de TCP nous semble venir d'un constat simple : une exploitation efficace des réseaux hétérogènes n'est possible qu'avec le développement de passerelles logicielles. La tâche est ardue parce que ces passerelles doivent être capables de retransmettre des messages

d'une technologie réseau vers une autre et que la multiplicité de ces dernières ajoute un degré de complexité supplémentaire.

Cependant, nous pensons que la mise en place de telles passerelles permet d'obtenir un gain important pour les performances des communications inter-grappes et que cette voie, certes difficile, ne doit pas être écartée, comme c'est encore trop souvent le cas actuellement. Enfin, la maîtrise de ces mécanismes de bas-niveau ouvre des perspectives dans des domaines comme les communications collectives ou encore la fragmentation des messages sur plusieurs réseaux.

3.2.2.2 Quel routage dans les réseaux hétérogènes ?

Une fois acquis le principe que les passerelles et le routage font partie intégrante de la solution, encore faut-il savoir de quel type de routage s'agit-il et surtout à quel niveau dans l'architecture doit-il être mis en place ?

Le choix du niveau : un compromis entre efficacité et transparence Deux alternatives se posent à nous, comme le montre la figure 30 :

- soit la retransmission est faite à haut-niveau, c'est-à-dire au niveau applicatif proprement dit. Dans ce cas, ce sont des processus MPI qui jouent le rôle de routeurs et effectuent les retransmissions nécessaires. C'est l'approche adoptée par une solution telle que MetaMPICH (cf. 2.1.5.2). Une variante consisterait à effectuer ce travail au niveau du module de communication sur lequel repose la couche haute de MPI, mais à l'heure actuelle, aucune solution ne se réclame de cette approche ;
- soit la retransmission des messages est effectuée à bas-niveau, c'est-à-dire dans la bibliothèque de communication utilisée.

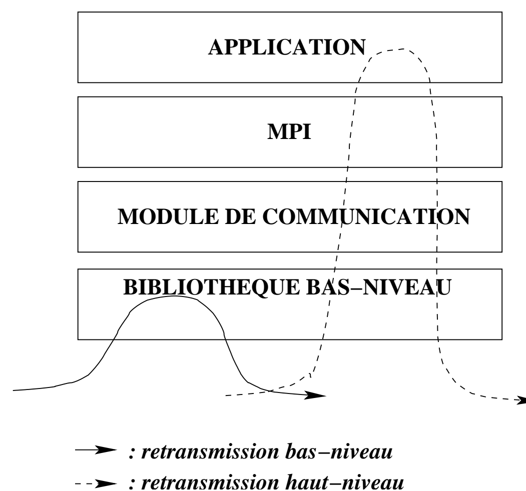


FIG. 30 – Deux niveaux pour le routage

Les deux approches ont leur lots d'avantages et d'inconvénients. Dans le cas d'une approche haute, la politique de routage est modifiable directement au niveau applicatif par

l'utilisateur qui peut agir comme bon lui semble. Le routage peut donc changer au cours de l'exécution de l'application, ce qui permet de tenir compte de l'engorgement des routeurs très sollicités quand toutefois cette fonctionnalité est disponible. Le défaut principal de cette approche, surtout lorsque nous sommes dans un contexte multithread est que la retransmission ne peut se faire que sur des messages entiers. Il n'est pas possible de réémettre des fragments à la volée et de créer des effets de pipeline avec les processus légers.

Dans le cas d'une approche basse, il est possible de procéder à de tels envois pipelinés, mais le problème est que la politique de routage devient moins contrôlable par l'utilisateur qui doit s'en remettre au système bas-niveau, même s'il a la possibilité de lui dicter ses choix en lui passant des informations par le biais de fichiers de configuration par exemple. Également, dans le cas de processeurs hétérogènes, les routeurs devront être modifiés, ce qui n'était pas le cas des routeurs au niveau applicatif, car ils connaissent la composition des messages grâce aux types MPI. Cependant, une approche basse est plus performante qu'une approche haute car le surcoût est moindre : il faudrait en effet «sortir» de la bibliothèque de communication et modifier les tampons deux fois, avec des recopies ([AEN01]).

Ainsi, l'approche basse est la plus compatible avec notre objectif de conservation d'un débit élevé pour les communications entre deux processus appartenant à deux grappes hétérogènes.

Routage statique contre routage dynamique Une autre question réside dans la nature du routage : celui-ci doit-il être dynamique ou bien statique ? Si le premier est plus souple (par définition) et permet de prendre en compte la charge des passerelles, le second a l'avantage de la simplicité de la mise en œuvre. De plus, si l'on considère que les routeurs logiciels sont une source de perte de capacité de calcul (quand on retransmet les messages, on ne fait pas de calcul) alors ces pertes pourraient être majorées en cas de routage dynamique (redétermination des routes). Il faut également choisir à quelle fréquence le recalcul doit intervenir, ce qui est loin d'être trivial. Une solution, dans un contexte où les sessions sont fusionnables et séparables, serait de ne modifier les routes que lors de ces changements majeurs de configurations (une sorte de routage semi-statique).

3.2.3 Des sessions dynamiques sans impact notable pour les performances

Nous abordons maintenant le deuxième point essentiel dans notre travail, c'est-à-dire la gestion de la dynamicité. Nos objectifs sont de pouvoir permettre à des applications en cours d'exécution de pouvoir se regrouper ou se séparer.

Prenons par exemple le cas d'une application s'exécutant depuis plusieurs heures et dont on désire modifier le jeu initial de données. Pour ce faire on lance une seconde application de pilotage qui va se fixer sur l'application de départ (*computational steering*). Cette opération est une fusion des deux sessions et une fois le travail accompli, la partie «pilotage» peut se séparer de la partie «calculs» pour la laisser poursuivre son exécution.

La mise au point de telles applications dans MPI est encore difficile, notamment en raison du manque de disponibilité d'implémentations du standard MPI-2, qui définit l'interface pour la gestion dynamique des processus. Qui plus est, l'interface connaît en l'état des limitations pour arriver à un tel résultat facilement.

Nous allons donc aborder successivement l'approche que nous défendons pour offrir de telles fonctionnalités, qui est basée sur la notion de session plutôt que celle de processus applicatifs. Nous examinerons les conditions d'application et le cadre d'utilisation avant de montrer la compatibilité de l'ensemble vis-à-vis de l'architecture définie.

3.2.3.1 Avantage des sessions sur le modèle client-serveur

Le standard MPI-2 offre plusieurs approches dans le domaine de la gestion dynamique des processus. La première approche est très nettement orientée client-serveur, avec des fonctions telles que `MPI_Open_port`, `MPI_Close_port`, `MPI_Comm_accept` et `MPI_Comm_connect`. Pour le développement de programmes, cela complique quelque peu la tâche car dans l'hypothèse où deux applications MPI (c'est-à-dire deux sessions différentes) voudraient communiquer, l'ensemble des processus devra procéder à de nombreuses opérations de connexions afin qu'ils soient tous reliés deux à deux.

La seconde approche (`MPI_Comm_spawn`) correspond plus à nos objectifs mais est limitée puisque elle ne permet qu'une réplication de l'application sur de nouveaux nœuds (extension) et l'ensemble des fonctionnalités de MPI n'est plus disponible pour la session résultante car on obtient à ce moment un ***intercommunicateur*** pour lequel les opérations collectives, par exemple, sont inopérantes.

Nous voulons donc obtenir un mécanisme intermédiaire, permettant de fusionner ou de séparer deux applications facilement tout en permettant une utilisation complète du standard. Nous nous dirigeons vers une approche ressemblante à celle de `MPI_Comm_modify`, décrite dans [LG94], avec moins de limitations.

3.2.3.2 Caractéristiques des applications fusionnables

Nous faisons un certain nombre de suppositions quant au contexte dans lequel ces fusions peuvent s'effectuer : au niveau du lancement des applications, nous allons nous appuyer sur un programme dédié (un lanceur), ce qui correspond mieux au cas de figure que nous imaginons. Il s'agit du premier signe de notre approche orientée «session» et non «processus».

Ensuite, nous allons considérer que ces opérations de fusion ou de séparation se produiront rarement au cours de l'exécution. Ainsi, nous ne sommes pas dans un contexte «volatile» où le rythme des connexions et des déconnexions est élévé. Ceci nous permet de mettre en place des mécanismes efficaces certes, mais ne cherchant pas à êtres optimaux en ce qui concerne la durée des opérations concernées.

Enfin, il est important de savoir quelle politique de numérotation est adoptée. Nous pensons qu'il existe une asymétrie dans les opérations de fusion et de séparation pour les sessions concernées. Si nous reprenons l'exemple de l'application de calcul et de visualisation, alors il nous semble valide de considérer comme prioritaire la première application, la seconde ne venant que s'y greffer par la suite. Ainsi nous souhaitons que les processus membres de la première session conservent leur rangs tandis que seuls ceux de la seconde les voient modifiés. Globalement, l'ensemble des numéros formera toujours un intervalle d'entiers consécutifs et commençant à zéro.

3.2.3.3 Intégration dans l'architecture proposée

La gestion de la dynamicité ne doit pas être trop intrusive par rapport aux autres fonctionnalités. En particulier, l'écart de performances entre des environnements statiques et dynamiques doit être négligeable.

Quant à l'intégration de la gestion dynamique des processus dans l'architecture proposée, il est important de décider à quel niveau elle doit intervenir. Nous avons décidé de ne pas introduire les mécanismes dans les couches hautes de l'architecture, mais au contraire d'étendre les fonctionnalités des couches de plus bas niveau, dans l'interface générique masquant les protocoles.

En particulier, les informations concernant les sessions seront centralisées et diffusées par le lanceur de programmes, si bien qu'il est indispensable pour les processus applicatifs d'être constamment à l'écoute de ce lanceur. Pour réaliser cette écoute, nous avons une fois encore recours aux processus légers. Nous avons montré qu'ils pouvaient efficacement et simplement gérer de multiples protocoles réseaux différents. Ajouter un tel processus (écoutant en pratique une socket TCP) selon les mêmes mécanismes que dans le cas de l'hétérogénéité ne sera pas très coûteux. Les processus légers peuvent donc agir pour mettre en place toute une panoplie de services très divers. Il faut cependant se garder d'une certaine inflation qui finirait par provoquer l'écroulement de l'ensemble.

3.3 Exploitation de configurations à plus large échelle

Bien que nos cibles privilégiées soient les grappes hétérogènes et les grappes de grande taille la question de l'échelle à laquelle ce travail se destine est légitime. En effet, pourquoi ne pas travailler sur une échelle plus large comme dans le cas des grilles de calcul ? Les principes mis en place peuvent-ils être réutilisés ou bien doit-on faire appel à de nouveaux ? Nous allons donner des éléments de réponse dans cette section.

3.3.1 À propos des niveaux de hiérarchie dans MPI

Nous avons étudié dans l'état de l'art des systèmes dont certains proposaient une gestion de la hiérarchie plus ou moins évoluée. Dans ce domaine, il n'y a pas de consensus très net, mais la tendance qui se dessine est tout de même celle d'un nombre fixe de niveaux dans la hiérarchie. Ce nombre est variable selon les solutions, et mis à part l'exception de MPICH-G2 qui considère quatre niveaux (cf. 2.1.3.4), la plupart des implémentations de MPI pour les grappes de grappes n'en considèrent que deux : les niveaux intra- et inter-grappes.

Nous pensons que cela est trop rigide et qu'il devrait être possible d'avoir un nombre variable de ces niveaux. L'erreur commise, selon nous, est de vouloir transférer complètement au système la connaissance qu'à l'utilisateur de la topologie. Peut-être serait-il plus efficace que le système de communication ne voit pas les niveaux dans leur globalité : une vision locale des choses serait suffisante du moment qu'un contrat est passé entre les différentes couches. Si une couche inférieure est destinée à l'exploitation d'une configuration particulière, alors la couche supérieure n'a pas besoin de connaître le détail de cette configuration. De même, la couche inférieure doit s'engager à fournir le meilleur service possible en

sachant que d'autres éléments dépendent d'elle.

3.3.2 Compatibilité et complémentarité des approches

Nous pensons résolument que notre système n'est pas un concurrent de ceux qui exploitent les grilles de calcul, mais un partenaire. Nous avons pensé notre architecture avec la complémentarité à l'esprit et non dans une perspective de compétition.

En effet, une exploitation efficace des grappes de grappes peut constituer une clef dans la maîtrise des grilles et à ce titre, les systèmes à plus large échelle devraient pouvoir s'appuyer sur le nôtre. Nous sommes persuadés qu'une meilleure exploitation des composantes locales apportera un gain pour les performances globales.

Une telle intégration dans des systèmes à plus large échelle peut s'accomplir de deux façons : soit nous restons dans l'univers de MPI et utilisons encore cette bibiothèque pour toutes les communications, soit nous décidons que les communications à longues distances ne sont plus du domaine de cet outil et confions à un autre ce travail. Ces alternatives sont décrites dans les paragraphes ci-dessous et montrent les efforts accomplis dans ce domaine.

3.3.2.1 Intégration de notre architecture dans MPICH-G2

Si l'on désire rester dans un univers strictement MPI, alors il nous semble judicieux de réutiliser des systèmes conçus pour une exploitation des configurations à large échelle. C'est le cas des architectures inter-opérables, dont nous avons mentionné l'inadéquation avec les configurations de types grappes de grappes car conçues prioritairement pour d'autres emplois. Dans cette optique d'une utilisation exclusive de MPI, MPICH-G2 est le seul candidat existant actuellement, même si de nouveaux systèmes s'inspirant de lui apportent une alternative (GridMPI, par exemple).

Nous reprenons tels quels les mécanismes de G2 et intégrons notre implémentation en tant que *Vendor MPI* sous-jacent (cf. Figure 31). Ceci est possible car nous avons scrupuleusement suivi les évolutions de MPICH et notre version du logiciel est par conséquent à jour pour permettre cette intégration[4].

Ceci constitue une illustration de la notion de contrat évoquée précédemment : le *Vendor MPI* est supposé le plus apte pour l'exploitation des grappes de grappes (communications intra-grappes), tandis que les communications à large échelle sont déléguées à G2. Cette approche est plus extensible que celle originalement adoptée par G2 car nous pouvons rajouter des niveaux dans le *Vendor MPI* et le niveau le plus haut est obtenu grâce aux mécanismes de MPICH-G2.

3.3.2.2 Intégration de notre architecture dans PADICOTM

La seconde approche est celle ne reposant pas uniquement sur MPI pour effectuer l'ensemble des communications. Là encore, la gestion des configurations locales sera confiée à notre implémentation tandis qu'un autre outil sera chargé des communications à plus large

[4] c'est pratiquement ce qui empêche de nombreux projets étudiés d'être intégrés dans G2 : la version de MPICH prise comme base de départ n'est pas mise à jour assez régulièrement

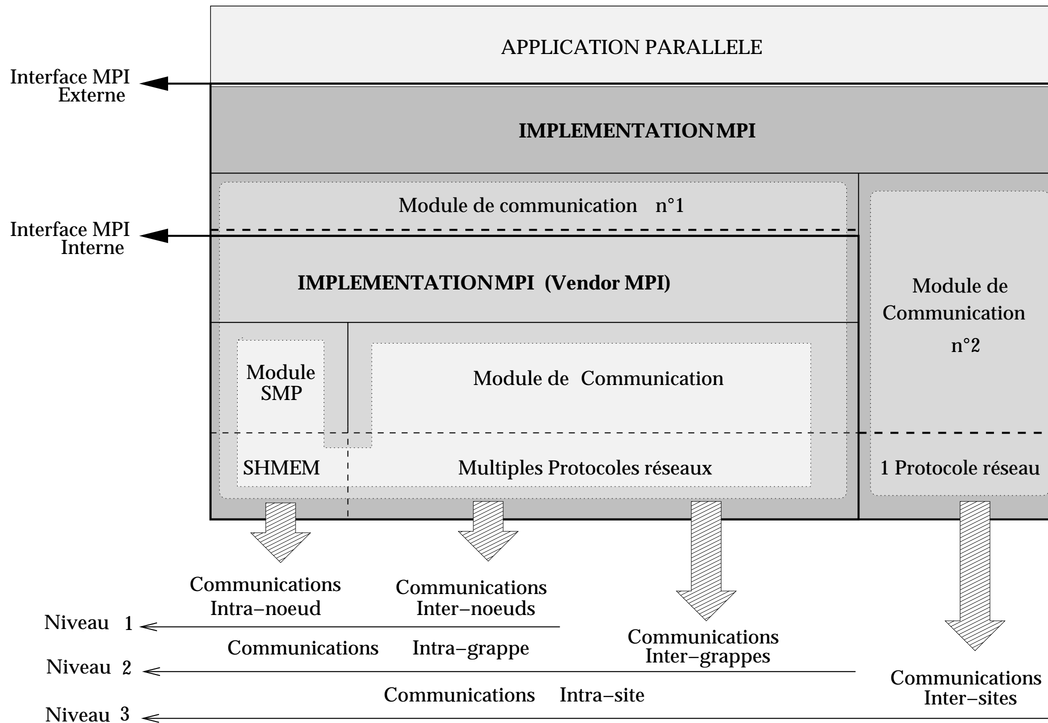


FIG. 31 – Une possibilité d'extension en utilisant notre architecture en tant que *VendorMPI* pour MPICH-G2

échelle. En l'occurence, il s'agira de CORBA, utilisé également pour mettre en place un service d'inter-opérabilité.

Le problème qui se pose alors est celui de l'accès à la couche de communication qui doit être partagé entre ces deux intergiciels. Cette problèmatique est au cœur du travail de thèse d'Alexandre DENIS ([Den03]) qui a développé le logiciel PADICOTM répondant spécifiquement à ces besoins. Conceptuellement, il n'y a pas de grande différence avec l'approche précédente d'un point de vue de l'implémentation sous-jacente de MPI, qui est utilisée dans la même perspective. Ce travail d'intégration a été effectué par les développeurs de PADICOTM, qui ont utilisé notre implémentation de MPI comme base de départ et y ont apporté des modifications minimes.

3.4 Bilan de la proposition

Quel bilan pouvons-nous tirer de notre architecture ? Nous allons revoir les points essentiels sur lesquels nous avons mis l'accent pour ce travail.

3.4.1 Une architecture flexible et extensible

Tout d'abord, cette architecture est flexible et extensible, car il est possible de rajouter des niveaux dans la hiérarchie intra- comme inter-nœuds. De plus, cette architecture est

intégrable facilement au sein de systèmes de type inter-opérables, que ce soit avec MPI ou d'autres outils. Enfin, l'ajout de nouveaux protocoles réseaux est simple et ne nécessite pas de devoir modifier en profondeur la structure même du logiciel.

3.4.2 Des communications potentiellement efficaces sur réseaux homogènes et hétérogènes

Au niveau des communications, cette architecture devrait permettre d'obtenir des performances homogènes, grâce sa gestion de la progression des communications reposant sur son moteur. Le niveau des performances en environnements homogènes ou hétérogènes devraient donc être similaires. De plus, les décisions de scrutation sont centralisées au sein d'une autorité commune même quand les protocoles ne sont pas au courant de l'existence des autres acteurs de communication.

L'utilisation des processus légers doit permettre un recouvrement des communications par des calculs ainsi qu'une réactivité accrue de l'application et la mise en place d'appels non-bloquants véritablement asynchrones. Les processus de niveau utilisateur sont *a priori* peu intrusifs pour le déroulement du calcul et les changements de contexte sont peu coûteux.

Enfin, le système est symétrique en ce qui concerne les niveaux de hiérarchie et les configurations supportées.

3.4.3 Des applications fusionnables et séparables aisément

Au niveau des applications, cette architecture possède un support pour les applications dynamiques. Aucun processus-démon n'est employé pour mettre en place ce service, qui utilise comme élément de base la session plutôt que le processus applicatif. Cette gestion de la dynamicité est peu coûteuse grâce aux processus légers.

*

Chapitre 4

Réalisation : MPICH-Madeleine, une implémentation multi-grappes, multi-réseaux et multi-sessions du standard MPI

Nous abordons maintenant la partie pratique de notre contribution puisqu'il s'agit de la mise en œuvre de l'architecture décrite dans le chapitre précédent. Nous allons dans un premier temps décrire les logiciels sur lesquels notre travail est fondé car, ainsi que nous l'avons évoqué précédemment, un travail *ex-nihilo* était trop considérable. De plus, le fait de devoir utiliser une bibliothèque de processus légers nous impose de nous pencher sur les outils existants afin de choisir le plus adéquat.

L'organisation de ce chapitre est donc la suivante : nous allons décrire une implémentation libre de MPI que nous avons modifiée pour réaliser notre travail : MPICH. Nous décrirons ensuite les outils indispensables à la mise en œuvre, à savoir les bibliothèques de processus légers Marcel et de communication haute-performance Madeleine. Ces deux élements sont les piliers de l'environnement de programmation PM2.

Viennent ensuite les descriptions de l'implémentation des fonctionnalités recherchées, c'est-à-dire le support multi-protocole (hiérarchie et hétérogénéité) et multi-session (gestion dynamique des processus). Nous concluons ce chapitre en donnant des stratégies alternatives d'implémentation ainsi que quelques réflexions sur cette realisation.

4.1 Le substrat : le logiciel MPICH

Message Passing Interface Chameleon (MPICH) est une implémentation libre du standard MPI, réalisée par une équipe de recherche sise à ARGONNE NATIONAL LABORATORY. Il s'agit, avec LAM/MPI, de l'implémentation la plus populaire et de nombreuses réalisations existantes – y compris commerciales – de MPI dérivent de MPICH.

Pourquoi être repartis de MPICH ? Retravailler une implémentation existante est une tâche qui demande une bonne connaissance de ses objets et structures, surtout lorsque nous

nous proposons de mettre en place une architecture utilisant des processus légers. À ce titre, l'expérience d'autres développeurs est très utile. Notre directeur de mémoire de DEA, Loïc PRYLLI, ayant contribué au développement de MPI-BIP (cf. 2.1.2.2) a donc pu nous faire partager sa connaissance dans la mise au point de MPICH-Madeleine. L'étude de son travail nous a fourni les éléments nécessaires pour une mise en route correcte. Ce travail, qui a constitué en quelque sorte un prototype de la version actuelle, est décrit dans [Mer00].

4.1.1 Une architecture stratifiée

L'architecture de MPICH dont il est question ici est celle de la première version du logiciel, c'est-à-dire l'implémentation correspondant à la révision 1.2 du standard MPI. C'est donc cette version du logiciel qui nous a servi de base de travail, ce qui implique que nous avons obtenu une implémentation de MPI conforme à la norme 1.2 ([GLDA96]). Il existe par ailleurs une version de MPICH correspondant à MPI-2 (MPICH2), qui est encore en version *bêta*. Cependant, si l'architecture a été totalement repensée, des similitudes subsistent. Ce point est discuté plus bas (cf. 4.1.3).

MPICH est un logiciel dont l'organisation est stratifiée, comme le montre la figure 32. La couche la plus haute implémente des objets comme les groupes et les communicateurs ou encore les opérations collectives. Ces mécanismes reposent sur une couche appelée l'*Abstract Device Interface* (ou ADI dans le reste du document) qui est une couche de portabilité mettant en œuvre les communications point-à-point, la gestion des files de messages et des requêtes ainsi que la scrutation pour les protocoles sous-jacents. Ces protocoles voient leurs interfaces masquées par une légère couche de portabilité appelée la *Channel Interface*.

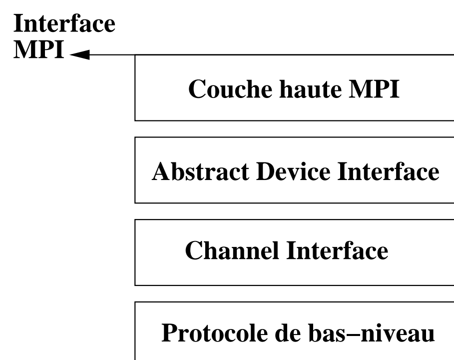


FIG. 32 – Structure stratifiée de MPICH

Ainsi, le portage de MPICH au-dessus d'une nouvelle technologie réseau peut s'effectuer à deux niveaux distincts :

- soit à un niveau relativement bas, c'est-à-dire au niveau de la *Channel Interface*. Un tel portage est en général assez rapide car cette interface n'implémente quelques fonctions de communication point-à-point bloquantes et non-bloquantes. Dans un tel cas, l'ADI est réutilisée telle quelle, aucune modification n'est nécessaire. La simplicité de cette approche confère à MPICH sa popularité pour le développement de bibliothèques MPI destinées aux grappes de PC ;

– soit au niveau de l'ADI, auquel cas il est possible de modifier les mécanismes de scrutation ou la gestion des files de messages pour mieux les faire correspondre au matériel ciblé. C'est l'approche choisie par certains constructeurs pour réaliser leur implémentation de MPI à partir de MPICH [1].

Il est bien évident que la seconde approche est plus compliquée, car elle nécessite un travail conséquent d'analyse et de réécriture, mais elle permet en revanche de se débarrasser des mécanismes jugés inadéquats ou préjudiciables pour l'obtention de bonnes performances.

4.1.2 L'Abstract Device Interface (ADI)

Nous donnons dans cette partie un coup de projecteur sur l'Abstract Device Interface, et en particulier nous détaillons sa gestion des protocoles multiples.

4.1.2.1 Rôles et intérêts de l'ADI

L'ADI est donc la couche logicielle qui forme le cœur de l'implémentation MPICH. Elle met en œuvre nombre d'objets et de services indispensables à un fonctionnement performant des communications. Son rôle est de fournir à la couche supérieure les fonctionnalités de base (typiquement, les communications point-à-point).

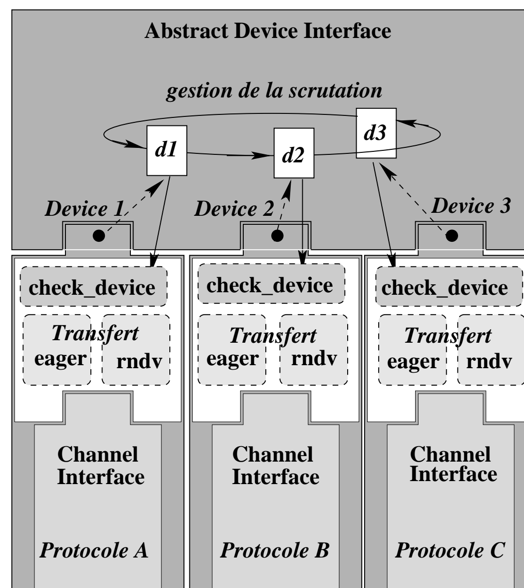


FIG. 33 – Organisation de l'ADI

L'ADI a également été conçue pour gérer simultanément un ensemble de protocoles réseaux, qui sont manipulables par le biais d'instances d'objets appelés des modules (*ADI devices*). Ces modules ont la charge d'implémenter la *Channel Interface* ainsi que les différents modes possibles de transmission, comme le mode *eager* ou bien *rendez-vous*, qui sont

[1] c'est le cas de SGI en particulier qui a choisi une approche incrémentale en écrivant successivement la *Channel Interface*, puis l'ADI puis tout le logiciel.

là encore instanciés par des objets particuliers. C'est enfin dans ce module que se trouve implémentée la fonction chargée de vérifier l'arrivée des messages pour un protocole donné (fonction `check_device` (cf. la figure 33)).

L'implémentation de l'ADI ne fait pas appel à des processus légers, mais autorise néanmoins leur emploi dans une application. Le but recherché étant de permettre la mise en place d'appels non-bloquants qui soient véritablement asynchrones (typiquement, on veut que ce soit un processus léger qui fasse un appel à `MPI_Isend`). L'ADI a également connu une révision puisque la version disponible actuellement est la seconde ([LG96], [LG]).

4.1.2.2 Cohabitation des différents modules de communication

Ainsi que nous l'avons expliqué dans le paragraphe précédent, l'ADI est prévue pour faire face à la multiplicité des modules de communication (*devices*). La figure 33 montre son organisation dans un tel cas. Dans notre exemple, nous supposons que l'ADI supporte trois protocoles de communication différents : A, B et C.

L'ADI possède alors trois *devices*, correspondant à chacun de ces protocoles (*Device 1, 2, 3*). Dans chacun de ces *devices*, on trouve une implémentation des modes de transfert pour messages courts (mode *eager*) ou longs (mode *rendez-vous*). Ces modes de transfert sont mis en œuvre au-dessus de la *Channel Interface*, écrite spécifiquement pour chaque protocole. C'est également le cas pour la fonction effectuant la vérification que des données sont disponibles (`check_device`).

Dans l'ADI, la gestion de ces différents modules est relativement basique en ce qui concerne la scrutation des communications. L'ensemble des *devices* est organisé autour d'une liste, dont les éléments sont parcourus successivement lorsque l'ADI décide de faire progresser les communications. Chaque élement est en fait un pointeur sur une structure correspondant à un *device*, structure permettant l'accès à la fonction de progression des communications `check_device` :

```
device  = MPID_device_set->device_list;
while (device) {
  result = (*device->check_device)( device );
  if (result > 0) {
    *error_code = result;
    break;
    }
  device = device->next;
}
```

C'est donc une approche de type tourniquet (*round robin*) qui est employée. Ceci est peu étonnant car l'architecture de MPICH suit un schéma de type multi-modulaire.

Enfin, le nombre de ces modules est fixé à l'*installation* de MPICH. De plus, le choix d'utiliser tel module plutôt que tel autre est décidé à l'*initialisation* de l'application et se fait sur la base du rang du processus distant, si bien qu'il est impossible de changer de protocole de communication entre deux processus au cours du déroulement de l'application.

4.1.2.3 Constat quant à la gestion de la scrutation

Cette approche est nativement multi-protocole, mais curieusement, ceci n'a pas été exploité pour réaliser une version de MPICH destinée aux grappes de grappes ou même «simplement» multi-réseau. En revanche, c'est ce procédé qui a été employé pour les versions dérivées de MPICH et destinées aux grappes homogènes de machines multi-processeurs. De plus, le choix du module s'effectuant à l'initialisation, le routage est statique et à moins – une fois encore – de modifer l'ADI n'a aucune chance de devenir dynamique. Enfin, l'implémentation en l'état de l'ADI n'est guère compatible avec une mise en œuvre de MPICH à base de processus légers pour un moteur de progression. Une seule conclusion s'impose donc : pour le travail que nous voulons réaliser, ***l'ADI doit être repensée et modifiée.***

4.1.3 MPICH et l'évolution de MPI

MPICH a naturellement suivi l'évolution des spécifications du standard. MPICH2, dont le but est d'implémenter les fonctionnalités de MPI-2 a été totalement repensé, même si l'architecture conserve des similitudes avec celle de son prédécesseur : on trouve toujours une couche de haut-niveau s'appuyant sur une nouvelle ADI (ADI3).

Cette dernière met en place un nouvel élément, qui est en fait un moteur de progession des communications. Ce moteur est prévu dès l'origine pour une progression classique (i.e comme celle que l'on trouve dans l'ADI2, mais améliorée) ou bien indépendante du déroulement de l'application, comme dans notre architecture (cf. 3.2.1).

4.2 L'outil : la suite logicielle PM2

Nous introduisons dans cette section les outils que nous avons utilisés pour injecter notre architecture dans MPICH, obtenant ainsi une version dérivée répondant aux objectifs fixés. Nous allons successivement détailler les caractéristiques de ces outils pour justifier de leur choix.

Afin de mettre au point notre architecture, nous avons vu qu'une bibliothèque de processus légers était indispensable. De plus, notre but étant d'aboutir à une version de MPI à la fois très efficace et disponible pour un nombre important de plate-formes, nous avions besoin d'une bibliothèque de communication qui soit à la fois très performante et générique.

Ces choix sont importants car ils conditionnent non seulement le niveau des performances obtenues mais également le travail à effectuer pour disposer d'un support pour un panel varié de technologies réseaux.

Ainsi notre choix s'est porté sur l'environnement de programmation PM2, qui nous offrait les outils nous semblant les mieux appropriés pour une telle réalisation.

4.2.1 Évolutions de PM2

PM2 est originellement un environnement de programmation basé sur le paradigme des appels de procédures à distance ou *Remote Procedure Call* (RPC). La première version de PM2

a été réalisée par Raymond NAMYST dans le cadre de sa thèse de doctorat ([NM95]) en 1995 et son but était de permettre l'exécution parallèle d'applications fortement irrégulières en virtualisant l'architecture sous-jacente au moyen de processus légers migrables. L'environnement était donc fondé sur deux bibliothèques : d'une part la bibliothèque Marcel pour la partie processus légers et d'autre part PVM pour la partie communications.

Cet environnement a évolué car la partie communications était insuffisamment performante. PVM a donc été remplacée par une seconde bibliothèque de communication dédiée, Madeleine. L'introduction de Madeleine a entraîné une mutation car il devenait essentiel que les deux aspects (multithreading et communications) soient capables de coopérer efficacement.

Progressivement, PM2 est passé du statut d'environnement de programmation destiné à des utilisateurs finaux à celui de support d'exécution (*runtime system*) pour des couches logicielles de plus haut niveau comme des intergiciels (*middleware*), en particulier MPI ou bien CORBA. Il est cependant important de comprendre que s'il est toujours possible d'utiliser indépendamment l'une de l'autre les bibliothèques Marcel et Madeleine, leur comportement diffère quand elles sont employées conjointement. Ainsi, Madeleine est une bibliothèque de communication compatible avec les mécanismes du multithreading. Ce point est essentiel et fait écho à la remarque concernant la difficile intégration d'éléments logiciels distincts (cf. 2.1.2.1). Dans le cas de Marcel et Madeleine, cette intégration existe et est effective.

C'est cette version de PM2 (ou plutôt de Marcel + Madeleine) que nous avons employée pour implémenter notre logiciel. Une dernière remarque concernera les RPC : si ce paradigme est toujours disponible dans l'actuel PM2, nous ne l'utilisons pas pour notre mise en œuvre : en ce sens, il ne s'agit donc pas d'un portage de MPI au-dessus de PM2.

4.2.2 La bibliothèque de communication Madeleine

Nous décrivons maintenant la bibliothèque de communication Madeleine. Après un tour d'horizon de ses traits généraux, nous nous attardons sur son interface avant de donner un petit exemple concret d'utilisation. Nous faisons également le point sur les capacités de Madeleine en ce qui concerne la gestion des configurations de type «grappes de grappes». Madeleine est le cœur du travail de thèse d'Olivier AUMAGE ([Aum02b]).

4.2.2.1 Caractéristiques de Madeleine

Madeleine est une interface de communication portable destinée à l'exploitation des grappes de PC interconnectés par des réseaux hauts-débits. Comme il s'agit en fait du sous-système de communication de PM2, elle est donc optimisée pour une utilisation conjointe avec Marcel.

Madeleine possède des propriétés nous intéressant directement :

- Madeleine est *multithread*, aussi bien dans sa conception que dans son utilisation, ce qui signifie que non seulement elle utilise les processus légers pour mettre en place certaines fonctionnalités (par exemple le service de retransmission des messages) mais qu'elle est aussi réentrante, permettant à plusieurs processus légers d'effectuer de façon concurrente des opérations de communication ;

- Madeleine est *multi-paradigme* c'est-à-dire qu'elle permet l'emploi, pour chaque protocole supporté, de différentes méthodes de transfert (envoi de messages, écriture dans une zone de mémoire distante, DMA, etc.) ;
- Madeleine est *multi-protocole* car elle permet aux applications de gérer simultanément plusieurs protocoles réseaux différents.

Madeleine a été portée au-dessus de nombreux protocoles et interfaces de communication : TCP, UDP, VRP, BIP, GM, SBP, VIA, SISCI et même ... MPI ! Les performances affichées par Madeleine sont de tout premier plan, en particulier dans le domaine des réseaux rapides.

Enfin, Madeleine peut être considérée comme une suite logicielle plus que comme une bibliothèque de programmation, car elle utilise des outils annexes très pratiques, comme un analyseur de fichiers de configuration (cf. Figure 36) et un lanceur d'application (le logiciel Léonie).

4.2.2.2 Interface de Madeleine

L'interface de Madeleine est plutôt orientée vers le passage de messages (ce qui simplife la tâche lors d'un portage de MPI), avec un nombre restreint de fonctions. Nous allons les détailler ainsi que les différents modes de construction des messages.

Fonctions d'émission et de réception
L'interface de MADELEINE se restreint à 6 fonctions :

`mad_begin_packing`	Début de construction d'un nouveau message
`mad_begin_unpacking`	Acceptation d'un message entrant
`mad_end_packing`	Finalisation d'une construction de message
`mad_end_unpacking`	Finalisation de la réception d'un message
`mad_pack`	Empaquetage d'un bloc de données
`mad_unpack`	Dépaquetage d'un bloc de données

Toutes ces fonctions correspondent à des appels bloquants, ce qui peut sembler un problème à première vue car MPI possède tout un ensemble d'opérations de communication non-bloquantes. Cependant, comme nous utilisons également des processus légers, cette difficulté peut être résolue.

Modes d'empaquetage et de dépaquetage
MADELEINE permet à l'application de spécifier des contraintes quant à l'émission et la réception des données transmises. Par exemple, lors d'une opération d'empaquetage (`mad_pack`), il est possible d'imposer que les données soient immédiatement disponibles du côté récepteur lors de l'opération `mad_unpack` correspondante. Au contraire, on peut relâcher complètement cette contrainte de disponibilité pour permettre à MADELEINE d'optimiser au maximum le mode de transmission en fonction du réseau sous-jacent. L'expression de telles contraintes par l'application constitue véritablement le point clé permettant l'obtention de bonnes performances tout en utilisant une interface complètement générique. La liste des différentes contraintes supportées par MADELEINE en matière d'empaquetage/dépaquetage des données est la suivante du côté émetteur :

`send_SAFER` indique que MADELEINE doit empaqueter les données de telle manière que des modifications ultérieures de la zone mémoire correspondante ne puissent pas modifier le message. C'est en particulier nécessaire si la zone des données à émettre est réutilisée avant que le message ne soit envoyé ;

`send_LATER` indique que MADELEINE ne peut pas accéder aux données avant que la fonction `mad_end_packing` ne soit appelée. Ceci signifie que toute modification de ces données effectuée entre leur empaquetage et leur envoi entraîne une mise à jour du contenu du message ;

`send_CHEAPER` c'est le mode par défaut. Il autorise MADELEINE à faire de son mieux pour traiter les données le plus efficacement possible. La contrepartie est qu'aucune hypothèse ne peut être faite sur le moment où MADELEINE accède aux données. Par conséquent, celles-ci doivent rester inchangées jusqu'à la fin de l'opération d'envoi.

Du côté récepteur on a les modes suivants :

`receive_EXPRESS` force MADELEINE à garantir que les données correspondantes sont *immédiatement* disponibles après l'opération de dépaquetage `mad_unpack`. Typiquement, ce mode est obligatoire si les données sont nécessaires pour procéder aux opérations de dépaquetage suivantes. Avec certains protocoles réseau cette fonctionnalité n'est pas très coûteuse alors qu'avec d'autres, cela peut avoir des répercussions sensibles, tant sur le plan de la latence que du débit.

`receive_CHEAPER` autorise MADELEINE à différer l'extraction des données correspondantes jusqu'à l'exécution de la fonction `mad_end_unpacking`. Par conséquent, aucune hypothèse ne peut être émise sur le moment exact où les données seront extraites. MADELEINE fait de son mieux pour minimiser le temps de transmission des messages (cela dépend du réseau sous-jacent).

4.2.2.3 Exemple pratique d'utilisation de Madeleine

La figure 34 donne un exemple d'utilisation de l'interface de MADELEINE. Nous souhaitons envoyer un tampon d'octets dont la taille est inconnue du processus récepteur. Celui-ci doit donc extraire la taille du tampon (un entier) avant d'extraire le tampon lui-même, parce que la zone de destination doit être allouée dynamiquement. Dans cet exemple, la contrainte est que l'entier indiquant la taille doit être extrait en mode `EXPRESS` *avant* de pouvoir dépaqueter les données correspondantes. Ces dernières peuvent quant à elles être extraites en mode `CHEAPER`, pour plus d'efficacité.

4.2.2.4 Support des configurations hétérogènes et des grappes de grappes

Idées et concepts principaux Madeleine possède également des fonctionnalités destinées à l'exploitation des grappes de grappes ([Aum02a], [ABEN02]) en plus des caractéristiques multi-protocole de Madeleine (cf. 4.2.2.1).

Pour ce faire, Madeleine introduit la notion de *réseau virtuel hétérogène*. Lorsque des grappes distinctes sont interconnectées, Madeleine utilise les réseaux locaux d'interconnexion pour en construire un plus large, englobant l'ensemble de ces grappes. Grâce à ce

Côté émetteur	Côté récepteur

`conn = mad_begin_packing(...);`
`mad_pack(conn,&size,sizeof(int),`
` send_CHEAPER,receive_EXPRESS);`
`mad_pack(conn, array, size,`
` send_CHEAPER,receive_CHEAPER);`
`mad_end_packing(conn);`
	`conn = mad_begin_unpacking(...);`
	`mad_unpack(conn,&size,sizeof(int),`
	` send_CHEAPER,receive_EXPRESS);`
	`array = malloc(size);`
	`mad_unpack(conn, array, size,`
	` send_CHEAPER,receive_CHEAPER);`
	`mad_end_unpacking(conn);`

FIG. 34 – Envoi de message avec MADELEINE.

support multi-protocole, Madeleine autorise l'utilisation des réseaux rapides pour les communications intra-grappes.

Du côté des communications inter-grappes, Madeleine est très souple : les schémas de communication peuvent être soit directs, soit avec des retransmissions, car un service de passerelle est disponible. Ces passerelles logicielles permettent à un nœud équipé de plusieurs technologies réseaux distinctes de procéder à une retransmission des messages émanant d'un processus appartenant à une première grappe et destinés à un processus membre d'une autre grappe. Ce mécanisme permet donc de ne pas recourir uniquement à TCP dans le cas des communications inter-grappes et les liens rapides sont alors pleinement exploitables. Ces passerelles retransmettent non seulement les messages mais jouent également le rôle de routeurs. On peut donc cascader de tels routeurs, ce qui présente l'intérêt de pouvoir exploiter des grappes dont l'interconnexion ne forme pas un graphe complet (i.e à la différence de PACX-MPI). Ces routeurs sont implémentés à l'aide de processus légers, ce qui permet de mettre en place des pipe-lines logiciels.

Ces mécanismes sont totalement transparents pour l'application, lui donnant l'impression de communiquer sur un unique réseau global, qu'il est alors possible de qualifier de *réseau virtuel hétérogène*. La création d'un tel réseau virtuel ainsi que la manipulation des différents protocoles se fait par l'intermédiaire d'objets appelés des *canaux*. Madeleine distingue au niveau applicatif deux types de canaux : les canaux physiques et les canaux virtuels.

- les canaux physiques instancient un protocole de communication utilisé pour l'exploitation d'un réseau physique. Par exemple, si une machine dispose d'une carte Myrinet, il sera possible de créer plusieurs canaux physiques correspondants à cette technologie. Ce mécanisme peut être considéré comme une forme de multiplexage ;
- les canaux virtuels sont eux construits à partir de canaux physiques uniquement (on ne peut donc pas mettre en place un canal virtuel au-dessus d'un autre canal virtuel). Cette construction se base sur des fichiers de configuration écrits par l'utilisateur, et s'effectue au démarrage de l'application. C'est lors de cette phase que sont déterminées les passerelles entre les différents canaux physiques (i.e les différentes instances des protocoles réseaux) et ce choix n'est pas remis en cause ultérieurement (le routage est donc statique). Les canaux physiques qui servent de base à un canal virtuel sont «absorbés» par ce dernier est deviennent par conséquent invisibles pour l'application.

Un exemple concret d'utilisation Afin de mieux appréhender comment s'utilise concrètement ces fonctionnalités, nous allons donner un petit exemple. Supposons que l'on veuille

interconnecter deux grappes de trois nœuds chacune. La grappe A (avec les nœuds `G1`, `G2` et `G3`) possède un réseau rapide de type Myrinet (avec le protocole GM). Quant à la grappe B, `H1`, `H2` et `H3` possèdent un réseau rapide de type SCI (avec le protocole SISCI). Nous allons également supposer que le nœud G3 dispose aussi d'une carte SCI et que TCP/Ethernet est disponible sur l'ensemble des six nœuds (cf. la Figure 35).

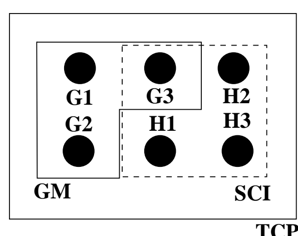


FIG. 35 – Exemple de grappe de grappes hétérogène

L'utilisateur commence par écrire un fichier décrivant les technologies disponibles sur les différents nœuds puis il fournit ensuite un fichier de description des canaux (cf. la Figure 36). Ce sont ces deux fichiers qui vont être utilisés par l'application afin de construire ses

Fichier de réseaux

```
networks : ({
   name  : ethernet;
   hosts : ( G1,G2,G3,H1,H2,H3);
   dev   : tcp;
 },{
   name  : myrinet;
   hosts : (G1,G2,G3);
   dev   : gm;
 },{
   name  : sci;
   hosts : (H1,H2,H3,G3);
   dev   : sisci;
});
```

Fichier de canaux

```
channels : ({
   name  : tcp_channel;
   net   : ethernet;
   hosts : ( G1,G2,G3,H1,H2,H3);
 },{
   name  : sci_channel;
   net   : sci;
   hosts : (H1,H2,H3,G3);
 },{
   name  : myri_channel;
   net   : myrinet;
   hosts : (G1,G2,G3);
});
vchannels : {
   name : global_channel;
   channels : ( myri_channel,
                sci_channel  );
};
```

FIG. 36 – Fichiers de configuration

canaux de communication. On voit que dans notre exemple, nous construisons trois canaux physiques correspondant aux réseaux sous-jacents. Ensuite nous créons un canal virtuel au-dessus des deux canaux physiques des réseaux rapides. Au niveau applicatif, nous allons donc pouvoir utiliser deux canaux : soit le canal physique correspondant à TCP, soit le canal virtuel correspondant au réseau virtuel hétérogène Myrinet/SCI.

Dans le cas d'une communication entre un nœud de la grappe A et un nœud de la grappe B, si nous choisissons d'opter pour le canal virtuel, alors tous les messages vont d'abord transiter via le réseau Myrinet pour être acheminés vers le nœud G3, qui joue le rôle de

passerelle. G3 retransmet ensuite le message vers son véritable destinataire en utilisant cette fois-ci le réseau SCI. Dans un tel cas de figure, il est donc possible de se passer totalement de TCP.

4.2.3 La bibliothèque de processus légers Marcel

Après cette rapide description de la bibliothèque de communication Madeleine, vient le tour de l'autre pilier logiciel de PM2 : la bibliothèque de processus légers Marcel. Nous allons donc donner les caractéristiques de cette bibliothèque avant d'examiner plus précisément en quoi l'ordonnanceur de Marcel peut améliorer les performances de notre architecture. La thèse de Vincent DANJEAN est consacrée à Marcel ([Dan04]).

4.2.3.1 Caractéristiques de Marcel

Marcel est une bibliothèque de processus légers de niveau utilisateur : la conséquence est que les temps de création, de destruction et de changement de contexte entre les processus sont très courts en comparaison des processus légers de niveau noyau. Ceci est aussi vrai pour les opérations de synchronisation car elles ne font pas appel au noyau du système d'exploitation.

Marcel présente une interface orientée POSIX et son utilisation conjointe avec Madeleine est optimisée. Marcel possède également des fonctionnalités particulières, car son ordonnanceur est adaptable selon l'architecture : dans le cas d'une machine uniprocesseur, il s'agira d'un ordonnanceur en espace utilisateur uniquement tandis que pour une meilleure exploitation des machines multi-processeurs, Marcel utilisera un ordonnanceur hybride à deux niveaux. Un tel ordonnanceur couple les processus légers de niveau utilisateur à des processus légers de niveau noyau, reconnus par le système et susceptibles d'être affectés à des processeurs physiques différents. L'ordonnancement de Marcel est préemptif, les processus légers ne sont donc pas dans l'obligation de rendre explicitement la main.

Marcel implémente également un modèle des *Scheduler Activations* ([Dan00]), ce qui permet aux processus légers d'effectuer des appels-système bloquants sans pour autant immobiliser tout le processus UNIX auquel ils appartiennent. Enfin l'ordonnanceur de Marcel permet d'effectuer des traitements intéressants dans le cas des communications. Ce point est décrit dans la partie ci-dessous.

4.2.3.2 Influence de l'ordonnancement sur la réactivité

Nous avons vu précédemment qu'une approche basique de type tourniquet avait pour effet d'augmenter le temps de transfert moyen des différents réseaux. L'utilisation de processus légers permet de réduire cet inconvénient quand on affecte un tel processus pour réaliser les opérations de scrutation. Cependant, dans ce cas, le temps d'un changement de contexte de l'ordonnancement devient déterminant car il aura une influence directe sur les performances, surtout pour les messages de courte taille.

Marcel évite ces écueils de deux manières : d'une part en offrant des temps de changements de contexte très courts et en proposant un mécanisme original pour éviter qu'un processus léger devant effectuer des réceptions de messages soit ordonnancé alors qu'aucune

donnée n'est disponible ([DN03]).

Le principe est le suivant : un processus léger enregistre une fonction de rappel (un *callback*) qui sera exécutée lors des changements de contexte. Typiquement, le code de cette fonction consistera à vérifier l'arrivée ou non d'un message. Dans le cas où un message est effectivement présent, alors le processus léger sera ordonnancé et dans le cas contraire il restera en attente. Ainsi, dans le cas de multiples protocoles, la détection d'un message intervient plus rapidement.

Par exemple, supposons que trois processus légers soient responsables de la réception de messages pour trois protocoles réseaux distincts : P1, P2 et P3. Supposons également que P1 doit être ordonnancé avant P2 qui doit l'être lui-même avant P3 et qu'un message arrive uniquement pour P3. Dans le cas d'un ordonnanceur classique, P1 aura d'abord la main, vérifiera la présence ou non d'un message puis passera la main à P2 qui fera de même. C'est quand P3 sera ordonnancé que le message arrivé sera effectivement pris en compte. Dans le cas de Marcel, l'arrivée du message pour P3 sera détectée lors du changement de contexte entre P1 et le processus qui lui succède. Comme il s'agit d'un message destiné à P3, c'est donc ce dernier qui aura la main et non P2. Ainsi, ce système permet d'être plus réactif face aux événements d'entrées-sorties.

En quelque sorte, Marcel, par le biais de son ordonnanceur, pourra donc jouer le rôle d'*arbitre* entre les différents protocoles, rôle qui jusqu'ici était dévolu à l'ADI (cf. 4.1.2) dans le cas d'une gestion multi-modulaire. Donc les processus légers de *scrutation*, qui sont au centre du moteur de progression (cf. 3.2.1), vont se muer en processus légers de *réception* des messages. En pratique cela n'impliquera aucun changement au niveau de la mise en place du moteur, il suffira d'écrire les fonctions nécessaires de *callback*.

4.3 Mise en œuvre du support multi-protocoles

Cette section aborde le cœur même de la réalisation puisque nous avons vu qu'architecturalement, c'est la mise en place d'un support multi-protocole efficace qui est problématique. Nous allons donc décrire dans cette section les éléments logiciels que nous avons mis en place pour arriver à une conformité au cahier des charges c'est-à-dire obtenir une implémentation multi-réseaux et multi-grappes de MPI.

Tout d'abord, nous allons détailler l'organisation du logiciel puis aborder le problème de l'introduction du multithreading dans MPICH. Nous décrirons ensuite la façon dont nous avons implémenté le moteur de progression qui a servi de base à la réalisation des deux sous-modules de communication, à savoir le module gérant les communications par mémoire partagée et celui gérant les communications réseaux.

4.3.1 Vue d'ensemble de la réalisation

MPICH-Madeleine est donc une implémentation dérivant de MPICH, héritant sa structure logicielle stratifiée que la figure 37 permet d'appréhender dans sa globalité. Nous retrouvons la couche haute de MPICH, celle qui met en place notamment les opérations collectives. Vient ensuite une ADI, qui en accord avec l'architecture décrite en 3.2.1, repose sur un unique module de communication (*ADI device*). Ce module exporte au niveau de l'ADI

les interfaces des modes de transfert *eager* et *rendez-vous* et repose sur trois sous-modules :

- un sous-module responsable des communications intra-nœuds, utilisant de la mémoire partagée (cf. ci-dessous 4.3.4). Ce module est fondé sur une bibliothèque minimale que nous avons réalisée ;
- un sous-module responsable des autres communications (i.e intra- comme inter-grappes) qui est fondé quant à lui sur la bibliothèque de communication Madeleine (cf. ci-dessous 4.3.3) ;
- un dernier sous-module gérant le cas particulier des communications intra-processus. Ce sous-module n'utilise que des recopies mémoire (`memcpy`). Les deux premiers de ces sous-modules n'utilisent pas qu'une bibliothèque de communication, ils implémentent également une partie du moteur de progression avec le concours de la bibliothèque de processus légers Marcel.

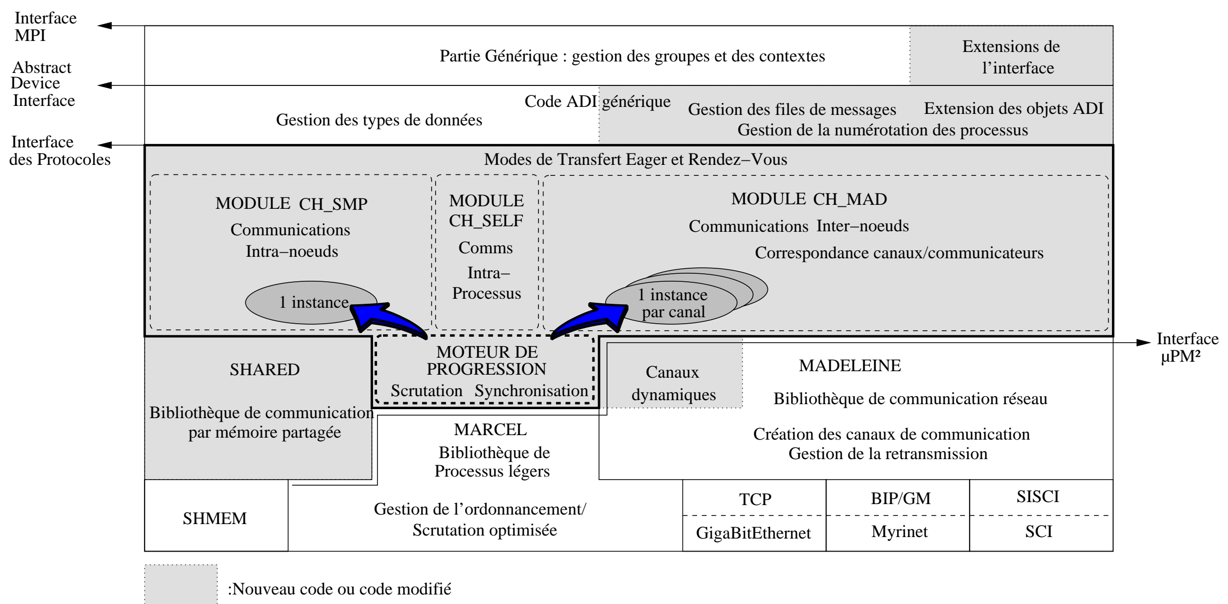


FIG. 37 – Couches logicielles de MPICH-Madeleine

Suivant le constat établi en 4.1.2, notre mise en place d'une architecture multithreadée nous a conduit à modifier une partie importante de l'ADI ainsi que la couche haute de MPICH et pas exclusivement à des fins d'extension de l'interface. Ceci est schématisé sur la figure 37, qui délimite clairement les parties du logiciel que nous avons introduites ou réécrites.

De plus, nous rappelons que si tout repose sur la suite logicielle PM2, le mécanisme des appels de procédures à distance (*Remote Procedure Call*) n'est pas du tout employé pour réaliser MPICH-Madeleine.

Enfin, d'un point de vue opérationnel, nous pouvons dresser les constats suivants :

1. au niveau de l'architecture, les différents protocoles de communication sont accédés par le biais des communicateurs MPI.
2. au niveau de Madeleine, les canaux de communication Madeleine sont les objets de base manipulés par l'application pour accéder aux différents protocoles.

Pour la réalisation, les choses sont limpides : nous allons bien évidemment affecter à chaque communicateur un canal Madeleine correspondant à un protocole de communication. On voit donc que les canaux Madeleine constituent naturellement[2] les bons relais de mise en place des mécanismes de gestion de la topologie.

4.3.2 MPICH et Le multithreading

Nous avons déjà evoqué le fait que les processus légers pouvaient être utilisés dans une perspective d'optimisation des communications intra-nœuds dans le cas de machines multi-processeurs (cf. 2.1.2.1). Cependant, ils sont également envisagés pour remédier à certains problèmes constatés au niveau de la conception des implémentations de MPI (e.g les appels asynchrones, cf. 5.3.3). et non au niveau du standard lui-même.

Mais l'introduction des processus légers dans une telle bibliothèque de communication est problématique pour le développement et les performances, comme nous l'avons plusieurs fois évoqué.

Nous allons examiner les modifications que nous avons apportées à MPICH et à l'ADI pour les rendre compatibles avec notre architecture. Nous détaillons plus particulièrement l'utilisation et le fonctionnement des processus légers dans MPICH-Madeleine.

4.3.2.1 Les améliorations du support du multithreading dans MPICH et l'ADI

L'introduction des processus légers a entraîné des modifications sur plusieurs aspects, mais nous mettons l'accent sur ceux que nous jugeons les plus importants après avoir rappelé le contexte initial.

Au départ, un support minimal Ainsi que vu dans le chapitre précedent (cf. 3.2.1), la réalisation d'un moteur de progression s'appuyant sur des processus légers nécessite la réécriture d'une partie non négligeable de l'ADI et des couches hautes de MPICH.

Il serait cependant exagéré (et faux) de dire que rien n'avait été fait à ce niveau dans MPICH : l'ADI avait bien été conçue pour une utilisation conjointe avec une bibliothèque de processus légers (typiquement, *pthread*). L'objectif était d'autoriser l'utilisation des processus légers au niveau applicatif en tant qu'outils de programmation. Il fallait donc que les sections critiques de l'ADI soient protégées afin d'assurer un fonctionnement intègre de MPICH.

Mais ce travail n'allait pas jusqu'à proposer une infrastructure souple et modifiable permettant la mise au point d'une ADI multithreadée et l'examen des quelques mécanismes mis en place originellement nous a convaincu qu'il s'agissait plus de pis-aller que d'autre chose. Qui plus est, l'implémentation de certains mécanismes[3] était grossièrement fausse.

Le défi était donc de mettre en place au sein de l'architecture existante nos propres mécanismes sans rentrer en conflit avec ceux pré-existants et que nous souhaitions conserver.

[2] dans Madeleine, les canaux sont assimilés à des communicateurs MPI

[3] la gestion des files de messages notamment . . .

Extension des mécanismes de synchronisation Nous avons d'abord procédé à la centralisation des mécanismes relatifs aux processus légers, comme les outils de synchronisation par exemple. De nouveaux ont été introduits pour les objets de type «requête» en réception (*rhandle*) ou en émission (*shandle*), par le biais d'une structure de synchronisation, ainsi que le montre la déclaration :

```
typedef struct {
  MPIR_OPTYPE   handle_type; /* type de handler    */
  volatile int  is_complete; /* état de la requête */

  /************************/
  /*    autres champs     */
  /************************/

  MPID_RNDV_t   recv_handle; /*    structure de    */
                             /*  synchronisation   */
} MPIR_COMMON;
```

Cette structure de synchronisation vient en fait compléter le champ `is_complete` qui servait à indiquer l'état de la requête en cours. Dans un contexte mono-thread, ceci est suffisant. Dans le contexte qui est le nôtre, non seulement ce champ doit être déclaré *volatile* mais en plus complété par la structure de synchronisation. Pour le type, nous avons la déclaration suivante :

```
typedef CH_MAD_SSYNCRO_t MPID_RNDV_t
```

Et la véritable structure est celle-ci :

```
typedef struct{
  MPID_HANDLE_MUTEX_DECLARE;            /* mutex pour l'accès global      */
                                        /* à la structure requête         */
  MPID_HANDLE_COND_DECLARE_4COMPLETE;   /* mutex     pour le is_complete  */
  MPID_HANDLE_MUTEX_DECLARE_4COMPLETE;  /* condition pour le is_complete  */

  volatile int push_set;                /* champ pour la fonction push    */
  MPID_HANDLE_COND_DECLARE_4PUSH ;      /* mutex     pour le push         */
  MPID_HANDLE_MUTEX_DECLARE_4PUSH;      /* condition pour le push         */

  volatile int status_set;              /* champ pour le champ status     */
  MPID_HANDLE_COND_DECLARE_4STATUS;     /* mutex     pour le status       */
  MPID_HANDLE_MUTEX_DECLARE_4STATUS;    /* condition pour le status       */

  MPID_HANDLE_COND_DECLARE_4CANCEL;     /* mutex     pour le is_cancelled */
  MPID_HANDLE_MUTEX_DECLARE_4CANCEL;    /* condition pour le is_cancelled */
} CH_MAD_SSYNCRO_t;
```

Nous avons donc un mutex pour gérer l'accès à la structure associée à la requête, ainsi que des mutex et des variables conditionnelles qui nous servent à gérer les champs volatiles indiquant l'état de la progression : `is_complete`, `push_set`, `status_set` et `is_cancelled`.

La multiplicité de ces variables est un choix d'implémentation car une telle approche est plus gourmande en ressources et complique la gestion des requêtes, mais une granularité plus fine assure une meilleure progression des communications. C'est l'introduction de ces mécanismes qui nous a obligé à revoir entièrement les fonctions MPI telles que

{Wait,Test}/{all,any,some} (cf. ci-dessous).

Enfin, les fonctions de test de l'ADI (`MPID_RecvIcomplete`, `MPID_RecvComplete`, `MPID_SendIcomplete`, `MPID_SendComplete`) ont également été totalement réécrites et nous sommes passés d'un simple test de variables à des traitements plus perfectionnés et en accord avec nos mécanismes.

Réécriture des fonctions de progression dans la couche haute Au niveau de la couche haute de MPICH, nous avons réécrit les fonctions permettant de tester ou d'attendre la disponibilité d'un message. Concrètement, cela signifie que les fonctions suivantes :

- `MPI_Waitall`, `MPI_Waitany`, `MPI_Waitsome` ;
- `MPI_Testall`, `MPI_Testany`, `MPI_Testsome`.

ne sont plus celles d'origine. Il s'agit là de la différence notable entre le logiciel de départ et notre version pour la couche haute.

Ajout de l'interface MPI-2 pour les processus légers Nous avons enfin écrit les fonctions concernant les processus légers et incluses dans le standard MPI-2 (et non pas dans le *"MPI Journal of Developpment"*). Ces fonctions sont peu nombreuses et leur implémentation assez directe. Sont donc disponibles :

- `MPI_Init_thread`, qui initialise l'environnement dans le cas de l'utilisation de processus légers au niveau applicatif. À noter que dans le cas de MPICH-Madeleine, il n'est pas nécessaire de faire un tel appel pour que les processus légers soit utilisables ;
- `MPI_Query_thread`, qui permet de connaître le niveau de multithreading supporté. Dans notre cas, nous supportons le niveau le plus haut, c'est-à-dire `MPI_THREAD_MULTIPLE` ;
- `MPI_Is_thread_main`, qui permet de savoir si le processus léger appelant est le *principal*, c'est-à-dire celui qui créé les autres et surtout exécute le code applicatif.

4.3.2.2 Les différents types de processus légers dans MPICH-Madeleine

L'emploi de processus légers dans nos modules de communication (`ch_mad` et `ch_smp`) permet de mettre en place deux éléments : d'une part le moteur de progression des communications (pour la réception des messages) et d'autre part les envois de messages en mode non-bloquant. En effet, tous les appels dans Madeleine étant bloquants (cf. 4.2.2.2), il nous faut trouver un moyen d'obtenir des fonctions dont la sémantique sera correcte vis-à-vis de celle des spécifications du standard MPI. Nous distinguons deux types de processus légers qui cohabitent dans notre implémentation :

1 – Les processus légers permanents
Le nombre de tels processus légers demeure constant au cours de l'exécution d'une application. On trouve dans cette catégorie :

- le *processus léger principal*, qui exécute le calcul proprement dit. C'est le seul existant à l'initialisation et il lance les autres processus légers permanents qui exécutent le code

du moteur de progression des communications. Ce processus crée également des processus légers temporaires, dans le cas d'appels à des fonctions de communication non-bloquantes ;

– les *processus légers du moteur de progression des communications* responsables de la scrutation pour les différents protocoles sous-jacents (mémoire partagée, protocoles réseaux). Ces processus légers permanents créent à leur tour d'autres processus légers, temporaires, dans le cadre du mode de transfert *rendez-vous* ;

Il y a mis à part ces processus légers d'autres qui existent également, mais ils sont intégrés au fonctionnement de Madeleine, et par conséquent demeurent invisibles pour MPICH, l'ADI et le moteur de progression.

2 – Les processus légers temporaires
Le nombre de ces processus légers est variable au cours de l'exécution. Ils sont en général créés par les processus légers permanents. On distingue deux emplois pour ces processus :

– l'implémentation des fonctions d'émission non-bloquantes du standard (e.g `MPI_Isend`);
– l'envoi de messages d'acquitement qui ne peuvent être émis directement par le moteur car dans ce cas, cela pourrait conduire à des situations d'interblocage. Il est indispensable de déléguer ces envois de message de contrôle (`OK_TO_SEND`) ou de données (`RNDV_DATA`) du mode de transfert *rendez-vous* à des processus légers temporaires.

Tous ces processus légers temporaires sont détruits une fois reçu le message qu'ils ont la charge de transmettre.

4.3.2.3 Discussion

Il est possible de discuter du bien fondé d'une approche reposant sur l'emploi de processus légers dont la finalité réside dans l'envoi d'un unique message car la création et la destruction de processus légers peuvent être des opérations impactant les performances à la longue. Cependant, dans le cas de la bibliothèque Marcel, opérant au niveau utilisateur, elles sont dans les faits peu coûteuses.

Une autre technique aurait consisté à gérer un ensemble (*pool*) de tels processus légers. Nous avons décidé de ne pas mettre en place un tel mécanisme, car cela aurait compliqué la mise en œuvre pour un gain en performances *a priori* minime.

Enfin, nous employons un nombre important de mécanismes de synchronisation, mais dans ce cas encore, du fait d'une localisation au niveau utilisateur, ces synchronisations présentent un coût négligeable[4].

4.3.3 Support multi-réseaux et multi-grappes : le module `ch_mad`

Après avoir examiné la manière dont le multithreading a été introduit dans MPICH et l'ADI, nous abordons maintenant la façon dont nous avons mis en place les sous-modules de communications. Nous rappelons qu'ils sont au nombre de deux, avec un sous-module

[4] pour la version de Marcel uniprocesseur en tout cas . . .

destiné aux transferts par mémoire partagée et un second destiné aux transferts réseau. Ce dernier sous-module, `ch_mad`, fait l'objet de cette partie.

4.3.3.1 Rôle et fonctions de `ch_mad`

La première version de MPICH-Madeleine était basée uniquement sur ce module. Elle est décrite dans ([Mer00]), mais présentait des restrictions quant à l'emploi des processus légers et des canaux Madeleine.

Depuis, ce travail a été remanié en profondeur et l'actuel `ch_mad` peut-être vu comme une sorte d'instance du moteur de progression des communications. Son rôle est de procéder à la réception des messages transitant par les différents réseaux disponibles.

Ce module gère également les canaux Madeleine et est responsable de l'interfaçage de ces objets avec les communicateurs MPI. C'est aussi cette partie du logiciel qui s'occupe de la gestion de la numérotation des processus, aussi bien dans les cas statiques que dynamiques.

4.3.3.2 Implantation du moteur de progression dans `ch_mad`

Le moteur de progression des communications est constitué d'un ensemble de processus légers qui scrutent les arrivées de messages. Conceptuellement, il est assimilable à une pompe amorcée à l'initialisation et ne devant jamais se bloquer sauf lors de l'appel à une primitive de réception de messages de la bibliothèque de bas-niveau. Les processus légers temporaires, ainsi que le principal se synchronisent par rapport aux processus du moteur et non l'inverse.

Pratiquement, chaque canal Madeleine déclaré par l'utilisateur déclenche la création d'un processus léger auquel il est associé. C'est donc le même mécanisme qui est utilisé et ce quel que soit le protocole réseau sous-jacent.

Le point fort du moteur, outre ses performances (cf. le chapitre 5), est sa réutilisabilité. Le mécanisme mis en place pourrait être transferé dans une implémentation plus conventionnelle de MPI.

Dans la suite, nous examinons les conséquences de l'existence de ce moteur sur la structure des messages échangés dans MPICH-Madeleine, aussi bien pour les informations de contrôle que pour les données. Nous détaillons ensuite les différents modes de transfert disponibles, avec leur implémentation et les optimisations réalisées. Enfin, la gestion des files de messages est expliquée.

4.3.3.3 Structure des messages

Dans la version de départ de MPICH, la réception des messages n'était pas centralisée : en effet, l'acquisition des données dans le cas d'un transfert de type *rendez-vous* (cf. 4.3.3.5) n'était pas effectuée avec la réception des autres messages.

Notre premier travail a donc été de regrouper toutes les réceptions afin d'être sûrs que ces actions n'étaient entreprises que par le moteur de progression. De plus, certaines particularités de Madeleine ont guidé la structure des messages échangés.

Taxonomie des messages Il existe huit types différents de messages dans MPICH-Madeleine (cf. Figure 38) dont la structure nous est dictée à la fois par l'architecture et le mode de fonctionnement de Madeleine. Les messages sont classiquement formés de deux parties : un en-tête suivi éventuellement d'un corps. Les messages dans Madeleine étant construits de façon incrémentale, l'en-tête et le cas échéant, le corps, correspondent chacun à une opération d'empaquetage/dépaquetage. Notre souci est de minimiser le nombre de ces opérations, même si cela implique d'émettre des messages plus gros que ce qui est nécessaire. Dans le cas où le message ne contient pas de charge utile, nous ne procédons qu'à l'empaquetage de l'en-tête.

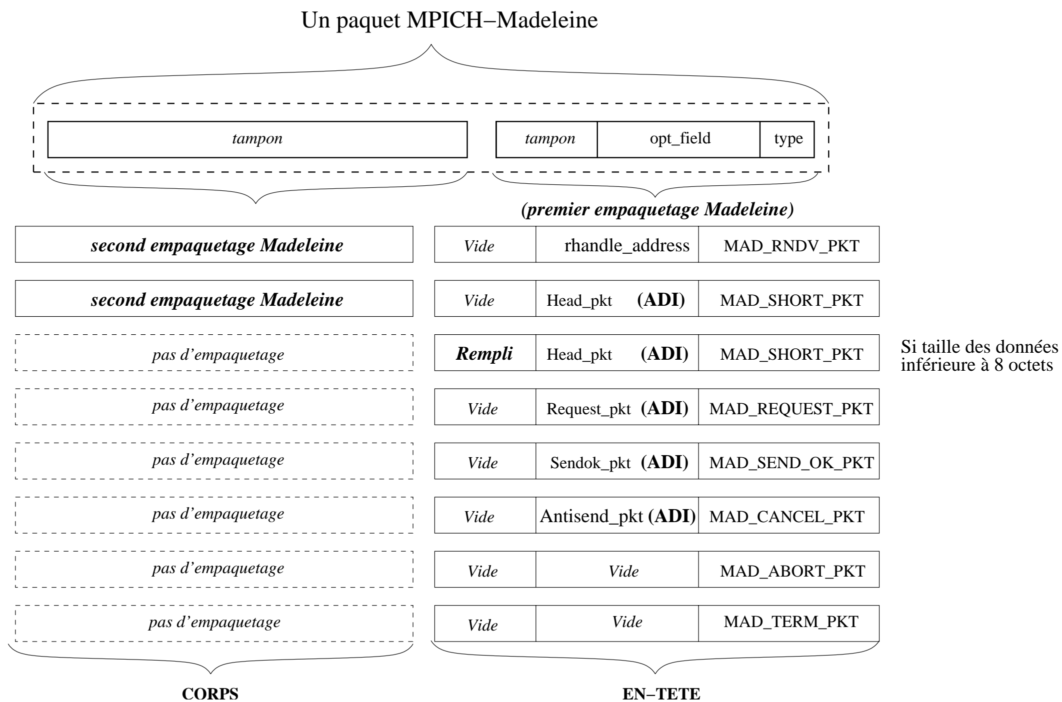


FIG. 38 – Structure des messages dans MPICH-Madeleine

Dans la version originale de l'ADI, la fonction de réception des messages ne recevait que des paquets ADI et l'acquisition des données dans le mode de transfert *rendez-vous* était effectuée à un moment autre que lors de la vérification de l'arrivée des messages. Comme nous venons de le dire, nous avons centralisé toutes les réceptions et avons gommé cette distinction entre messages de contrôle et messages de données.

Il est cependant toujours nécessaire de fournir l'information concernant le type de message qui est reçu, ce qui explique la présence du champ `type`. En fait, nous avons réutilisé les paquets pré-existants dans l'ADI et les avons encapsulés dans les nôtres, ce qui nous a permis d'introduire les types supplémentaires recherchés. Ensuite, les messages de madeleine n'étant pas auto-décrits et la réception s'effectuant sur un canal entier sans préciser la source (exactement comme une réception dans MPI avec le drapeau `MPI_ANY_SOURCE` positionné), nous devons obligatoirement préciser la taille des données que nous allons recevoir, ainsi que la source. Donc, notre en-tête devrait contenir un champ «Taille» et un champ «Source». Cependant, ils sont déjà présents dans les paquets ADI contenus dans nos propres

en-têtes.

Il y a donc une asymétrie congénitale : d'une part les messages ADI et d'autre part les messages nouvellement introduits, correspondant aux données dans le cas d'un transfert en mode *rendez-vous*, à une information d'abandon de l'exécution (*abort*), et à une information de terminaison. La figure 38 montre clairement ceci, car le champ `opt_field` de l'en-tête est soit vide soit contient un paquet ADI régulier ou une information de localisation de la requête dans le cas du mode de transfert *rendez-vous*. Cette adresse (`rhandle_address`) est donc copiée dans l'en-tête, en lieu et place d'un paquet ADI et le champ `opt_field` est vide dans le cas de paquets d'abandon ou de terminaison.

De plus, à des fins d'optimisation, nos en-têtes contiennent un petit tampon qui pourra être rempli dans le cas d'envoi de messages de très faible taille (en pratique, quelques octets).

Cette structure des messages a été établie dans l'optique d'une utilisation de Madeleine comme bibliothèque de bas-niveau, mais elle demeure néanmoins réutilisable avec d'autres outils (e.g cas des communications par mémoire partagée).

Implémentation En pratique, nous n'avons besoin de créer un type de donnée que pour l'en-tête :

```
typedef struct {
   CH_MAD_MESSAGE_TYPE  type;                    /* type de message      */
   CH_MAD_HEAD_FIELD_t  opt_field;               /* paquet ADI ou autre  */
   char                 padding[OPT_BUF_SIZE];   /* tampon pour les      */
} CH_MAD_HEADER_t;                               /* très petits messages */
```

avec

```
typedef union{
  MPID_Aint                rhandle_address;
  MPID_PKT_HEAD_T          head_pkt;
  MPID_PKT_REQUEST_SEND_T  request_pkt;
  MPID_PKT_OK_TO_SEND_T    sendok_pkt;
  MPID_PKT_ANTI_SEND_T     antisend_pkt;
} CH_MAD_HEAD_FIELD_t;
```

Mis à part le champ `rhandle_address`, tous les autres sont des paquets ADI que nous avons repris tels quels.

4.3.3.4 Le mode de transfert *eager*

Description du mode de transfert MPICH et l'ADI mettent en place des structures qui autorisent un changement dynamique de mode de transfert selon la taille du message. Cela permet d'utiliser des techniques différentes et potentiellement plus appropriées pour la taille considérée.

Nous avons choisi d'implémenter deux modes différents et le premier d'entre eux est le mode *eager*. Il s'agit d'une des techniques les plus répandues dans les bibliothèques de passage de messages. Le principe est simple : la communication est effectuée par l'envoi d'un unique message contenant l'intégralité des données à échanger entre les processus émetteur et récepteur (cf. Figure 39).

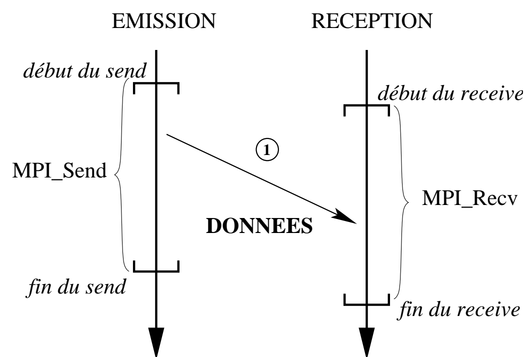


FIG. 39 – Principe du mode de transfert *eager*

Ce mode de transfert implique un certain nombre de copies additionnelles des données de l'utilisateur, aussi bien en émission qu'en réception, selon le système de communication bas-niveau employé. Cependant, l'avantage réside dans l'envoi d'un unique message, réservant cette technique pour les transmissions de messages de courte taille.

Implantation et optimisations Ce mode de transfert a été implémenté de plusieurs manières, afin de procéder à des optimisations selon la taille des données ou l'état de la requête associée au message (attendue ou inattendue).

1 – Cas des requêtes attendues : La transmission est effectuée avec un unique message Madeleine, émis en deux phases :

- un empaquetage (côté émetteur)/dépaquetage (côté récepteur) de l'en-tête du message, en mode `EXPRESS`, ce qui nous permet d'avoir accès aux informations indispensables pour vérifier les files de messages en réception, comme la taille des données, la source ou encore le *tag* (étape n°1 sur la figure 40) ;
- un empaquetage/dépaquetage du tampon utilisateur contenant les données à transmettre, en mode `CHEAPER`. Comme nous sommes dans le cas de messages *attendus*, c'est-à-dire qu'un appel à `MPI_Recv` à déjà été fait, nous connaissons l'adresse du tampon utilisateur en réception, ce qui nous évite une copie additionnelle (étape n°2 sur la figure 40).

Dans ce cas, nous n'effectuons aucune copie additionnelle au niveau de l'ADI ou de MPI (i.e *zéro-copie*, au sens habituel du terme). À noter que le tampon pour les très petits messages contenu dans l'en-tête n'est pas utilisé dans ce cas.

2 – Cas des requêtes inattendues : Dans ce cas encore, la transmission se fait avec un seul message, constitué d'une série de deux empaquetages/dépaquetages, comme dans le cas précédent (étapes n° 1 et 2 sur la figure 41). La différence est que l'adresse de réception étant inconnue au moment de la réception du message, nous devons impérativement conserver les données dans un tampon intermédiaire de réception. Quand l'adresse de réception est enfin connue (i.e quand l'appel à une fonction de réception est fait), nous pouvons procéder à la copie des données depuis ce tampon intermédiaire vers le tampon final utilisateur (étape n°3 sur la figure 41).Nous avons donc une

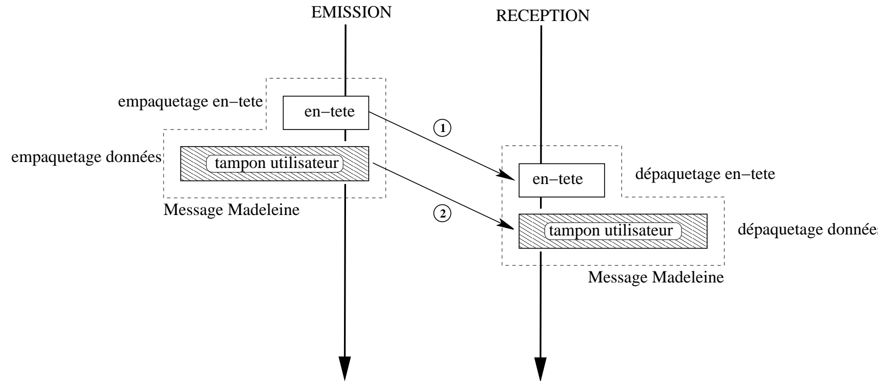


FIG. 40 – Implantation du mode de transfert *eager* dans le cas de messages attendus

seule copie additionnelle, du côté récepteur. Le tampon pour les très petits messages n'est pas utilisé non plus dans ce cas. Le tampon intermédiaire de réception n'est pas alloué dynamiquement car un appel à `malloc` pour chaque message inattendu ralentirait inutilement le moteur de progression. Pour contourner ce problème, chaque processus léger de réception possède une file de tampons pré-alloués, ce qui accélère le traitement au prix d'une augmentation de la consommation de mémoire.

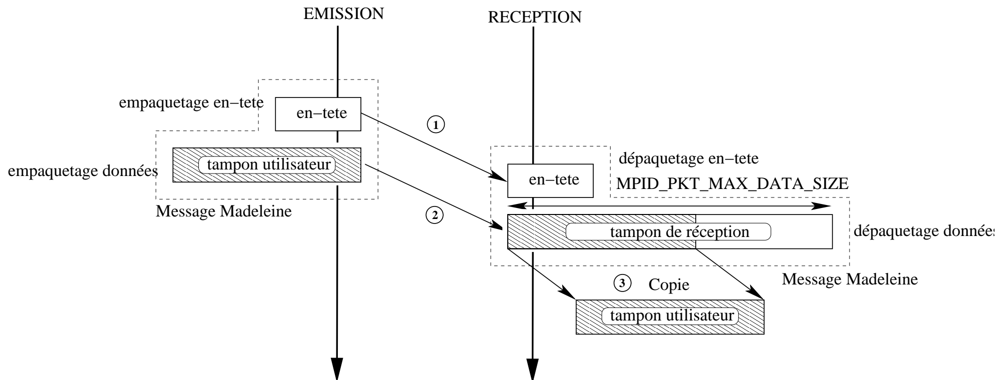


FIG. 41 – Implantation du mode de transfert *eager* dans le cas de messages inattendus

3 – Cas des messages de très petite taille : Ce cas ne concerne que les messages dont la taille est inférieure `OPT_BUF_SIZE` (en pratique, cette constante est fixée à 8 octets mais peut être modulée selon les besoins). Lorsqu'une telle situation se produit, la transmission du message est encore effectuée avec un seul message Madeleine (comme dans les cas précédents), mais avec un ***unique*** empaquetage/dépaquetage de l'en-tête, qui contient le tampon avec les données (étape n°2 sur la figure 42). Ce tampon doit être rempli à l'émission, ce qui implique une copie additionnelle du côté émetteur (étape n°1 sur la figure 42), et deux copies additionnelles du côté récepteur (étapes n°3 et 4 sur la figure 42). Nous avons donc au total trois copies intermédiaires, ce qui peut constituer à première vue un facteur de dégradation des performances, mais ce coût reste inférieur à celui d'un empaquetage Madeleine (nous sommes tout de même dans le cas de messages très petits, les opérations de recopie mémoire sont donc peu

coûteuses), ce qui nous permet d'obtenir de meilleurs résultats qu'avec l'implantation habituelle. Cette implantation entraîne une autre conséquence : la latence des messages de taille nulle se trouve légèrement augmentée (puisque l'en-tête est plus long de `OPT_BUF_SIZE` octets qu'il faut bien transmettre sur le réseau). Ceci a une incidence sur les performances du mode transfert *rendez-vous* (cf. 4.3.3.5).

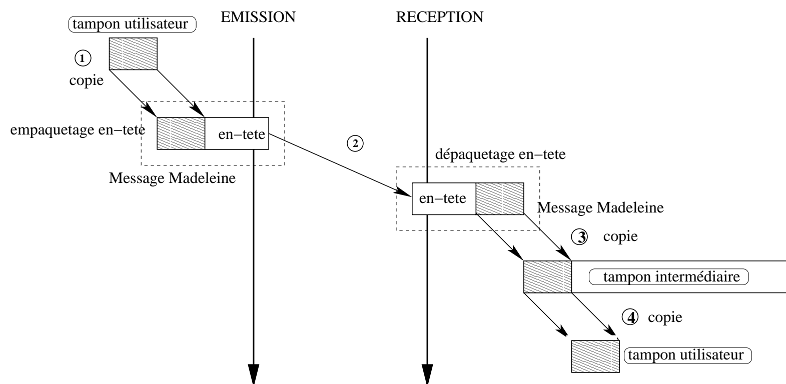


FIG. 42 – Implantation du mode de transfert *eager* dans le cas de messages très courts

4.3.3.5 Le mode de transfert *rendez-vous*

Description du mode de transfert Ce mode de transfert est le second que nous ayons mis en place dans notre implémentation. Il est également très répandu, et employé pour des messages dont la taille est importante. Contrairement au mode de transfert *eager*, il se déroule en trois étapes (cf. la figure 43) :

1. le processus émetteur envoie un message de requête (*Request To Send*) ;
2. dès que le processus récepteur connaît l'adresse de réception des données (avec un appel à `MPI_Recv` par exemple), il renvoie un message faisant office d'accusé-réception (*Clear To Send*) ;
3. à la réception de ce message, le processus émetteur peut transmettre les données.

On remarquera que l'adresse de réception étant connue, il est possible d'éviter toute recopie intermédiaire. Ce mode de transfert constitue donc un moyen aisé pour mettre en place des transferts de type *zéro-copie*.

Nous avons dit précédemment que l'implantation du mode de transfert *eager* pour les messages de très petite taille avait une conséquence sur les performances du mode de transfert *rendez-vous* : en effet, les messages de requête et d'accusé-réception sont en réalité des messages dont la taille des données est nulle (juste un en-tête). Augmenter la taille des en-têtes, c'est mécaniquement ralonger le temps total de transfert pour envoyer un message en mode *rendez-vous*. Il faut donc être vigilant à ne pas mettre trop de données dans les en-têtes sous peine de pénaliser les envois de messages avec un tel mode.

Implantation et optimisations Nous avons implémenté ce mode de transfert dans le cadre de notre moteur de progression d'une seule manière. Le déroulement s'effectue selon les

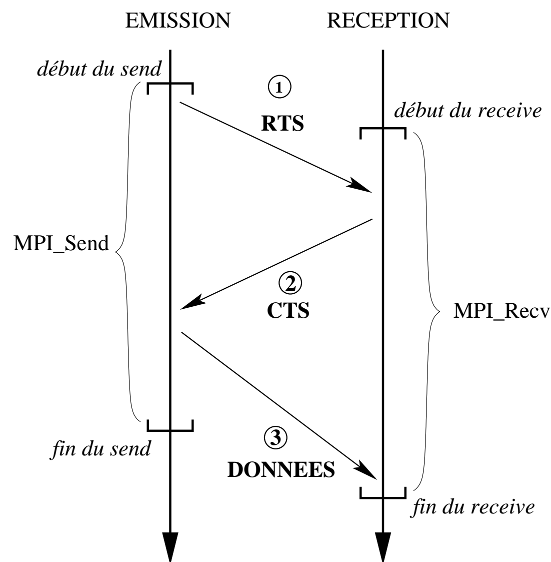


FIG. 43 – Principe du mode de transfert *rendez-vous*

étapes suivantes (cf. la figure 44) :

1. le processus émetteur envoie un message de requête au processus récepteur. Ce message est en fait constitué d'un unique empaquetage Madeleine, d'un en-tête, en mode `EXPRESS` ;
2. Le processus récepteur (i.e le processus léger du moteur de communication) reçoit ce message et attend de connaître l'adresse de réception des données pour envoyer un message d'accusé-réception au processus émetteur ;
3. ce message d'accusé-réception est également constitué d'un unique en-tête de message, contenant l'adresse de la structure de donnée gérant cette réception (*rhandle*) : c'est en fait le moyen d'identifier la communication en cours du côté du processur récepteur. Ce message est émis par un processus léger temporaire, créé par le moteur de progression ;
4. le processus émetteur reçoit ce message, créé un processus léger temporaire, qui va se charger du reste de la communication : il envoie un unique message qui sera constitué d'une série de deux empaquetages/dépaquetages (l'en-tête en mode `EXPRESS` suivi du tampon de données, en mode `CHEAPER`) ;
5. le processus émetteur empaquète l'en-tête qui contient l'adresse du *rhandle* ;
6. le processus récepteur dépaquète cet en-tête, ce qui permet au moteur de communication de savoir quelle requête doit être traitée pour procéder au dépaquetage des données dans le tampon utilisateur ;
7. la dernière séquence d'empaquetage/dépaquetage du tampon de données à lieu, en mode `CHEAPER`.

À noter qu'une optimisation est possible dans le cas de messages attendus : dès que la requête de réception est postée (e.g par un appel à `MPI_Recv`), il serait possible d'envoyer

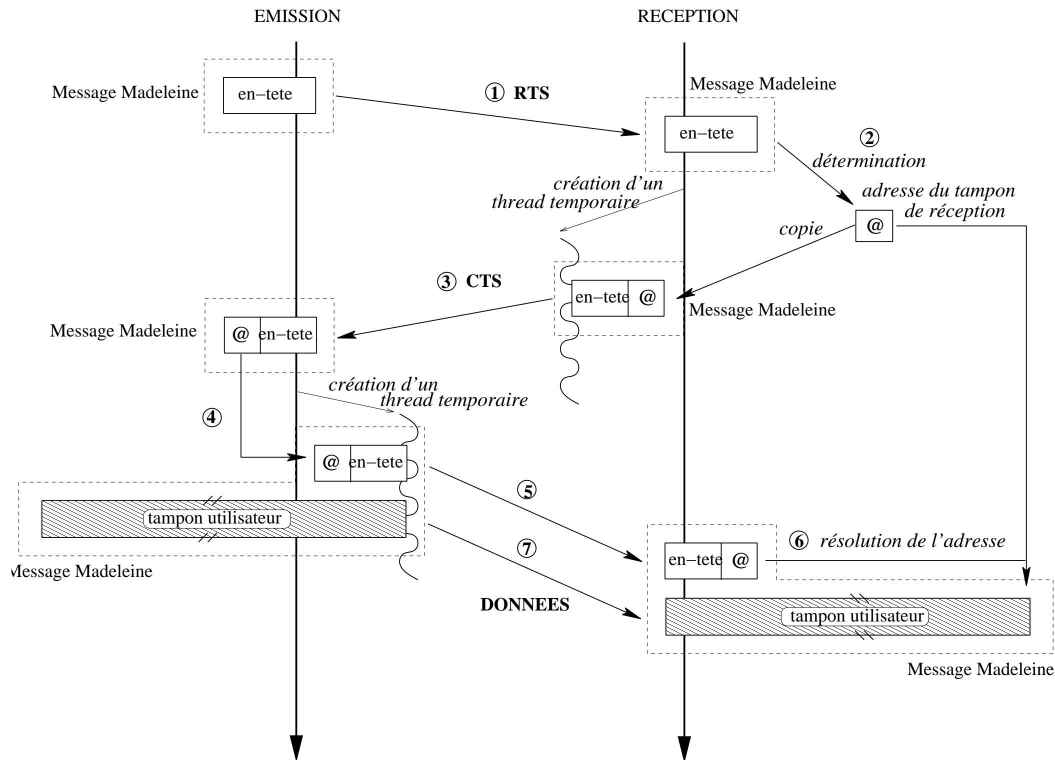


FIG. 44 – Implantation du mode de transfert *rendez-vous*

immédiatement l'accusé-réception, ce qui éviterait d'émettre un message de requête inutile. La latence en serait diminuée d'autant, ainsi que le traffic sur le réseau.

4.3.3.6 Gestion des files de messages

L'introduction du moteur de progression des communications nous a également donné l'opportunité de modifier la gestion des files de messages. Ces modifications ne sont pas obligatoires mais permettent d'éviter des points de contention au niveau de la réception, ce qui est toujours appréciable dans un contexte multithreadé.

Dans la version actuelle de MPICH, chaque processus possède un paire de files de messages, qui sert uniquement à la réception (il n'y pas de files pour les émissions). Cette paire est constituée :

- d'une première file pour conserver les messages *attendus*;
- d'une seconde file pour conserver les messages *inattendus*.

Ces deux files sont accessibles par le truchement d'une structure de données appelée l'*en-tête de queue* (*queue header*) qui contient un pointeur sur chaque premier élément des deux files, ainsi qu'un mutex. La structure correspondante est donc la suivante :

```
typedef struct {
    MPID_THREAD_DS_LOCK_DECLARE; /* macro pour le type mutex     */
    MPID_QUEUE unexpected;       /* file de requêtes inattendues */
    MPID_QUEUE posted;           /* file de requêtes   attendues */
} MPID_QHDR;
```

Le mutex est ici employé dans le cas où plusieurs processus légers appeleraient des primitives de réception simultanément. Donc, nous aurions pu conserver ce mécanisme tel quel, avec une paire globale de queues partagée par l'ensemble des canaux de communications.

Cependant, nous avons choisi de mettre en place une paire de files de messages pour *chaque* instance de protocole réseau. Cela entraîne une nouvelle fois une augmentation de la consommation mémoire, mais permet une véritable réception en parallèle sur plusieurs réseaux différents.

4.3.3.7 Interfaçage des canaux Madeleine et des communicateurs MPI

La partie la plus importante de ce module – exceptée la réception de données – est finalement l'interfaçage entre les canaux Madeleine et les communicateurs MPI. Rappelons que cet interfaçage a pour but de permettre l'exploitation de la topologie et des différents réseaux au niveau applicatif et que cette exploitation passe par l'utilisation des passerelles et des routeurs mis en place au niveau de Madeleine. Une fois que les nœuds devant jouer le rôle de passerelles ont été choisis par l'utilisateur avec les fichiers de configuration, Madeleine se charge de la création de processus légers qui vont retransmettre les messages d'un réseau vers un autre en utilisant des pipelines logiciels. Tout ceci est transparent pour MPI, mais est utilisé en pratique pour les configurations hétérogènes. Un interfaçage adéquat permet à l'utilisateur de manipuler ces mécanismes simplement.

Mise en place Comme indiqué en 4.3.1, les communicateurs MPI correspondent conceptuellement aux canaux de communication de Madeleine. À l'initialisation, si le nombre de ces canaux est supérieur à un, alors nous créons un ensemble de communicateurs à partir de `MPI_COMM_WORLD` avec l'opération `MPI_Comm_split` et nous attachons à chacun de ces communicateurs son canal.

Deux communicateurs peuvent bien entendu se voir affecter le même canal, mais un communicateur ne peut posséder qu'un unique canal de communication. Tout comme il existe dans MPI un communicateur particulier correspondant à l'ensemble des processsus (`MPI_COMM_WORLD`), nous avons obligatoirement besoin d'un canal avec une propriété similaire. Ceci n'est pas très contraignant, car cela veut juste signifier que les processus peuvent bien communiquer deux à deux, ce qui est une condition *sine qua none* dans le modèle de programmation de MPI. Le canal en question pourra être physique ou virtuel et sera affecté au communicateur `MPI_COMM_WORLD`. Il est appelé le *canal par défaut.*

Extension des structures de données de l'ADI Cet interfaçage nécessite une extension de la structure de donnée définie dans l'ADI et servant à la réalisation des communicateurs MPI. Nous avons rajouté un champ supplémentaire, en l'occurence la structure `channel_box`, comme le montre la définition suivante :

```
struct MPIR_COMMUNICATOR {
  /*************************/
  /*     autres champs      */
  /*************************/

  MPID_THREAD_DS_LOCK_DECLARE; /* mutex */
  CH_MAD_channel_box_t         *channel_box;
};
```

Cette structure centralise les attributs du canal qui sont souvent accédés au cours de l'application durant les phases de communication. Sa déclaration est :

```
typedef struct {
    p_mad_channel_t  channel;          /* pointeur sur le canal */
    MPID_QHDR       *channel_q_header; /* files de réception    */
    int              channel_thresh;   /* valeur du seuil       */
  }
CH_MAD_channel_box_t;
```

Outre le pointeur sur l'objet canal, indispensable pour les appels à `mad_begin_packing` ou `mad_begin_unpacking`, nous ajoutons l'en-tête permettant d'accéder aux files de réception et surtout nous trouvons la valeur du seuil qui est utilisé comme transition pour passer du mode de transfert *eager* au mode *rendez-vous*. En particulier, chaque protocole pourra ainsi avoir un seuil différent, correspondant mieux à ses propres mécanismes de transmission bas-niveau et optimisant ainsi les performances. L'avantage de l'extension des communicateurs est que les informations vitales sont accessibles directement et sans traitements complexes, ce qui permet un gain de temps.

4.3.3.8 Correspondance entre les numérotations des processus dans Madeleine et dans MPI

Le modèle de programmation de MPI spécifie que l'ensemble des numéros des processus doit former un intervalle contigu de 0 à n. Madeleine, en revanche, propose un modèle plus souple car la numérotation peut ne pas être contigüe et commencer par un numéro plus grand que zéro. Nous avons donc mis en place des mécanismes permettant de gérer ces différences afin que les numérotations soient bien consistantes. Précisons que ces correspondances sont valables aussi bien en environnements statiques que dynamiques.

Les rangs dans MPI
Dans MPI, un processus possède plusieurs numéros :

- un numéro de rang global. C'est en particulier le numéro qu'un processus obtient quand il demande son rang dans `MPI_COMM_WORLD` (i.e `MPI_Comm_rank( MPI_COMM_WORLD, &rank)`);
- des numéros de rang locaux. Ces numéros correspondent aux rangs que le processus occupe dans d'autres communicateurs.

Les rangs dans Madeleine
Madeleine opère une distinction quelque peu similaire, car un processus possède aussi plusieurs numéros :

- tout d'abord un processus possède un *rang global*, mais pour le *lanceur de programme*, et ce numéro est reporté au niveau de Madeleine tel quel ;
- ensuite chaque processus possède un rang local dans les canaux dont il est membre. Ces rangs sont pour leur part compris en 0 et k où k est la taille du canal considéré. Ces numéros sont de plus contigus.

Dans Madeleine, il n'y a donc pas de canal priviliégié pour la numérotation des processus.

Passage d'un monde à l'autre
Pour mettre en place la correspondance des rangs entre les univers Madeleine et MPI, il faut remarquer que l'attribution du rang global Madeleine en tant que rang global MPI *est une erreur*. En effet, dans le cas statique cela est tout-à-fait possible mais pose de gros problèmes dans le cas dynamique. Par exemple, s'il est acceptable de lancer une session Madeleine dont le processus de rang minimal possède un numéro strictement positif cette situation n'est pas possible dans MPI. Il faut donc trouver une autre correspondance, valide quelle que soit la situation.

Nous avons donc effectué les correspondances suivantes :

- le rang global d'un processus MPI est déterminé comme étant le rang de ce processus dans le *canal par défaut*. Ceci est consistant avec la correspondance canaux–communicateurs, car le canal par défaut est affecté (entre autres) à `MPI_COMM_WORLD` et les rangs locaux dans Madeleine sont contigus et commencent à zéro ;
- les rangs locaux dans un canal Madeleine sont les rangs dans le communicateur MPI auquel il est attaché. Le cas précédent ne constitue finalement qu'un cas particulier de ce point.

4.3.3.9 Extensions de l'interface MPI

Finalement, d'après tout ce qui précède, les ajouts à l'interface sont très restreints. Dans un cas de figure multi-grappes et multi-réseaux, l'utilisateur a besoin de connaître le nombre de technologies réseaux disponibles (qui peuvent coincider avec les partitions de la grappe de grappes) et il doit pouvoir être capable de déterminer le communicateur correspondant au canal désiré. D'où les extensions suivantes :

- le nombre de canaux dans l'application est accessible avec la constante :

  ```
  MPI_COMM_NUMBER
  ```

- les communicateurs créés à l'initialisation sont rangés dans une table :

  ```
  MPI_USER_COMM
  ```

  Comme `MPI_COMM_WORLD` est très souvent utilisé, son index est aussi accessible directement par le biais de la constante `MPI_COMM_WORLD_INDEX`, c'est à dire que `MPI_USER_COMM[MPI_COMM_WORLD_INDEX]` est équivalent à `MPI_COMM_WORLD`.

Ce tableau est utile dans le cas d'écriture de programmes bouclant sur l'ensemble des communicateurs;

– enfin, comme l'utilisateur spécifie les noms des canaux dans les fichiers de configuration (cf. Figure 36) utilisés pour lancer l'application, il est plus pratique d'accéder au canal par ce nom plutôt que par un numéro. Nous avons donc implémenté la fonction suivante :

```
comm = MPI_Comm MPI_GetCommFromName( char * name )
```

Cette fonction renvoie le communicateur auquel est attaché le canal dont le nom est `name`.

Ces extensions ne sont absolument pas obligatoires pour utiliser MPICH-Madeleine : tout programme MPI classique pourra être compilé et utiliser cette implémentation. Ces extensions n'ont pour but que de permettre une meilleure exploitation de la topologie et ne mettent pas en péril la portabilité des applications existantes. Cependant, ces extensions pourraient être intégrées en tant qu'attributs des communicateurs, ce qui aurait l'avantage de nous passer du rajout de fonctions dans le standard.

4.3.4 Support des communications par mémoire partagée : le module `ch_smp`

4.3.4.1 Rôle et fonctions de `ch_smp`

Nous décrivons maintenant la mise en œuvre du support pour les communications par mémoire partagée. Cette fonctionnalité étant absente dans Madeleine, nous l'avons mise en place au niveau de MPICH et de l'ADI. La mise à profit de l'expérience acquise lors de l'implémentation du module `ch_mad` nous a permis de créer un second module dédié aux communications intra-nœuds et utilisant également une architecture multithreadée (i.e le moteur et les structures de synchronisation sont similaires à celles de `ch_mad`).

Dans cette optique, nous avons commencé par créer une bibliothèque minimale de communication par mémoire partagée qui a ensuite été intégrée dans MPICH grâce au module `ch_smp`. Dans un premier temps, nous allons décrire cette bibliothèque, puis détailler son intégration dans MPICH et l'ADI.

4.3.4.2 Une bibliothèque basique de communication par mémoire partagée

Les fonctionnalités disponibles La bibliothèque que nous avons implémentée fournit un ensemble de fonctionnalités restreint mais permettant néanmoins une intégration aisée dans l'ADI. Nous trouvons outre les primitives d'initialisation (`shmem_init`) et de terminaison (`shmem_exit`) les primitives de transfert de données :

– en émission : `shmem_eager_send` et `shmem_rndv_send`;
– en réception : `shmem_eager_recv` et `shmem_rndv_recv`.

Nous avons donc tenu compte du fait que les modes de transfert *eager* et *rendez-vous* allaient être implantés au-dessus. Nous avons également mis en place un ensemble de fonctions permettant la synchronisation des différents processus MPI participant aux communications. Il a également fallu mettre sur pied des mécanismes permettant de synchroniser des processus légers appartenant à des processus MPI différents.

Nous avons de plus utilisé les processus légers pour réaliser des transferts pipelinés afin d'obtenir de meilleures performances. Ceci explique la disposition de la zone mémoire décrite ci-dessous.

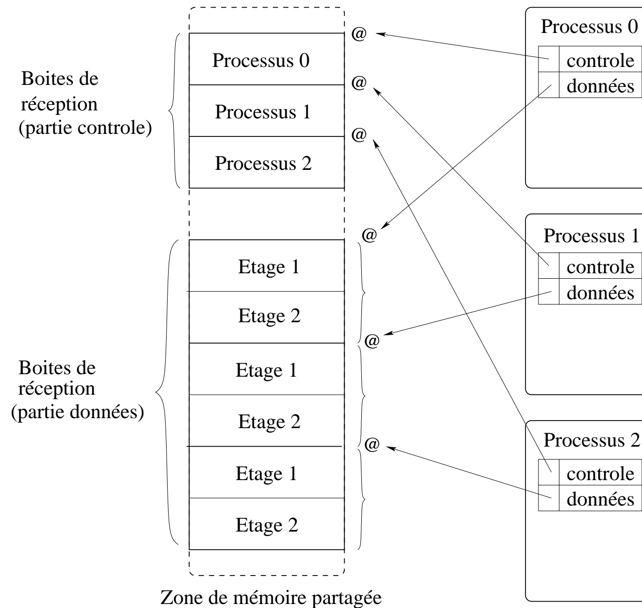


FIG. 45 – Organisation de la mémoire partagée (trois processus et un pipe-line à deux étages)

La disposition de la mémoire La figure 45 montre la façon dont nous utilisons la mémoire partagée. Deux zones sont employées : une zone servant à déposer des informations de contrôle (typiquement un en-tête de paquet) tandis qu'une seconde zone est destinée à recevoir les données proprement dites (le corps du message). Chaque zone est divisée selon le nombre de processus participants et selon la profondeur du pipeline (cas de la zone de données). La zone de contrôle contient en fait une structure dont le type est le suivant :

```
typedef struct {
   volatile int    smp_syncro;
   volatile int    head_ok;
   volatile int    read_ok[PIPE_DEPTH];
   CH_MAD_HEADER_t header;
} mem_map_t ;
```

Chaque processus possède donc deux adresses de référence, une adresse où il peut vérifier la disponibilité des informations de contrôle et une adresse où se trouvent les données. Les adresses de dépôt pour les processus émetteurs sont déduites en fonction de la taille des segments et du rang du processus destinataire du message.

En pratique, ces zones servent de “boîtes au lettres”, avec plusieurs processus écrivains par boîte mais un unique lecteur (le récepteur du message MPI). On remarquera que cette disposition est compatible avec la structure des messages du moteur de progression implémenté dans `ch_mad` (cf. ci-dessous).

4.3.4.3 Implantation du moteur de progression dans `ch_smp`

L'implantation du moteur de progression dans le cas de `ch_smp` n'est pas très éloignée de celle de `ch_mad`. Nous avons en effet réutilisé la même structure de messages en remplaçant les opérations d'empaquetage et de dépaquetage par des opérations de copies en mémoire. Dans le cas où un processus envoie un message avec une charge utile, on effectue donc une première copie pour l'en-tête et d'autres copies pour le corps si la taille des données dépasse celle de la zone mémoire (les transferts sont pipelinés ce qui implique d'augmenter le nombre de copies).

Ceci n'impacte pas beaucoup les performances car la taille d'un en-tête étant réduite, la première opération de copie n'est pas très sensible. De plus les pipelines permettent d'éviter que des processus légers attendent des données pendant que d'autres procèdent aux copies. Il est possible de recevoir simultanément un morceau du corps du message tandis qu'un autre morceau est en cours d'émission.

Nous avons de plus procédé à l'écriture des fonctions de rappel de Marcel (*callbacks*) qui sont exécutées par l'ordonnanceur à chaque changement de contexte. Dans notre cas, il s'agit de vérifier l'état des variables volatiles contenues dans la zone de contrôle.

En ce qui concerne les protocoles de transferts, nous avons implémenté les deux modes existants déjà dans `ch_mad` (*eager* et *rendez-vous*, donc). Mis à part le fait que les transferts de données sont pipelinés, il n'y a pas de différence notable par rapport à ce que nous avons expliqué en 4.3.3.4 et 4.3.3.5.

Comme dans le cas du moteur de `ch_mad`, un processus léger est responsable de la vérification de l'arrivée des messages. Cependant, à des fins d'optimisations il n'est créé que dans le cas où l'on détecterait effectivement que plusieurs processus MPI se partagent le même nœud multi-processeur.

Au niveau des files de messages, la mémoire partagée n'est pas considérée comme un canal de communication et par conséquent, le module `ch_smp` partage la paire de files du *canal par défaut*.

4.3.4.4 Intégration de `ch_smp` dans MPICH

L'intégration de ce nouveau module est effectuée au niveau de l'ADI et ne concerne que les phases d'initialisation et de terminaison. Concrètement, nous procédons à la détection des différents processus appartenant au même nœud et pour tous ces processus nous remplaçons le module `ch_mad` par `ch_smp`. La conséquence est donc que pour le sous-ensemble des processus localisés sur une même machine, il devient impossible de communiquer autrement que par la mémoire partagée.

Ceci a des conséquences pour la terminaison des programmes au niveau de l'arrêt de processus légers du moteur de `ch_mad` ainsi qu'au niveau de la gestion des files de messages. C'est ce qui nous empêche d'allouer une paire de files dédiées aux communications par mémoire partagée. En effet, l'opération `MPI_Cancel` n'utilisant pas de communicateur particulier, nous avons décidé d'employer le *canal par défaut* pour toutes les communications. Il faut donc que les files soient communes aussi bien dans le cas des communications réseaux que des autres.

Au final, l'intégration rapide de ce module nous a convaincu du potentiel de cette architecture adaptable à des bibliothèques de communication autres que Madeleine.

4.4 Mise en œuvre du support multi-sessions

Après avoir détaillé la mise en œuvre du support pour la hiérarchie et l'hétérogénéité, nous abordons maintenant l'autre aspect sur lequel nous avons concentré nos efforts, à savoir le support de la gestion dynamique des processus dans MPI. Notre objectif est de fournir là encore une interface simple mais des fonctionnalités riches.

En ce qui concerne la réalisation, nous avons décidé de ne pas implanter directement au niveau de MPICH et de l'ADI ce support et avons préféré étendre les fonctionnalités préexistantes de Madeleine pour arriver à remplir nos objectifs. En effet, si Madeleine proposait bien un support multi-grappes dont nous avons tiré parti de façon directe, elle ne disposait pas en revanche de fonctionnalités pour la gestion dynamique des processus. Les extensions nouvellement introduites bénéficient directement à notre implémentation de MPI, et plus largement à toutes les applications développées avec Madeleine.

Une fois les fonctionnalités recherchées mises en place au sein du support d'exécution, nous avons procédé à leur exportation vers les couches hautes de MPI et comme aucune fonction existante ne correspondait à ce que nous désirions, l'interface a donc été étendue.

Madeleine pouvant être considérée comme une suite logicielle plus que comme une «simple» bibliothèque de communication, le travail a été réalisé au niveau du lanceur d'applications et de la couche de communication proprement dite. Ces points sont détaillés ci-dessous ainsi que la description des nouvelles fonctionnalités dans MPI.

4.4.1 Le lanceur d'applications : Leony

Pour comprendre le travail que nous avons effectué, nous allons expliciter un point du fonctionnement de Madeleine que nous n'avons pas décrit jusqu'à présent : la phase de lancement des applications.

4.4.1.1 Les rôles du lanceur

Toute application développée avec Madeleine utilise un outil pour procéder à ce lancement : Léonie. Ce programme prend en arguments les fichiers de configuration décrits plus haut (cf. Figure 36) et transforme les données contenues dans ces fichiers en structures exploitables par Madeleine. C'est encore Léonie qui donne à un processus Madeleine son numéro de rang global. En particulier, notre propre commande de lancement de programmes MPI (`mpirun`) appelle Léonie de façon transparente pour l'utilisateur.

Ce programme n'est pas interactif et une fois l'application démarrée, il se contente d'attendre la fin des différents processus applicatifs. À ce titre, son rôle n'est pas négligeable puisque Léonie agit comme une entité de coordination et permet de synchroniser les processus lors de l'établissement ou de la terminaison des différents canaux de communication. Signalons enfin que Léonie communique avec les processus applicatifs via un ensemble de

sockets TCP.

4.4.1.2 Un lanceur de sessions

Principes Une seconde version de ce lanceur d'application a été développée afin de permettre les démarrages successifs de plusieurs applications indépendantes. Cette seconde version, appelée Leony est écrite en Python et est devenue interactive, ce qui manquait dans la première. Ainsi le lancement d'une application est presque identique au cas précédent, le seul changement étant qu'il faut désormais taper la commande de lancement (`add`) et préciser un nom pour l'application (i.e la session) que l'on souhaite démarrer. Ce changement de lanceur dans le cas d'applications statiques n'implique aucune modification des programmes écrits avec Madeleine : une application lancée avec l'ancienne version pourra l'être avec la nouvelle sans avoir à être recompilée. Il devient donc aisé de lancer plusieurs applications : il suffit de préciser à chaque fois les fichiers de configuration et le nom associé à la session pour commencer l'exécution. Pour la numérotation, Leony se charge toujours de l'attribution des rangs globaux des processus Madeleine mais désormais, lorsqu'une session se termine, les numéros des processus participants peuvent être réattribués à une nouvelle.

Prise en compte des informations de modification de session Cependant, si dans le cas d'applications statiques rien ne change, il n'en va pas de même dans le cas d'applications dynamiques. En effet, nous avons précédemment évoqué le fait que les événements de modification devaient être transmis aux processus applicatifs membres des sessions concernées (cf. 3.2.1.2). Cette prise en compte est effectuée dans Madeleine par le biais d'un processus léger dédié qui écoute la socket TCP de Leony et traite donc les commandes émises par cette dernière. Le lancement de ce processus léger n'est pas automatique et il faut donner un paramètre de plus à la commande `add` pour que ce mécanisme soit mis en place : `-dyn`. Le fait de ne pas donner ce paramètre implique que la session est statique.

4.4.1.3 La fusion de session vue par Leony

Nous allons voir maintenant comment la fusion est gérée dans Leony avant de détailler le processus.

La commande `merge` Nous avons introduit une nouvelle commande dans le jeu préexistant afin de pouvoir faire fusionner deux sessions : `merge`. Cette opération n'est pour l'instant possible que si elle est déclenchée par l'utilisateur qui doit donc explicitement indiquer à Leony quelles sessions il veut faire fusionner. Au niveau de la séparation des sessions ayant fusionné, cette intervention n'est pas nécessaire car on revient à la situation de départ et il suffit donc d'effectuer l'appel correspondant au niveau applicatif pour déclencher la séparation. La syntaxe de cette commande est très intuitive puisque il suffit de donner en arguments les noms des deux sessions que l'on souhaite faire fusionner. La seule restriction est que bien évidemment ces sessions soient dynamiques.

Principe de la fusion Nous allons voir comment cette fusion s'opère au niveau de Leony. Lorsqu'une session dynamique est lancée, toutes les autres sessions dynamiques déjà en cours d'exécution sont averties de sa présence. À ce moment, si une session désire s'étendre (avec un appel à la fonction Madeleine correspondante, cf. ci-dessous 4.4.2.2)) les processus restent bloqués jusqu'à ce que la seconde session effectue aussi cet appel et que la commande `merge` soit envoyée par Leony. Cette commande déclenche un recalcul de la configuration, et l'état de cette nouvelle situation est transmis à l'ensemble des processus membres des sessions d'origine pour qu'ils prennent connaissance des nouveaux venus. Les canaux sont réétablis et l'application reprend son cours habituel. Signalons que pour des raisons de performances et obtenir un redémarrage plus rapide, les connexions entre les différents processus membres du canal sont recréées en parallèle.

4.4.2 Implémentation dans Madeleine

Après avoir examiné les modifications apportées au lanceur de programmes, nous allons nous attarder sur celles introduites dans Madeleine. Nous avons déjà indiqué que les objets de base utilisés pour les communications étaient les canaux (cf. 4.2.2.4). Logiquement, ce sont ces même objets sur lesquels nous avons travaillé afin d'introduire les mécanismes voulus.

4.4.2.1 Les canaux extensibles dans Madeleine

Ainsi que spécifié en 4.2.2.4, Madeleine distingue des catégories différentes de canaux selon leurs propriétés. Nous n'avons évoqué jusqu'ici que les canaux physiques et les canaux virtuels, mais il existe par ailleurs des canaux de retransmission (utilisés dans les virtuels pour mettre en place les mécanismes de retransmission des messages entre les différents réseaux) ou encore les canaux de multiplexage. Nous avons introduit un nouveau type qui est le type *extensible*. Plus qu'un nouveau type, il s'agit en fait d'une propriété permettant de savoir quel canaux peuvent potentiellement fusionner avec d'autres.

Caractéristiques Cette propriété n'est toutefois pas applicable à l'ensemble des canaux disponibles. Nous avons décidé de restreindre l'accès à cette propriété. Deux cas sont possibles :

- soit les deux canaux *extensibles* correspondent au même protocole réseau sous-jacent (TCP, GM, SISCI, etc.) et dans ce cas, le canal résultant de la fusion sera donc un canal *physique* dont les éléments seront les processus membres des deux canaux initiaux ;
- soit les deux canaux correspondent à deux protocoles distincts et dans ce cas, le canal résultant est *virtuel* et englobant les deux canaux physiques de départ.

L'impossibilité actuelle dans Madeleine de créér des canaux virtuels à partir d'autres canaux virtuels entraîne que seuls des canaux physiques peuvent être déclarés *extensibles*.

Déclaration d'un canal extensible Concrètement, la déclaration d'un canal extensible se fait de la façon décrite ci-dessous :

```
channels : ({
   name  : tcp_channel;
   net   : ethernet;
   merge : yes;
   hosts : ( G1,G2,G3,H1,H2,H3);
 },{
   name  : sci_channel;
   net   : sci;
   hosts : (H1,H2,H3,G3);
 },{
   name  : myri_channel;
   net   : myrinet;
   hosts : (G1,G2,G3);
});
vchannels : {
   name     : global_channel;
   channels : ( myri_channel,
                sci_channel  );
};
```

Nous avons repris le même fichier d'exemple qu'en 4.2.2.4 mais cette fois-ci, le canal physique de communication correspondant au protocole TCP est déclaré ***extensible*** grâce à un nouvel attribut `merge` qui permet d'indiquer quel est le canal disposant de cette propriété pour cette session. En effet un unique canal peut être déclaré ***extensible*** dans une session donnée et en pratique il s'agira d'un canal couvrant l'ensemble des processus.

Conséquences sur le routage L'introduction de ces canaux extensibles a bien entendu des répercussions sur le routage dans le cas où l'on souhaite faire fusionner deux canaux physiques de différentes natures. En effet, la création du canal virtuel correspondant doit s'accompagner de la détermination d'une ou plusieurs passerelles pour effectuer les retransmissions d'un réseau vers l'autre. Ce choix dans le cas de canaux appartenant à la même session est possible en sélectionnant un processus membre des deux canaux s'exécutant sur une machine possédant les deux technologies d'interconnexion (passerelle). Cependant, dans le cas de la fusion, ce choix n'est pas possible car la passerelle sera désignée dans le meilleur des cas comme nœud pour deux processus MPI indépendants. Cette difficulté peut être contournée avec une extension des fichiers de configuration : l'utilisateur pourra désormais donner explicitement les routes entre les processus et ces définitions permettront d'avoir les routeurs désirés.

4.4.2.2 Extensions de l'interface de Madeleine

L'interface de Madeleine a été étendue pour offrir les fonctionnalités d'extension et de séparation des canaux. L'interface est basique mais a été néanmoins conçue dans le but d'une exploitation par des couches supérieures. Les deux principales fonctions introduites ont un prototype très similaire :

```
mad_expand_channel(p_mad_madeleine_t madeleine, char *name)
```

et

```
mad_shrink_channel(p_mad_madeleine_t madeleine, char *name)
```

Elle prennent en argument un pointeur sur l'objet canal considéré ainsi que son nom. Ce nom est celui fourni par l'utilisateur dans le fichier de configuration. Ces deux fonctions correspondent donc exactement aux fonctionnalités que nous voulions mettre en place dans MPI et permettent une exportation aisé vers cette interface. Ce point fait l'objet de la partie suivante.

4.4.3 Extension de l'interface MPI

Interface actuelle Les extensions apportées à l'interface MPI sont comme dans le cas du support multi-grappes résolument peu nombreuses et nous n'avons pour le moment implémenté qu'une unique fonction permettant à deux sessions MPI différentes – mais lancées toutes deux avec Leony – de fusionner pour n'en former plus qu'une. Son prototype est le suivant :

```
MPI_Comm_resize( MPI_Comm communicator, int operation)
```

et les opérations disponibles sont au nombre de deux :

- `MPI_COMM_OP_EXPAND` permet au communicateur de s'étendre ;
- `MPI_COMM_OP_SHRINK` permet au communicateur de revenir à l'*état d'origine*.

En pratique, comme nous cherchons à fusionner deux applications, cette opération sera réalisée avec `MPI_COMM_OP_EXPAND` qui est une opération collective devant être effectuée par les deux sessions. Ensuite un appel avec l'opération `MPI_COMM_OP_SHRINK` permet de revenir à la configuration de départ, c'est-à-dire celle d'avant le `MPI_COMM_OP_EXPAND`. Ces opérations rappellent celles existantes dans FT-MPI pour la tolérance aux pannes (cf. 2.2.3.2).

Nous avons également préféré étendre l'interface plutôt que de changer la sémantique de MPI, ainsi que le préconise [LG94].

L'appel à la fonction `MPI_Comm_resize` change non seulement la taille de la session mais aussi le rang des processus participants. Il est donc nécessaire de déterminer à nouveau ces paramètres, qui sont très souvent utilisés dans l'écriture des programmes. Le communicateur appelant reste un *intracommunicateur*, ce qui permet en particulier de pouvoir toujours effectuer des opérations de communications collectives sur l'ensemble des processus membres.

La seule restriction actuelle est que seuls des communicateurs dont la taille est égale à celle de la session peuvent appeler la fonction `MPI_Comm_resize`. Ceci n'est pas un problème car cela permet de conserver des configurations valides. En effet, il serait hasardeux d'intégrer de nouveaux processus au sein d'un communicateur couvrant partiellement la session : nous obtiendrions alors une session avec des tailles différentes selon les processus, ce qui n'est pas possible dans MPI. De plus, il est indispensable que toutes les communications en cours soient achevées avant de faire appel à cette fonction.

Enfin, on remarquera que nous ne faisons aucune supposition sur les applications : il est ainsi tout-à-fait possible de fusionner deux applications différentes (par exemple une

application de calcul et une autre de visualisation).

Conformité par rapport au standard Pour le moment, nous devons lancer explicitement les deux sessions MPI avec le lanceur de programme Leony. Il n'est donc pas encore possible de se passer de ce tiers. Une session existante ne possède pas la capacité de se répliquer sur d'autres machines, comme dans le cas de `MPI_Comm_spawn`.

Se passer de l'intervenant extérieur à MPI, cela revient à mettre en place des objets permettant la connexion entre deux processus comme des ports de communication. Si ces objets sont ajoutés dans Madeleine, nous pourrions les utiliser directement pour implémenter l'interface régulière de MPI-2 concernant la gestion dynamique des processus, c'est-à-dire les fonctions : `MPI_Open_port`, `MPI_Close_port`, `MPI_Comm_accept` et `MPI_Comm_connect`.

4.5 Stratégies alternatives d'implémentation

Nous discutons maintenant à propos d'autres stratégies possible pour la mise en œuvre des supports pour la hiérarchie et l'hétérogénéité. En ce qui concerne la hiérarchie, nous nous focalisons sur l'exploitation des machines multi-processeurs.

4.5.1 Support de la hiérarchie

Dans l'état de l'art, nous avons fait mention des approches de type hybride, utilisant deux outils distincts, par exemple MPI pour les communications inter-nœuds et une bibliothèque de processus légers pour les communications intra-nœuds qui s'appuyent sur l'espace d'adressage commun à tous les processus légers (cf. 2.1.2.1). Dans ce cas, un processus MPI est supporté de façon sous-jacente par un processus léger.

Il est à noter que nous bénéficions automatiquement des améliorations apportées à la bibliothèque Marcel, en particulier le fait que l'ordonnanceur des processus légers puisse être de niveau purement utilisateur, ou bien hybride, afin d'exploiter les machines multi-processeurs (cf. 4.2.3.1). Ceci entraîne une conséquence intéressante, car le support des grappes de telles machines peut être réalisé de deux manières :

- soit par le développement d'un module de communication dédié (cf. 2.1.2.2). Dans ce cas nous utilisons l'ordonnanceur Marcel classique (c'est-à-dire purement utilisateur) ;
- soit par l'utilisation de l'ordonnanceur hybride à deux niveaux. Dans ce cas, la présence d'au moins deux processus légers (le premier gérant les calculs, le second la progression des communications) garantissant l'utilisation de plusieurs processeurs simultanément. Il s'agit en fait d'une parallélisation des processus MPI réalisée de façon interne et totalement transparente pour l'application.

C'est la première approche qui a été choisie (cf. 4.3.4) car elle se fond mieux dans le cadre plus global du support multi-protocoles. La seconde approche ressemble beaucoup à la solution MPI + OpenMP, sauf que la parallélisation n'intervient pas au niveau du code applicatif, mais dans l'implémentation de MPI proprement dite. Cette approche n'a pas été testée à

l'heure actuelle mais mérite sans doute une évaluation pour apprécier le gain potentiel apporté par un processeur dédié à la progression des communications, même si dans ce cas les synchronisations sont plus coûteuses.

4.5.2 Support de l'hétérogénéité

Nous avons vu précédemment qu'il existait deux stratégies de portage dans MPICH : au niveau de l'ADI ou bien au niveau de la *Channel Interface* (cf. 4.1.1). De plus, nous avons fait état des capacités multi-protocoles de Madeleine (cf. 4.2.2.4). Il est alors légitime de se poser la question quant à la stratégie que nous avons adoptée : n'aurait-il pas été plus simple en fin de compte de travailler uniquement au niveau de la *Channel Interface* sans avoir à introduire du multithreading et par conséquent modifier l'ADI ?

Une telle approche, quasi-immédiate sur le principe possède l'avantage de la simplicité de la mise en œuvre. Elle se heurte cependant à deux écueils. D'une part, les appels de Madeleine étant bloquants, le recours à des processus légers pour la mise en place des appels non-bloquants devient nécessaire. D'autre part, cette approche aurait permis d'obtenir une implémentation de MPI capable d'exploiter «juste» des réseaux hétérogènes. En effet, ce n'est pas parce que la bibliothèque de communication sous-jacente est multi-protocole que l'implémentation de MPI le devient automatiquement ! C'est justement le problème des approches unimodulaires, dont nous cherchons à éviter le côté «boîte noire». ***La propriété multi-protocole vient de l'architecture, pas de Madeleine.*** À la limite, il eût été possible de se passer totalement de cette dernière et de n'utiliser que les protocoles bas-niveaux existants (BIP, GM, SISCI, TCP, etc.), un peu comme nous l'avons fait pour la mise en place des communications par mémoire partagée. Cependant, outre le fait que nous aurions dû alors avoir des mécanismes de communications réentrants (ce qui est le cas de Madeleine), les performances n'auraient peut-être pas été aussi bonnes qu'elles le sont actuellement. Nous aurions dû aussi concevoir un mécanisme permettant de traiter les fichiers de configuration de l'utilisateur ainsi qu'un système de lancement des applications. Ces outils existants dans Madeleine ont été repris à notre avantage. Enfin, nous aurions également dû mettre en place des mécanismes de routage et de retransmission au niveau de MPI, un peu comme ceux de MetaMPICH (cf. 2.1.5.2), sans compter qu'une partie non négligeable de notre travail concerne Madeleine (en particulier les canaux extensibles) dont le principe aurait dû lui aussi être réimplanté dans MPICH.

En conclusion, si le support de Madeleine nous permet une simplification du travail, grâce à son interface unifiée pour tout un ensemble de protocoles, ainsi qu'un accès à des services essentiels pour une approche grappe de grappes (performances, retransmission de messages, etc.), il est très important d'avoir bien ancré à l'esprit que c'est l'utilisation d'une bibliothèque de processus légers qui nous permet d'obtenir un système à la fois original et performant pour la gestion du multi-protocole.

4.6 Conclusion

Cette conclusion est l'occasion de dresser le bilan de ce qui a été accompli ainsi que d'engager quelques réflexions sur la manière dont MPICH-Madeleine a été mis en œuvre.

4.6.1 Bilan du travail réalisé

Nous pouvons faire la liste de l'ensemble des réalisations effectuées pour arriver au résultat. Nous avons ainsi conçu :

- l'architecture logicielle globale avec notamment l'introduction du moteur de progression de communications ;
- une version de l'ADI et de MPICH multithreadée ;
- le module de communication multi-réseau `ch_mad`, avec les protocoles de communications *eager* et *rendez-vous* et la gestion des communicateurs ;
- le module de communication `ch_smp` et la bibliothèque de communication par mémoire partagée sur laquelle il s'appuie ;
- l'introduction de nouvelles fonctionnalités dans l'interface MPI ;
- un travail dans Madeleine et Leony pour obtenir des canaux de communication dynamiques.

L'ensemble de ces éléments nous a permis d'arriver à une version opérationnelle de MPICH-Madeleine et conforme à notre cahier des charges du point de vue des fonctionnalités.

4.6.2 Réflexions sur la mise en œuvre

Au cours de ce chapitre, non n'avons pas manqué d'indiquer les alternatives possibles pour l'implémentation effective du logiciel. Ces alternatives nous ont permis d'alimenter notre discussion et de justifier nos choix. Nous n'allons pas ici revenir sur l'implémentation proprement dite mais il nous paraît important de donner un aperçu des points forts comme de ceux un peu plus faibles de notre logiciel.

4.6.2.1 Points forts du logiciel

Les points forts de MPICH-Madeleine sont nombreux, tant au niveau de l'architecture que de sa réalisation.

Tout d'abord, cette architecture est fondée sur un moteur de progression utilisant des processus légers qui est réutilisable aisément. En outre elle nous semble propice à la mise en place de fonctionnalités du standard MPI-2, comme les communications unilatérales (*one-sided communications*) par exemple.

Cette version de MPI s'adresse aux grappes hétérogènes comme homogènes et les nœuds peuvent être de nature uni- ou multi-processeurs. Nous ne négligeons pas non plus les configurations plus ambitieuses et notre travail peut être intégré au sein de systèmes plus complexes. Un aspect pratique est que les applications peuvent changer de réseau d'interconnection sans avoir à être recompilée puisqu'il suffit de changer les fichiers de configuration pour changer de réseau.

Notre approche de la gestion de la topologie est simple pour l'utilisateur mais elle demeure néanmoins flexible et riche car plusieurs schémas de communications inter-grappes sont possibles : avec ou sans retransmissions.

Enfin, le système n'est pas asymétrique, dans la mesure où nous ne faisons pas de différences entre les communications intra- et inter-grappes. Les performances globales seront donc plus homogènes.

4.6.2.2 Limitations actuelles de l'implémentation

Cependant, nous avons relevé des points concernant l'implémentation qui pourraient être améliorés.

Tout d'abord, alors que nous nous situons en contexte hétérogène, nous ne supportons que des architectures de processeurs homogènes. Du point de vue des outils utilisés, Madeleine étant déjà capable de fonctionner dans un tel environnement, les modifications n'interviendraient qu'au niveau des *devices* et de l'ADI. Cette introduction aurait fatalement un impact négatif sur les performances mais permettrait l'exploitation de nouvelles configurations. Il nous semble de plus qu'une politique de type «*reader makes right*» serait la plus appropriée pour ce problème.

Ensuite, une conséquence de l'utilisation des processus légers est que nous aurons en règle générale des temps de transferts plus longs que ceux des implantations de MPI spécialisées pour un réseau particulier. Cependant, ainsi que le montrent les performances (cf le chapitre suivant), nous nous situons toujours à des niveaux comparables.

L'implémentation des différents modes de transfert (*eager* et *rendez-vous*) est faite sans savoir ce que fait Madeleine en interne, ce qui implique une redondance des traitements. Par exemple, dans le cas d'un envoi de message en mode *rendez-vous* au niveau de MPI, nous utilisons notre implémentation, alors qu'il est tout-à-fait possible que Madeleine procède déjà de la sorte. Nous perdons mécaniquement du temps avec des *handshakes* inutiles réalisés au niveau de l'ADI, mais ces temps étant très courts, leur influence sur les performances est négligeable (nous sommes dans un contexte de messages longs). Le problème actuel se situe plutôt dans le cas ou le protocole *eager* est décidé dans l'ADI mais que Madeleine effectue un *rendez-vous* ou vice-versa. Pratiquement, ces cas litigieux sont évités car nos seuils sont alignés sur ceux de Madeleine.

Au niveau des communications, deux points sont susceptibles d'être améliorés : il s'agit des types de données définis par l'utilisateur (*user-defined datatypes*) qui passent encore par un tampon intermédiaire pour les émissions/réceptions et des opérations de communications collectives, qui sont celles de MPICH. Elle pourraient notamment être adaptées pour tirer parti de la topologie sous-jacente (machines multi-processeurs, grappes et partitions, etc.). De plus le système n'est pas asymétrique, c'est-à-dire que nous sommes dans l'incapacité de détecter qu'une communication est effectuée à courte ou longue distance. Dans ce cas, nous pourrions introduire des traitements supplémentaires afin d'optimiser les performances (compression par exemple). Ce problème pourrait disparaître avec l'introduction d'informations supplémentaires dans les fichiers de configuration de Madeleine.

Le dernier problème concerne la dynamicité puisque en l'état actuel des choses, les sessions MPI doivent être lancées par la même instance du lanceur d'applications. Nous aimerions que deux sessions MPI quelconques mais fonctionnant au-dessus de Madeleine puissent fusionner. Ce dernier point est en fait une limitation de Madeleine plus que de MPICH-Madeleine proprement dite, tout comme le fait que le recours à une approche centralisée avec Leony pose des problèmes pour l'extensibilité.

4.6.3 À propos de MPICH2 ...

Nous n'avons pas traité le cas de MPICH2 jusqu'à présent, mais nous souhaitons indiquer que dans le cadre de notre recherche, nous avons été invité par les concepteurs de MPICH pour mener une réflexion quant au développement et à l'intégration de nos mécanismes dans MPICH2.

Cette approche nous semble intéressante dans la mesure où la structure de MPICH2 met en place des principes que nous avons défendus dans ce document. C'est en particulier le cas du moteur de progression.

La gestion du multiprotocole est encore embryonnaire à l'heure actuelle, mais nous comptons travailler sur ces aspects dans le cadre de notre stage post-doctoral dans l'équipe de développement de MPICH.

Pour l'instant, nous avons déjà effectué un portage de Madeleine dans MPICH2, mais ce travail s'est déroulé au niveau de la *Channel Interface* et non de l'ADI (cf. 4.3.1).

Chapitre 5

Validation : résultats expérimentaux

Les chapitres précédents nous ont donné l'occasion d'exposer en détail notre architecture et la façon dont nous l'avons mise en œuvre en insistant parfois sur les optimisations réalisées afin de garantir un haut niveau de performances. Ce dernier chapitre montre que les idées implémentées pratiquement tiennent leurs promesses quand nous comparons MPICH-Madeleine à d'autres solutions. Dans un premier temps, nous décrivons les configurations matérielles utilisées pour effectuer ces différentes évaluations de performance, puis nous estimerons d'abord le surcoût de notre implémentation de MPI par rapport à la bibliothèque de communication Madeleine sur laquelle elle est fondée. Nous montrons ensuite que MPICH-Madeleine est capable d'exhiber de très bonnes performances aussi bien sur des grappes de machines multi-processeurs homogènes comme hétérogènes. Dans le reste de ce chapitre, un Kilo-octet représente 1024 octets et un Méga-octet représente 1024×1024 octets.

5.1 Description de la configuration matérielle

Nous avons principalement employé deux grappes hétérogènes pour mener à bien notre travail d'évaluation. La première de ces grappes, *Dalton*, est celle de l'équipe INRIA RUNTIME. La seconde grappe, *Jack*, est installée en Allemagne et sert de plate-forme de développement pour le projet CoC-Grid ([CoC]). Les chercheurs de ce projet procèdent également à l'évaluation de solutions logicielles pour l'exploitation des grappes de grappes, dont fait partie MPICH-Madeleine. Nous avons donc établi une collaboration (informelle, mais effective) entre nos deux équipes afin de partager les résultats de ces évaluations.

5.1.1 Dalton

Dalton est une grappe de quatre nœuds biprocesseurs munis de la technologie Intel HyperThreading. Ainsi, chaque nœud dispose de deux processeurs physiques possédant deux flots d'exécution simultanés. Ces processeurs sont de type Intel Xeon, cadencés à 2,66 GHz, avec 512 kilo-octets de mémoire cache. Chaque nœud possède également 1 Gigaoctet de mémoire vive. Au niveau des technologies réseaux, l'intégralité des nœuds possède un double câblage GigaBitEthernet et SCI (cartes de type D337). Deux des quatre nœuds sont de plus équipés de cartes Myrinet 2000.

5.1.2 Jack

La grappe Jack est composée de seize nœuds biprocesseurs. Huit de ces nœuds sont des AMD Athlon cadencés à 1,4 GHz, avec 512 Kilo-octets de mémoire cache et 512 Méga-octets de mémoire vive. Ces nœuds sont équipés des réseaux GigaBitEthernet et Myrinet (cartes Myrinet 2000). Les huit nœuds restants sont des Intel Xeons cadencés à 2,4 GHz avec 512 Kilo-octets de mémoire cache et 2 Gigaoctets de mémoire vive. Ces nœuds sont équipés de cartes SCI D331 et deux d'entre eux possèdent de plus une carte Myrinet, ce qui leur permet de jouer un rôle de passerelle.

5.2 Comparaison des performances avec Madeleine

Nous allons tout d'abord montrer que notre implémentation de MPI n'induit pas un surcoût important par rapport à la bibliothèque de communication Madeleine. Les tests conduits utilisent des canaux physiques ou virtuels et le programme est une série d'allers-retours de messages (*ping-pong*).

Dans le programme de test de Madeleine, les paquets ne sont envoyés qu'avec un unique empaquetage en mode `CHEAPER`, car la taille des données est connue *a priori*. Dans le cas d'un message MPICH-Madeleine, l'émission est effectuée généralement avec une série de deux empaquetages : un pour l'en-tête en mode `EXPRESS` et le cas échéant, un second pour le corps du message en mode `CHEAPER` (cf. 4.3.3.3). Afin d'obtenir des comparaisons justes et de pouvoir décomposer plus finement le surcoût introduit par MPICH-Madeleine, nous avons simulé dans le programme d'évaluation de Madeleine la façon dont nos messages sont transmis.

Les courbes montrent donc trois choses pour chaque catégorie de canaux (physiques ou virtuels) et pour l'ensemble des technologies réseaux considérées (SCI/SISCI, Myrinet/GM, GigaBitEthernet/TCP) :

- les performances de Madeleine avec le programme de test original : ces mesures sont dites effectuées *en mode direct* ;
- les performances de Madeleine simulant nos messages : ces mesures sont dites effectuées *en mode simulé* ;
- les performances de MPICH-Madeleine.

Nous présentons des résultats pour trois réseaux différents : SCI avec le protocole SISCI, Myrinet avec GM et GigaBitEthernet avec TCP. Dans chacun des cas, nous avons associé un unique canal au réseau sous-jacent. Les mesures ont été réalisées sur la grappe Dalton et les points des courbes sont en réalité obtenus avec une moyenne de 5 séries de 1000 allers-retours (nous faisons la moyenne sur la moitié du temps pris pour un aller-retour).

5.2.1 SCI/SISCI

Nous avons premièrement procédé à une évalution des performances de MPICH-Madeleine sur des canaux physiques qui sont ceux pour lesquels les performances sont maximales, avant de tester les canaux virtuels.

5.2.1.1 Canaux physiques

Les figures 46 et 47 nous montrent les performances dans le cas du réseau SCI pour les temps de transfert et le débit. Ces résultats peuvent être analysés de la façon suivante : au niveau de la latence, nous remarquons que pour des messages de faible taille (i.e inférieure à 8 octets), le surcoût est entièrement dû à MPICH. Pour des messages dont la taille est comprise entre 8 octets et 8 Kilo-octets, le surcoût se décompose en deux : une partie – la plus importante – est induite par le mode de transfert avec la séquence des deux empaquetages, l'autre partie est imputable à MPICH. Pour des messages de plus de 8 Kilo-octets, on remarque que le surcoût de MPICH devient négligeable est que c'est le fait d'envoyer un en-tête en mode `EXPRESS` suivi du corps en mode `CHEAPER` qui provoque une perte de performance. Enfin, l'optimisation consistant à envoyer un tampon de petite taille est clairement visible sur la courbe de latence : un léger saut se produit lorsque nous passons d'une taille de 8 à 16 octets dans les cas de Madeleine en mode simulé et de MPICH-Madeleine.

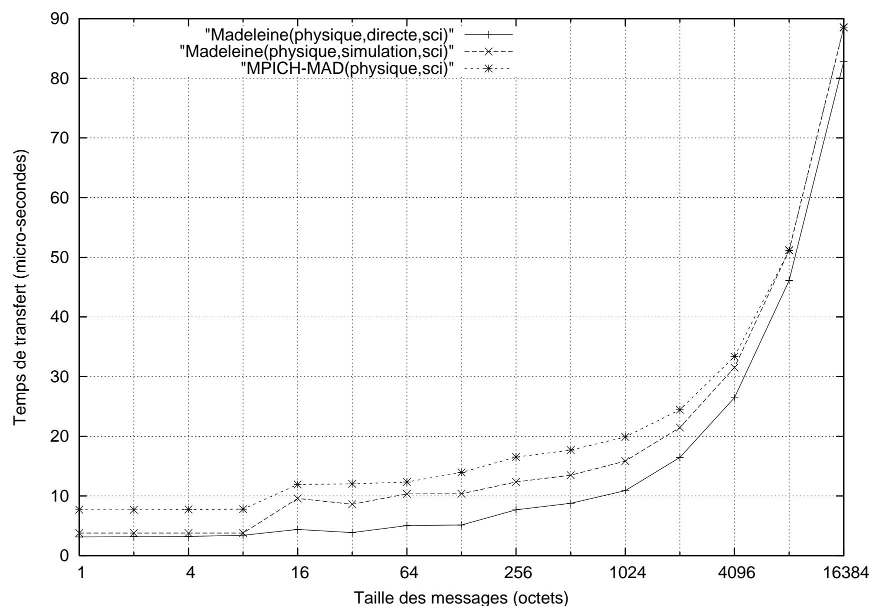


FIG. 46 – Comparaison des latences sur SISCI/SCI (canal physique)

L'observation des courbes de débit permet d'affiner cette analyse : pour des tailles de messages comprises entre 8 et 64 Kilo-octets, le surcoût est uniquement dû aux empaquetages successifs. Dans le cas de SCI, le seuil entre le mode de transfert *eager* et le mode *rendez-vous* est justement de 64 Kilo-octets : la courbe de débit marque un léger fléchissement à cette occasion. Pour des messages de taille plus importante, la différence provient essentiellement des messages de synchronisation car le fait d'envoyer un en-tête systématiquement ne se ressent presque pas.

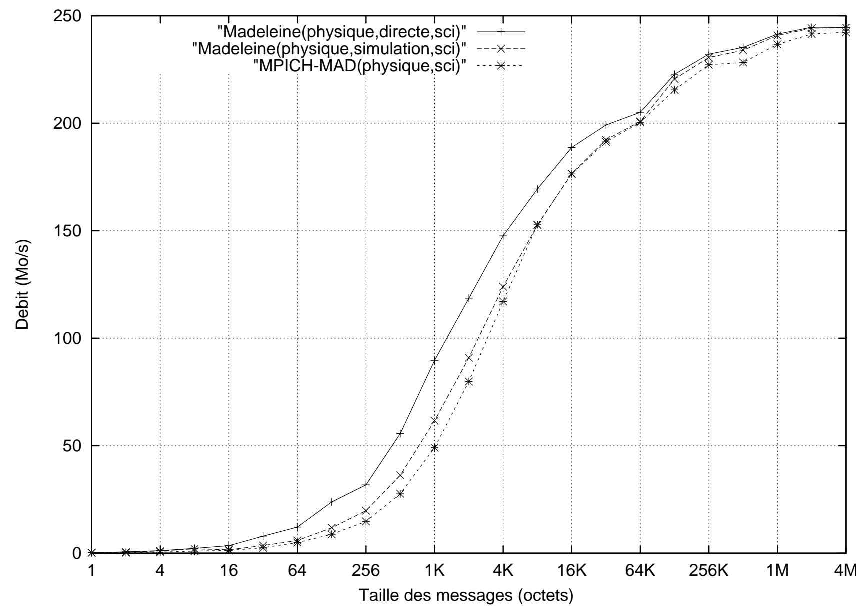


FIG. 47 – Comparaison des débits sur SISCI/SCI (canal physique)

5.2.1.2 Canaux virtuels

Les tendances observées avec les canaux physiques se confirment pour les canaux virtuels, comme le montrent les figures 48 et 49. Toutes les remarques concernant les différents seuils sont encore d'actualité : un tampon de 8 octets est utilisé pour les messages très courts et la transition entre les modes *eager* et *rendez-vous* s'effectue pour une taille de messages de 64 Kilo-octets. Cependant, dans le cas des canaux virtuels, le surcoût de MPICH-Madeleine par rapport à Madeleine est presque exclusivement causé par les empaquetages en série, ainsi que le montrent les courbes des débits.

5.2.1.3 Récapitulatif

Pour finir, le tableau suivant donne quelques chiffres-clefs des performances respectives de Madeleine et de MPICH-Madeleine :

		Temps de transfert minimal	Débit maximal
Madeleine	(canal physique)	$3,1\ \mu$s	$244,7$ Mo/s
MPICH-Madeleine	(canal physique)	$7,6\ \mu$s	$242,5$ Mo/s
Madeleine	(canal virtuel)	$7,9\ \mu$s	$244,5$ Mo/s
MPICH-Madeleine	(canal virtuel)	$10,2\ \mu$s	$242,3$ Mo/s

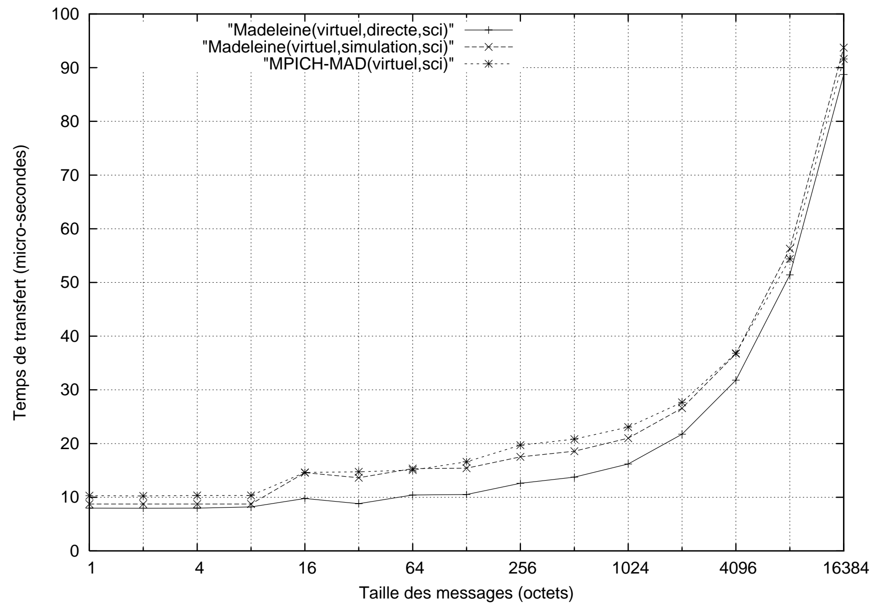


FIG. 48 – Comparaison des latences sur SISCI/SCI (canal virtuel)

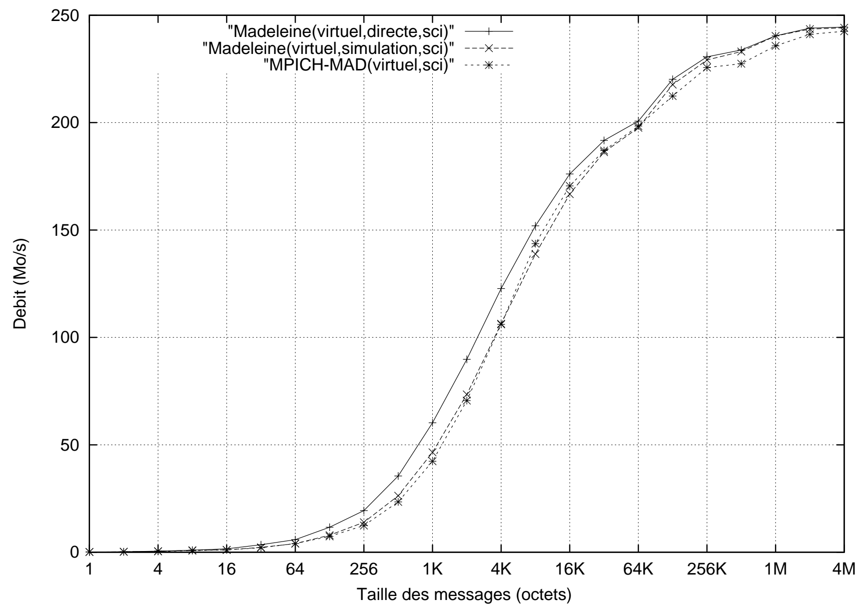


FIG. 49 – Comparaison des débits sur SISCI/SCI (canal virtuel)

5.2.2 Myrinet/GM

Nous passons maintenant au deuxième réseau haut-débit avec lequel nous avons procédé à cette évaluation : Myrinet, qui est sans doute à l'heure actuelle l'un des réseaux haut-débit les plus utilisés pour la construction de grappes de PC.

5.2.2.1 Canaux physiques

L'analyse des performances au-dessus du réseau Myrinet est légèrement différente que dans le cas de SCI. Nous remarquons d'emblée que le surcoût introduit est bien plus faible pour Myrinet que pour SCI, alors que les performances des deux technologies sont à peu près équivalentes. La décomposition de ce surcoût est également différente : pour des messages de taille inférieure à 4 Kilo-octets, le fait de devoir empaqueter les deux éléments du message (en-tête et corps) est négligeable par rapport à ce que MPICH introduit (cf. la figure 50). Nous pouvons toujours observer un léger saut pour la latence quand la taille des message dépasse 8 octets. Pour des messages de plus de 4 Kilo-octets, le même phénomène que dans le cas de SCI se reproduit : c'est la série des deux empaquetages qui provoque une perte de performance. Cependant, cette perte reste très limitée et la différence avec Madeleine est peu importante (quantitativement parlant).

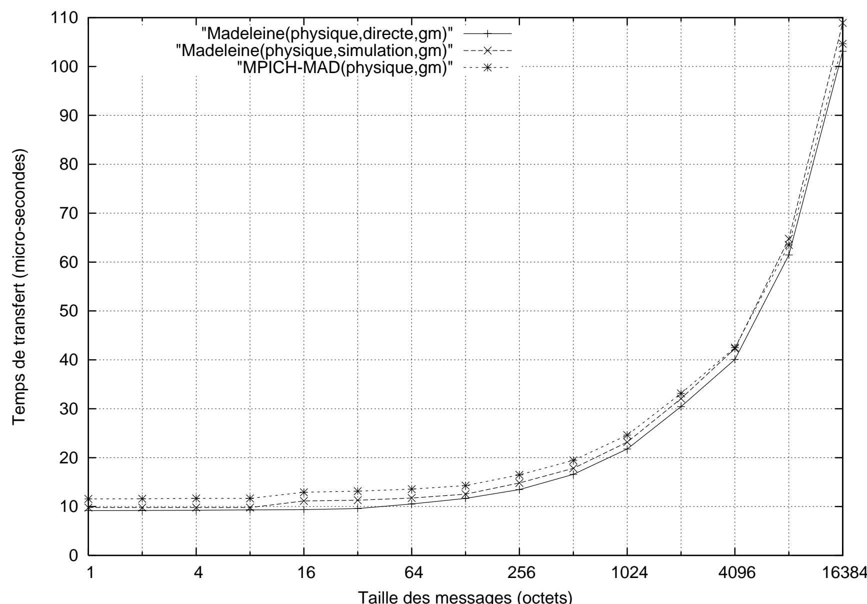


FIG. 50 – Comparaison des latences sur Myrinet/GM (canal physique)

Les courbes de débit (cf. Figure 51) confirment ceci : pour des messages dont la taille est comprise entre 4 et 128 Kilo-octets, c'est principalement le second empaquetage qui impacte les performances. Au-delà de cette taille, le mode de transfert *rendez-vous* est utilisé et le surcoût provient alors des messages de synchronisation nécessaires pour la mise en place de ce mode transfert. Cependant, pour des tailles de messages relativement importantes (plus de 1 Méga-octets), les débits de MPICH-Madeleine et de Madeleine se confondent

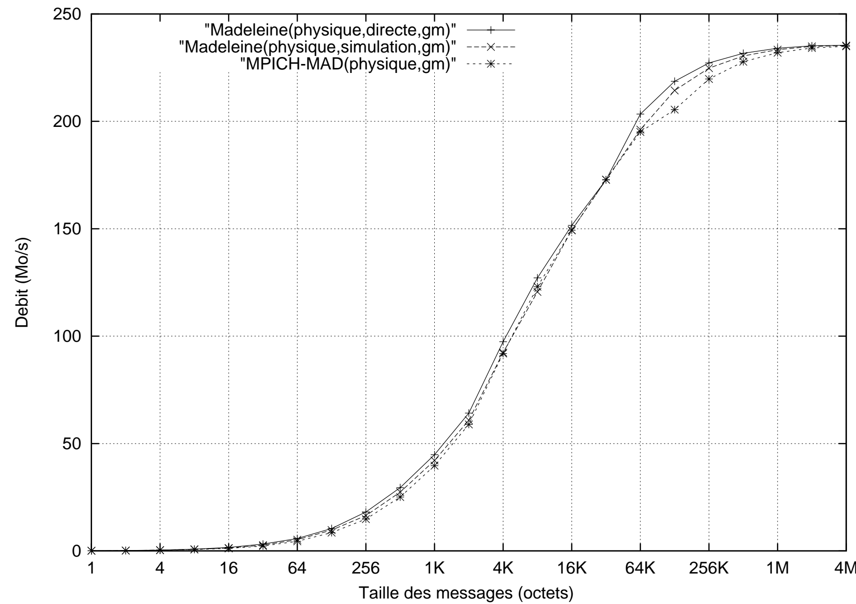


FIG. 51 – Comparaison des débits sur Myrinet/GM (canal physique)

pratiquement. Nous en profitons pour faire remarquer que la valeur de seuil dans le cas de Myrinet (128 Kilo-octets) est différente de celle employée pour SCI, ce qui est cohérent vis-à-vis de ce que nous avons expliqué en 4.3.3.7.

5.2.2.2 Canaux virtuels

Pour les canaux virtuels, l'exploitation de Madeleine dans le cas du mode de transfert *eager* est encore meilleure que dans le cas des canaux physiques car pour des tailles de messages allant jusqu'à 128 Kilo-octets, les trois courbes de débit se confondent (cf. la figure 53). Ensuite, le passage au mode *rendez-vous* entraîne un écart plus conséquent que dans le cas des canaux physiques, cet écart étant maximal pour des messages dont la taille est égale à la valeur du seuil de transition (i.e 128 Kilo-octets). Asymptotiquement, les débits de MPICH-Madeleine et de la bibliothèque Madeleine sont quasi-similaires.

5.2.2.3 Récapitulatif

Nous indiquons les temps de transferts minimaux ainsi que les débits maximaux dans le cas du réseau Myrinet à titre de comparaison.

	Temps de transfert minimal	Débit maximal
Madeleine (canal physique)	$9,1\ \mu$s	$235,5$ Mo/s
MPICH-Madeleine (canal physique)	$11,4\ \mu$s	$234,9$ Mo/s
Madeleine (canal virtuel)	$21,4\ \mu$s	$235,3$ Mo/s
MPICH-Madeleine (canal virtuel)	$23,7\ \mu$s	$234,4$ Mo/s

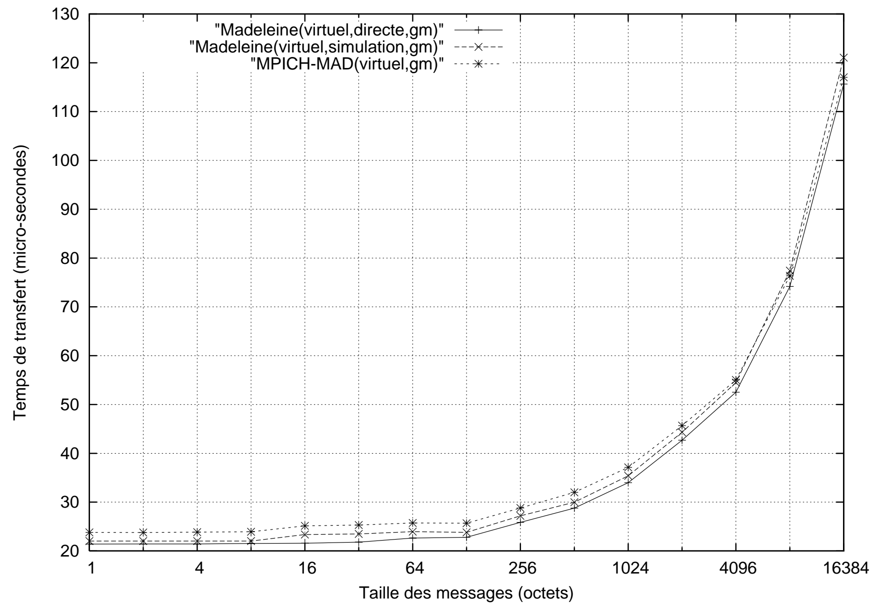


FIG. 52 – Comparaison des latences sur Myrinet/GM (canal virtuel)

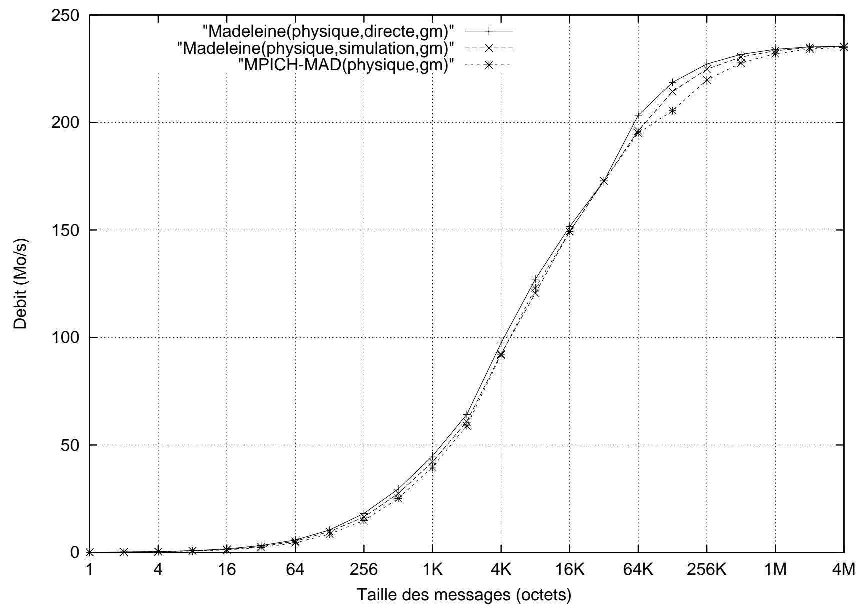


FIG. 53 – Comparaison des débits sur Myrinet/GM (canal virtuel)

5.2.3 GigaBitEthernet/TCP

Le dernier réseau pour lequel nous avons effectué des mesures est GigaBitEthernet, exploité par le protocole TCP. Bien qu'il s'agisse également d'un réseau à haut débit, l'utilisation d'un protocole dont la pile est localisée pour partie dans le noyau du système d'exploitation ne permet pas d'atteindre des performances similaires à celles vues précédemment.

Le cas de GigabitEthernet est quelque peu problématique car le matériel équipant la grappe Dalton ne fonctionnant pas correctement, nous avons choisi de présenter les performances obtenues sur la grappe Jack afin que les résultats soient significatifs et puissent confirmer les tendances observées avec les deux réseaux précédents.

5.2.3.1 Canaux physiques

Les résultats obtenus avec des canaux physiques sont montrés sur les figures 54 et 55. Nous voyons que pour les temps de transfert de messages de taille inférieure à 8 Kilo-octets, des différences existent et que le surcoût de MPICH est prépondérant par rapport à celui induit par les empaquetages successifs. Nous remarquons comme dans le cas de SCI un léger saut pour des tailles de messages supérieures à 8 octets, taille correspondant à l'optimisation où les données sont copiées dans l'en-tête. Pour une taille de message supérieure à 8 Kilo-octets, les performances sont quasi-identiques, ce qui est confirmé par les courbes de débit.

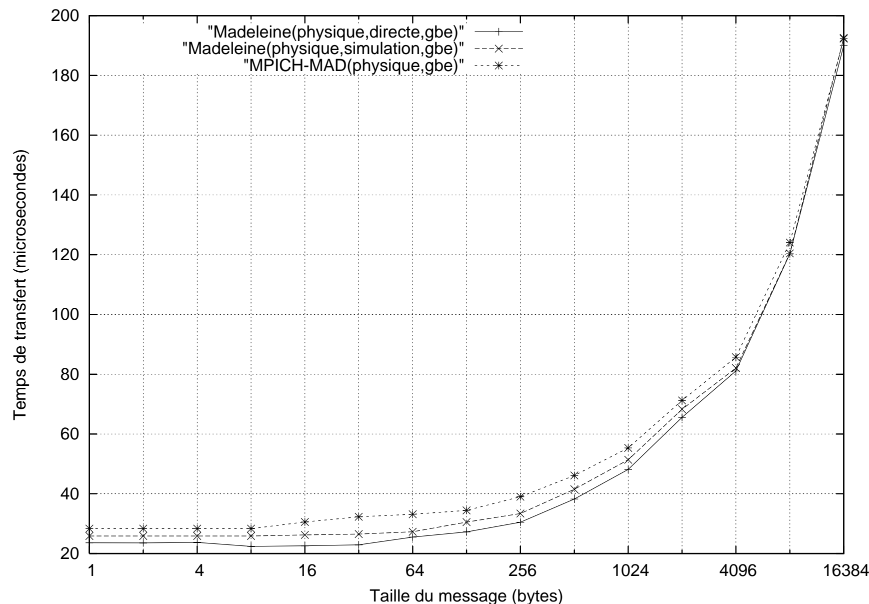


FIG. 54 – Comparaison des latences sur GigaBitEthernet/TCP (canal physique)

Ces dernières montrent clairement que la seule perte de performance entre Madeleine et MPICH-Madeleine est imputable au surcoût de MPICH, et que les empaquetages successifs ne se ressentent pas. En particulier, l'écart le plus important est observable pour des tailles de messages de 128 Kilo-octets, taille correspondant au seuil de transition du mode de transfert *eager* vers le mode *rendez-vous*. Ceci dit, MPICH-Madeleine parvient à réduire l'écart et ce

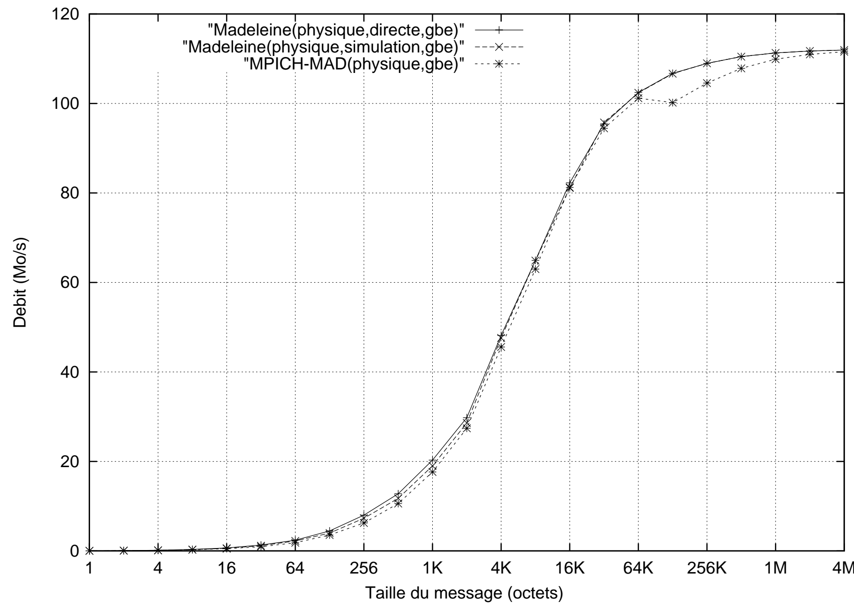


FIG. 55 – Comparaison des débits sur GigaBitEthernet/TCP (canal physique)

dernier est négligeable pour des tailles importantes de messages.

5.2.3.2 Canaux virtuels

Les constatations faites pour les canaux physiques s'appliquent également aux canaux virtuels (cf. les figures 56 et 57) : un léger saut se produit pour une taille de messages de 8 octets et à partir de messages de 4 Kilo-octets, les performances sont similaires. Entre 8 et 4 Kilo-octets, nous remarquons cette fois-ci que le surcoût de MPICH est inférieur à celui induit par la série d'empaquetages. Le point de transition situé à 128 Kilo-octets en toujours très marqué et le même phénomène que dans le cas des canaux physiques se répète, à savoir que l'écart en MPICH-Madeleine et Madeleine se réduit pour ne plus être perceptible.

5.2.3.3 Récapitulatif

Les chiffres donnés dans le tableau ci-dessous confirment bien que le protocole TCP empêche toute analyse fine des performances, en particulier pour les temps de transfert.

		Temps de transfert minimal	Débit maximal
Madeleine	(canal physique)	$22.4\ \mu$s	$111,9$ Mo/s
MPICH-Madeleine	(canal physique)	$28,1\ \mu$s	$111,6$ Mo/s
Madeleine	(canal virtuel)	$33,2\ \mu$s	$111,8$ Mo/s
MPICH-Madeleine	(canal virtuel)	$35,2\ \mu$s	$111,5$ Mo/s

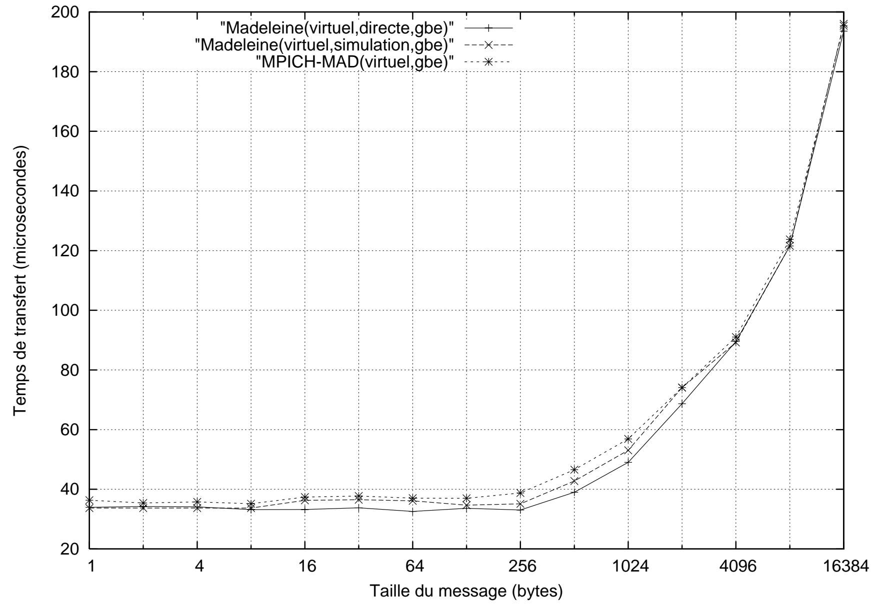


FIG. 56 – Comparaison des latences sur GigaBitEthernet/TCP (canal virtuel)

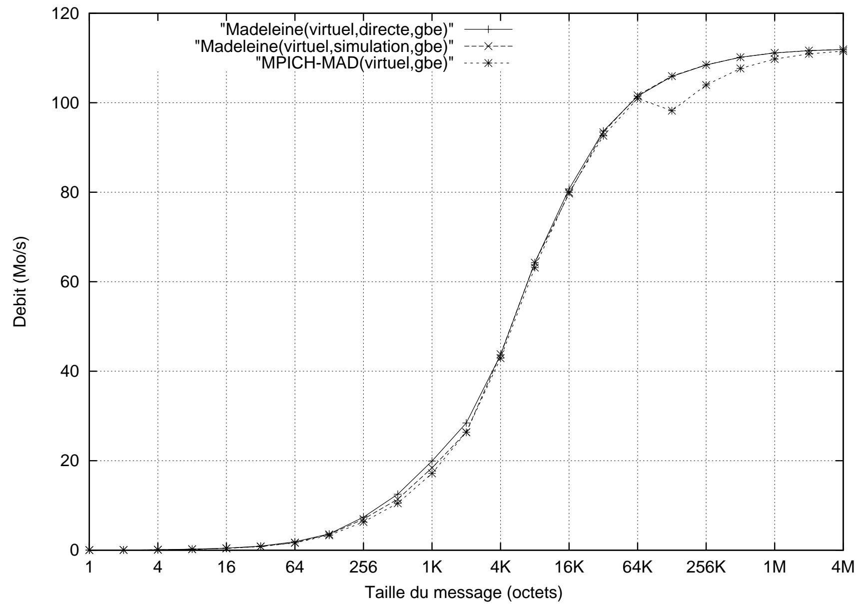


FIG. 57 – Comparaison des débits sur GigaBitEthernet/TCP (canal virtuel)

5.2.4 Conclusion

Au regard des mesures effectuées sur les trois réseaux considérés (SCI, Myrinet et GigaBitEthernet), nous pouvons affirmer que compte-tenu de la structure des messages de MPICH-Madeleine – structure contrainte par l'utilisation de Madeleine – notre exploitation de cette bibliothèque de communication est très satisfaisante. En effet, nous constatons que les couches supplémentaires de MPICH ainsi que l'introduction des processus légers n'ont qu'une influence minime sur les performances. La seule conséquence notable est que les temps de transferts minimaux sont plus élevés.

Nous constatons de plus que le fait d'émettre un message Madeleine avec deux empaquetages (l'en-tête en mode `EXPRESS` suivi du corps en mode `CHEAPER`) joue un rôle prépondérant dans la dégradation des performances. Les différences d'écart entre les réseaux considérés nous amène à penser que dans le cas de SCI, des optimisations pourraient être apportées à Madeleine pour cette série particulière d'empaquetages.

Enfin, la comparaison des performances obtenues au-dessus de Madeleine valide le premier principe de notre démarche, à savoir que l'introduction des processus légers ne grève pas les performances.

5.3 Évaluation sur grappes homogènes de machines multi-processeurs

Nous présentons maintenant les différents tests effectués sur des grappes homogènes de machines multi-processeurs. Nous comparons d'abord les latences et débits de MPICH-Madeleine avec ceux de solutions concurrentes puis montrons en quoi notre moteur de progression peut améliorer le temps d'exécution d'une application utilisant MPICH-Madeleine. Pour finir, nous montrons les résultats obtenus avec une application plus représentative que les tests habituels de type *ping-pong*. Sauf mention contraire, les résultats exposés dans cette section ont été obtenus sur la grappe Dalton.

5.3.1 Représentativité des tests

Les métriques habituellement utilisées pour l'évaluation des opérations point-à-point d'une implémentation de MPI sont le temps de transfert (ou *latence*) et le débit. Pour arriver à déterminer leurs valeurs, l'utilisation de programmes de test de type *ping-pong* est très répandue. Cependant, un test de cette nature est très artificiel, car il est assez rare que des applications réelles puissent apparier aussi simplement leurs émissions et leurs réceptions.

Pour contourner ce problème, nous avons choisi autant que faire se peut d'utiliser une «vraie» application. Il s'agit en l'occurence de *High Performance Linpack* ([HPL]), une application de calcul matriciel utilisée pour le classement du Top 500 ([TOP]).

Nous avons choisi de ne pas procéder à l'évaluation des opérations collectives : n'ayant pas travaillé à leur optimisation en fonction de la topologie, elles sont identiques à celles de MPICH. De plus, une telle évaluation doit être basée sur des méthodologies rigoureuses qui rendent le processus de mesure complexe (cf. [dSK99]).

5.3.2 Performance des communications point-à-point

Nous avons effectué des comparaisons entre MPICH-Madeleine et d'autres implémentations spécialisées dans l'exploitation de grappes homogènes de machines multi-processeurs. Nous avons sélectionné les systèmes suivants :

- pour les communications par mémoire partagée : MPICH-SHMEM, qui est développé – comme P4 – par Argonne pour le support exclusif des machines à mémoire partagée, MPICH-P4, MPICH-GM et SCI-MPICH qui ont été configurés tous trois pour le support des communications par mémoire partagée ;
- pour les communications avec le réseau Myrinet, nous avons choisi MPICH-GM ;
- pour les communications avec le réseau SCI, nous avons opté pour SCI-MPICH qui est la seule implémentation librement disponible actuellement. Nous avons de plus configuré le logiciel pour utiliser des processus légers chargés de recevoir les messages, qui est un comportement un peu similaire sur le principe à celui de MPICH-Madeleine ;
- pour les communications avec le réseau GigaBitEthernet, l'implémentation de référence est MPICH-P4.

Comme dans le cas des comparaisons avec Madeleine, chaque mesure est une moyenne de 5 séries de 1000 allers-retours.

5.3.2.1 Communications en mémoire partagée

Les performances obtenues pour les communications par mémoire partagée sont données par les figures 58 et 59. Nous remarquons que pour les latences, MPICH-Madeleine

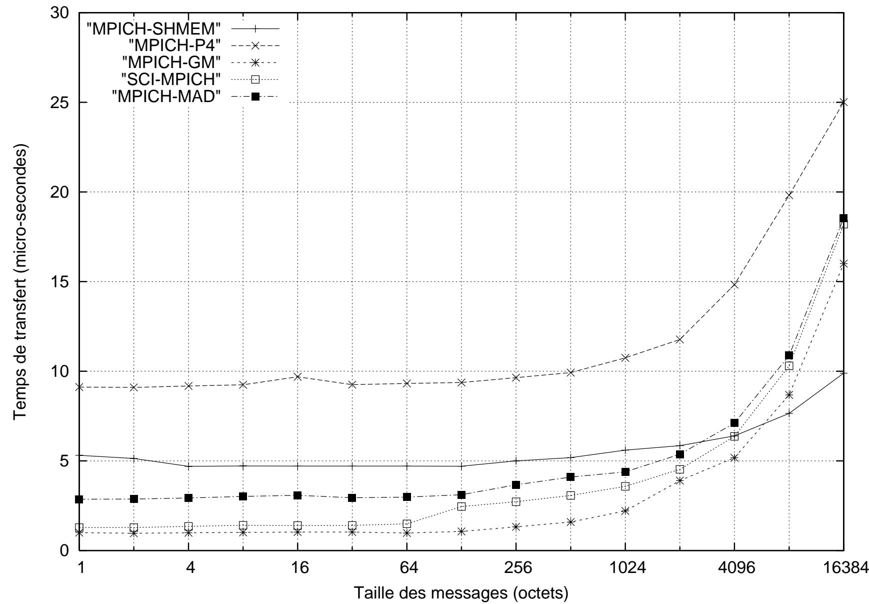


FIG. 58 – Comparaison des latences en mémoire partagée

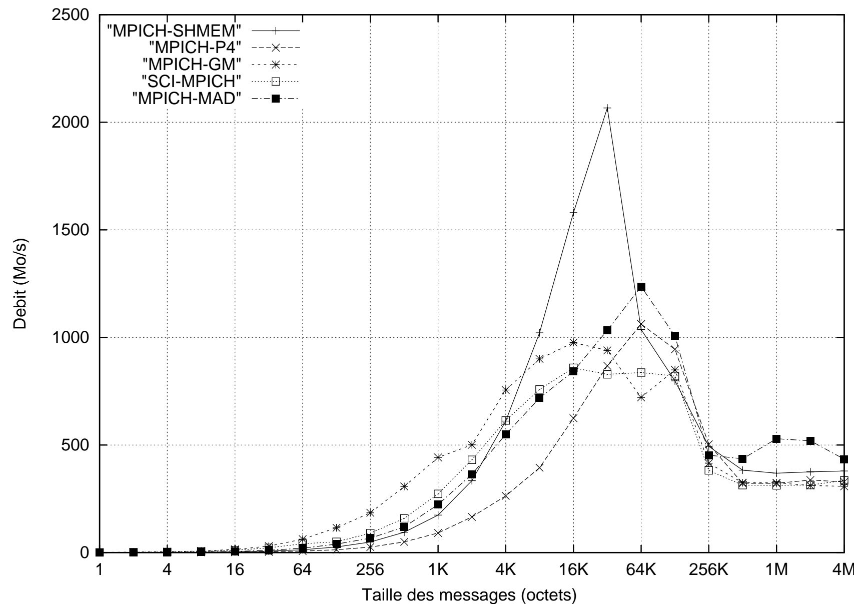


FIG. 59 – Comparaison des débits en mémoire partagée

se situe dans la moyenne : SCI-MPICH et surtout MPICH-GM ont des temps de transferts

plus bas, tandis que MPICH-P4 et MPICH-SHMEM ont des latences supérieures à la nôtre. On remarquera également que seul MPICH-GM est capable de descendre en-dessous de la micro-seconde pour ses transferts en mémoire partagée. Au niveau des débits (cf. la figure 59), toutes les coubes ont la même allure, avec un pic qui correspond à la taille maximale à partir de laquelle le cache n'est plus suffisant pour contenir tout le message. Nous remarquons cependant que le comportement de MPICH-Madeleine est très satisfaisant car nous arrivons à maintenir un débit plus important que les autres solutions pour des messages de taille supérieure à 64 Kilo-octets.

5.3.2.2 SCI/SISCI

Les figures 60 et 61 nous montrent les performances de MPICH-Madeleine comparées à celles de SCI-MPICH. Au niveau de la latence (cf. Figure 60), nous observons toujours le saut à 8 octets qui correspond à une optimisation de MPICH-Madeleine pour les messages très courts. Un saut similaire est observable pour SCI-MPICH, mais se produisant à 64 octets. Cette taille correspond à une limite pour un mode de transfert optimisé dans le cas du réseau SCI et comme SCI-MPICH est implémenté directement au-dessus de la couche bas-niveau, ce genre d'optimisations lui est possible, ce qui n'est pas notre cas. Ceci dit, il ne s'agit que d'un léger désagrément car pour des messages de taille supérieure à 64 octets les temps de transferts atteints par les deux implémentations sont identiques. Les courbes de

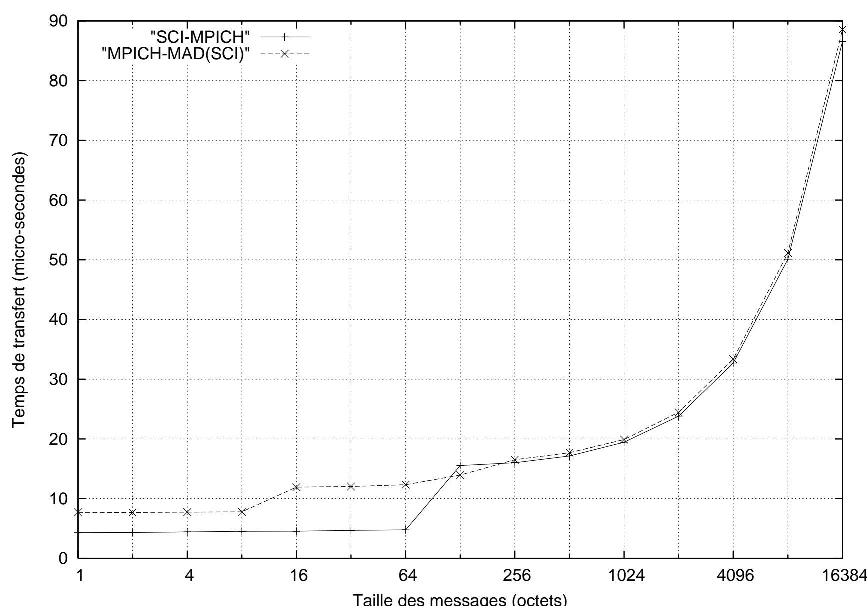


FIG. 60 – Comparaison des latences sur SISCI/SCI

débit (cf. Figure 61) confirment ce phénomène car entre 128 octets et jusqu'à 16 Kilo-octets,ils sont quasi-identiques. Au-delà de cette taille, MPICH-Madeleine offre des performances supérieures à celle de SCI-MPICH. Nous remarquons de plus que le demi-débit est atteint pour une taille de messages de 4 Kilo-octets dans le cas de notre implémentation de MPI.

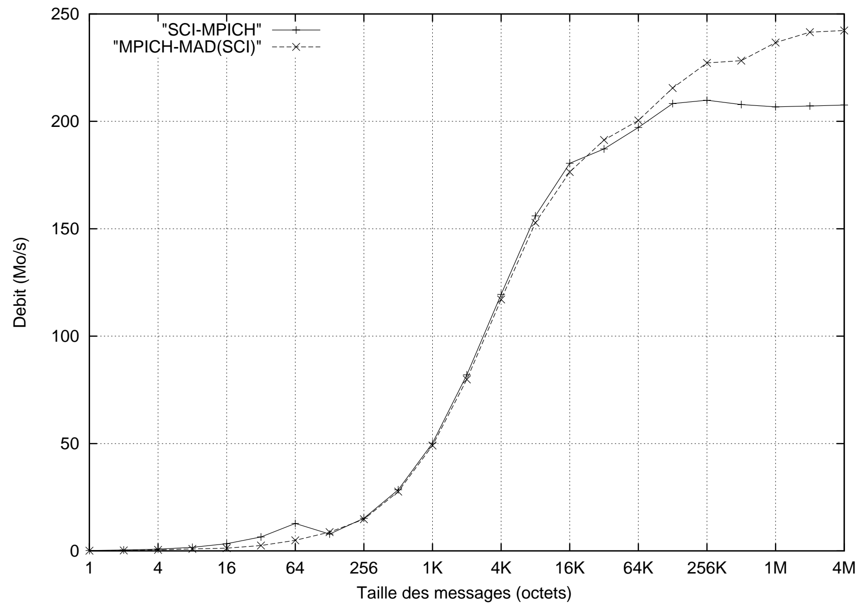


FIG. 61 – Comparaison des débits sur SISCI/SCI

5.3.2.3 Myrinet/GM

Dans le cas du réseau Myrinet, la situation est presque similaire : pour des tailles de messages inférieures à 256 Kilo-octets, MPICH-GM est plus rapide que MPICH-Madeleine (cf. les figures 62 et 63). Nous noterons cependant qu'entre 16 et 64 Kilo-octets, cet écart est

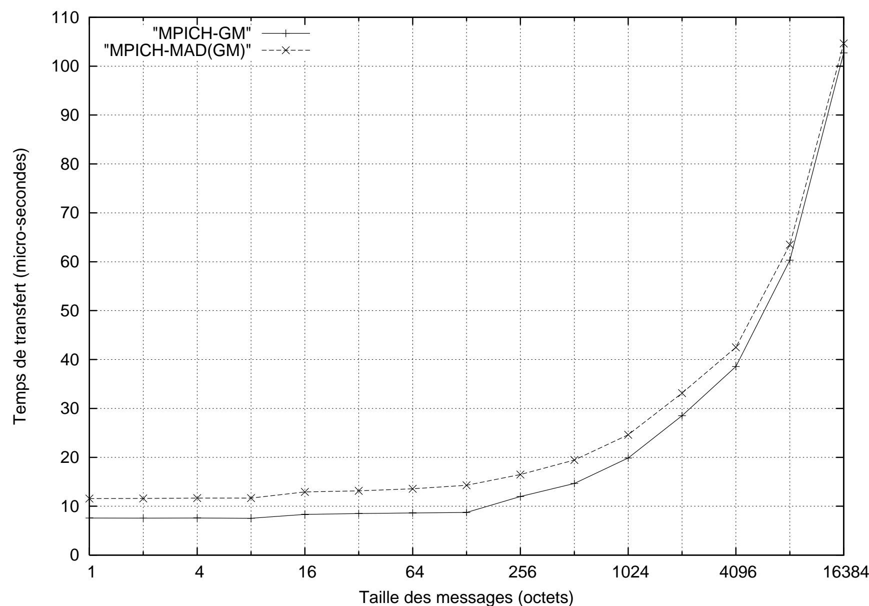


FIG. 62 – Comparaison des latences sur Myrinet/GM

vraiment très faible, et qu'à partir de 256 Kilo-octets, MPICH-Madeleine est légèrement plus performant que MPICH-GM, qui constitue l'implémentation de référence de MPI pour le réseau Myrinet, car développée par la société Myricom (fabricante de ce réseau). Enfin, le demi-débit est atteint pour des messages de 8 Kilo-octets pour MPICH-Madeleine.

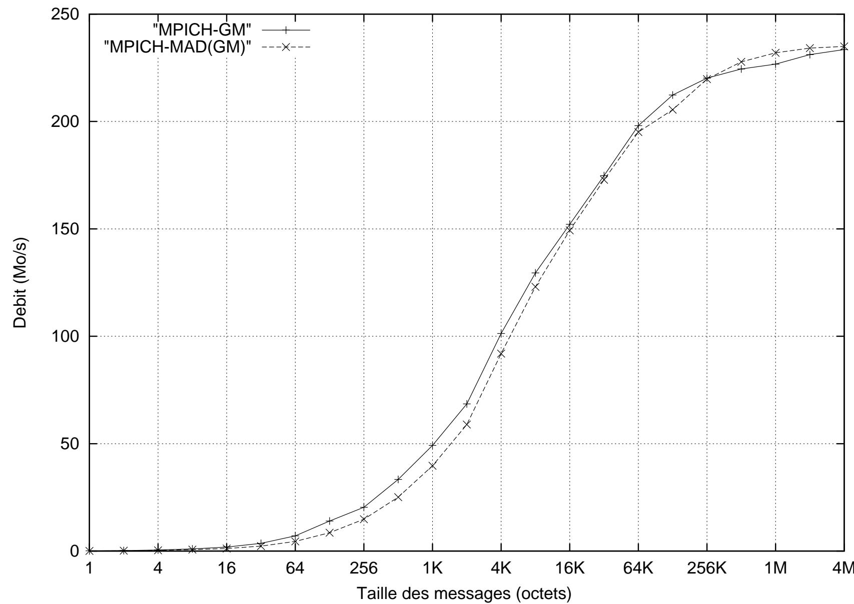


FIG. 63 – Comparaison des débits sur Myrinet/GM

5.3.2.4 GigabitEthernet/TCP

Nous terminons cette série de comparaisons avec les mesures de MPICH-P4. Nous rappelons à cette occasion que cette implémentation de MPI est souvent considérée comme celle de référence pour ce type de réseau. Les résultats sont synthétisés par les courbes des figures 64 et 65. La latence de MPICH-Madeleine est inférieure à celle de MPICH-P4 pour des messages de faible taille (i.e inférieure à 8 Kilo-octets) et entre 8 et 16 Kilo-octets, les performances des deux systèmes sont similaires. Au-delà de cette taille, un décrochage s'opère et MPICH-Madeleine exploite mieux le réseau GigaBitEthernet. Le demi-débit est atteint à partir de 8 Kilo-octets.

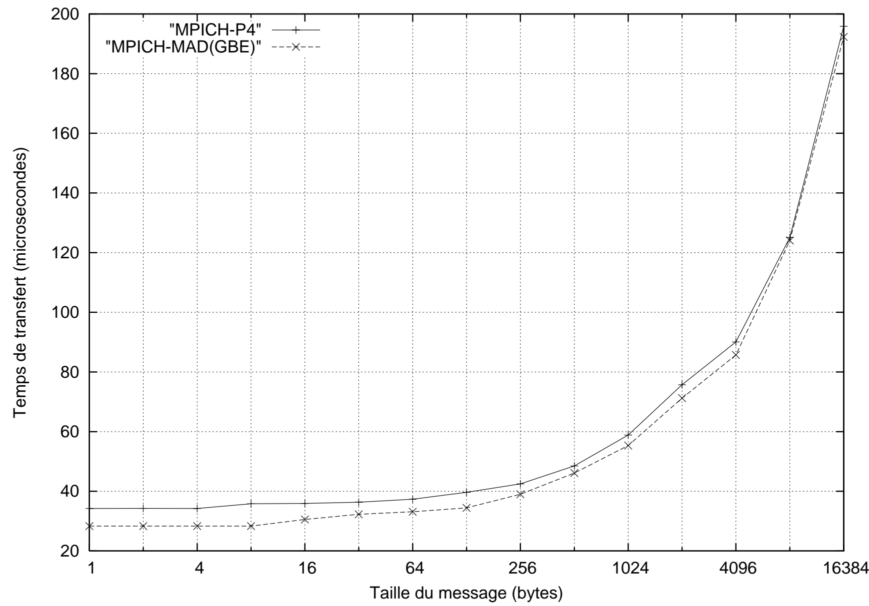


FIG. 64 – Comparaison des latences sur GigaBitEthernet/TCP

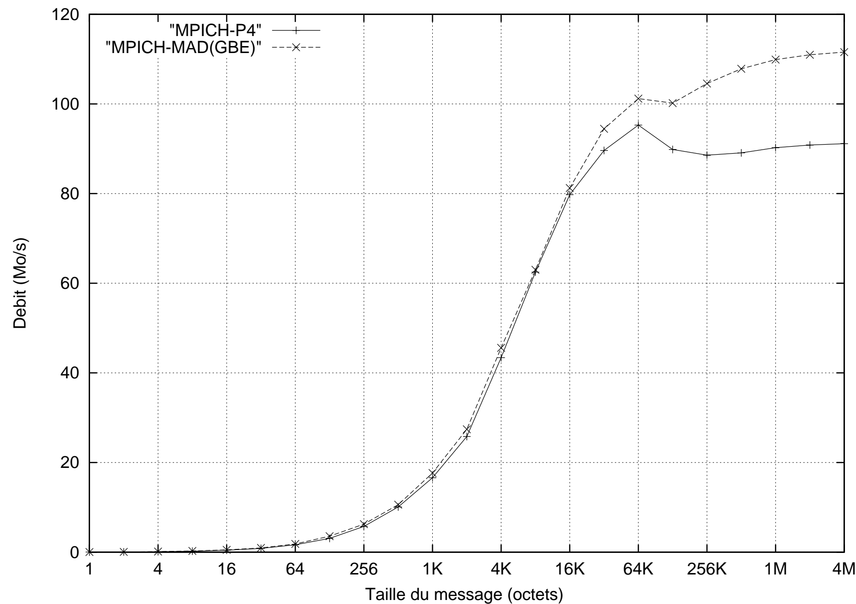


FIG. 65 – Comparaison des débits sur GigaBitEthernet/TCP

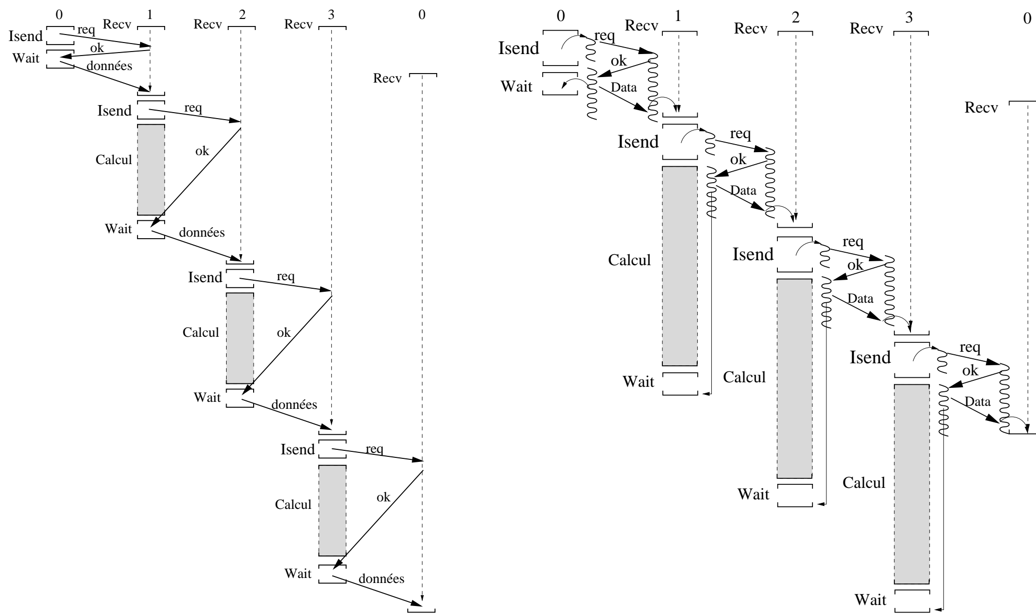
5.3.3 Amélioration de la progression des opérations non-bloquantes

L'amélioration de la progression des communications des applications utilisant MPICH-Madeleine est le point majeur sur lesquel nous avons travaillé ainsi que le montre notre architecture. Nous allons donc voir les résultats obtenus dans ce domaine.

5.3.3.1 Mise en évidence du problème et comportement des implémentations de MPI

Nous supposons que nous disposons d'un ensemble de n processus MPI (avec un processus par nœud) organisés en anneau. Le processus-maître, de rang nul, commence par effectuer un envoi non-bloquant (`MPI_Isend`) à son voisin (de rang 1) puis attend que cet envoi soit bien effectué avec un appel à `MPI_Wait`. Il attend ensuite de recevoir un message du processus $n - 1$. Les autres processus effectuent quant à eux les opérations suivantes :

- une réception bloquante (`MPI_Recv`) d'un message provenant du processus prédécesseur ;
- un envoi non-bloquant (`MPI_Isend`) du même message au processus successeur ;
- une phase de calcul pendant laquelle aucun appel à la bibliothèque MPI n'est effectué ;
- un appel à `MPI_Wait` à l'issue de cette phase de calcul pour attendre que le message envoyé avec `MPI_Isend` est bien parti.



(a) Implémentation non-bloquante et non asynchrone

(b) MPICH-Madeleine

FIG. 66 – Amélioration de la progression des communications non-bloquantes (exemple avec quatre processus)

Si le message échangé est de courte taille, c'est-à-dire que le mode de transfert est le mode *eager*, il n'y a pas de problèmes. Mais dans le cas où le message est suffisamment volumineux pour que le mode de transfert *rendez-vous* soit employé, alors des effets indésirés surviennent si l'implémentation ne fait pas appel à des processus légers. La figure 66(a) montre une telle situation dans le cas de quatre processus.

Nous voyons que le problème se manifeste au niveau des phases de calcul : pendant qu'un processus est dans cette phase, il est est incapable de prendre en compte le message

d'acquitement de son successeur pour envoyer les données. Ce message ne sera donc reçu qu'à la fin de cette phase de calcul et c'est lors de l'appel à `MPI_Wait` que la transaction a réellement lieu. Ainsi, `MPI_Isend` est non-bloquant, mais ne fait pas progresser les communications de façon asynchrone, ce que veut en fait le programmeur d'applications. La conséquence d'un tel comportement est que le programme est bien distribué, mais non parallèle, car les phases de calcul ne se recouvrent pas. Le temps total d'exécution d'un tel programme de test sera donc proportionnel au nombre de processus et à la durée des phases de calcul. Il convient de noter que ceci est purement un problème de mise en œuvre du standard, et pas de MPI lui-même.

La figure 66(b) montre le comportement de MPICH-Madeleine dans une telle situation : comme nous disposons de processus légers capables de faire progresser les communications même quand aucun appel à la bibliothèque MPI n'est fait, la prise en compte des messages d'acquitement se fait durant la phase de calcul, et non à la fin. Les données peuvent également être envoyées pendant cette phase, ce qui permet de débloquer le processus successeur dans l'anneau. Ce processus peut alors envoyer à son tour le message et ainsi de suite. Dans ce cas, les phases de calcul se déroulent vraiment en parallèle, ce qui fait que le temps total d'exécution est bien plus court que dans le cas d'une implémentation sans processus légers et que les communications non-bloquantes sont véritablement asynchrones.

5.3.3.2 Résultats expérimentaux

Nous avons mesuré le temps d'exécution pour un anneau composé de huit processus et comparons MPICH-Madeleine avec deux implémentations concurrentes : MPICH-P4 (pour le réseau GigaBitEthernet) et MPICH-GM (pour le réseau Myrinet). Comme nous ne disposions pas d'assez de nœuds équipés de cartes Myrinet sur la grappe Dalton, les mesures dans ce cas ont été effectuées sur la grappe Jack. Nous avons utilisé des messages de 2 Kilo-octets dans le cas du mode de transfert *eager* et de 1 Méga-octet dans le cas du mode *rendez-vous*.

Comparaison avec MPICH-GM

Version de MPI	Temps total (Protocole *eager*)	Temps Total (Protocole *rendez-vous*)
MPICH-GM	$11,5$ sec.	$130,8$ sec.
MPICH/Madeleine	$11,8$ sec.	$19,2$ sec.

Comparaison avec MPICH-P4

Version de MPI	Temps total (Protocole *eager*)	Temps Total (Protocole *rendez-vous*)
MPICH-P4	$5,6$ sec.	$45,5$ sec.
MPICH/Madeleine	$8,1$ sec.	$9,5$ sec.

La différence de durée entre les tests effectués sur Myrinet et ceux effectués sur GigabitEthernet est imputable aux différences entre les processeurs (Athlon vs Xeon) qui se répercutent sur la durée des phases de calcul.

Ce test pourra être jugé artificiel et peu représentatif [1], mais il met bien en évidence les problèmes de progression des communications des implémentations classiques de MPI.

5.3.4 Conservation de la puissance de calcul

Le début de cette section a été consacré aux performances des opérations point-à-point. Cependant, ainsi que nous le précisons au début de ce chapitre, il est nécessaire de devoir confirmer l'efficacité de notre implémentation de MPI en la confrontant à une application «du monde réel». Nous avons donc opté pour des évaluations de notre implémentation de MPI avec l'application *High Performance Linpack* (HPL).

5.3.4.1 High Performance Linpack

HPL ([HPL]) est une application de calcul matriciel qui est utilisée pour procéder au classement des machines parallèles pour le *Top 500*, en évaluant la puissance de calcul en nombre d'opérations flottantes à la seconde (flops). Cette application est donc plus représentative que les tests point-à-point habituellement employés car elle est écrite en Fortran et utilise la bibliothèque MPI en situation réelle.

Cependant, bien que HPL utilise MPI, les résultats obtenus dépendent surtout des *Basic Linear Algebra Subroutines* (BLAS) utilisées. Elles doivent être spécialement créées pour les processeurs considérés et l'emploi de BLAS génériques a pour effet de tirer les résultats vers le bas. Dans ce cas, le gain potentiel apporté par une implémentation de MPI très optimisée disparaît totalement. De même, l'utilisation d'un réseau haut-débit plutôt que FastEthernet n'apporte presqu'aucun gain de performances ! Il est donc essentiel de pouvoir disposer des BLAS correspondantes aux processeurs des machines cibles. Ceci pose tout naturellement un problème, même en environnement homogène, car tous les processeurs doivent être strictement identiques (taille des caches, fréquence d'horloge, etc).

5.3.4.2 Résultats expérimentaux

HPL effectue des opérations sur une matrice carrée dont le nombre de colonnes constitue la *taille du problème* (cf. les figures 67, 68, 69 et 70). Il existe une taille optimale qui permet de déterminer la puissance de calcul maximale de la machine parallèle cible, mais nous n'avons pas cherché à déterminer cette taille. Nous avons plus simplement voulu établir des comparaisons entre différentes implémentations de MPI et par conséquent nous avons fixé des tailles de problème suffisamment grandes pour que les tests soient significatifs. En ce qui concerne les mesures réalisées, chaque résultat est la moyenne de deux séries de six tests. Toutes les mesures ont été faites sur la grappe Dalton afin que les comparaisons soient le plus justes possible.

Résultats avec un nœud La figure 67 montre une évaluation effectuée sur un unique nœud, avec quatre processus qui communiquent donc en utilisant de la mémoire partagée. Nous avons choisi une configuration avec quatre processus car les nœuds de la grappe Dalton

[1] il ne l'est cependant pas moins qu'un test de type ping-pong, plus communément admis

possèdent une technologie de processeurs HyperThreading, ce qui fait que logiquement, nous disposons de quatre processeurs. Les systèmes évalués sont les mêmes que dans le cas des tests en mémoire partagée décrits en 5.3.2.1 : MPICH-SHMEM, MPICH-P4, MPICH-GM et SCI-MPICH. Les résultats obtenus sont très comparables et ne permettent pas de trancher en faveur d'une implémentation ou d'une autre.

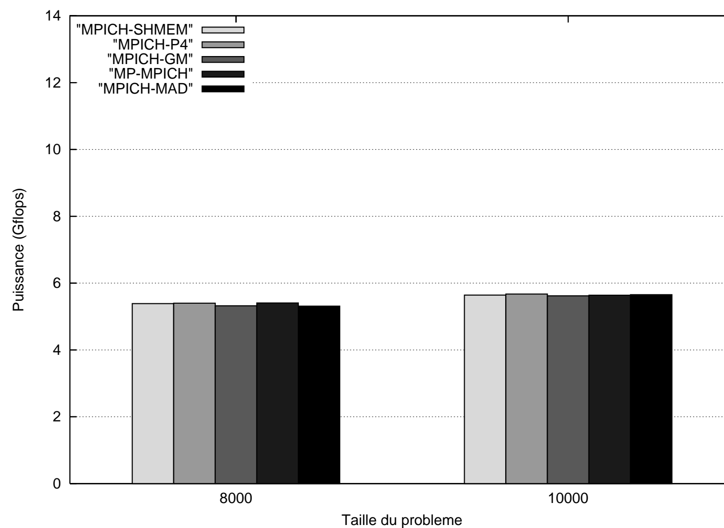


FIG. 67 – Test HPL sur un nœud avec 4 processus

Résultats avec 2 nœuds Nous avons ensuite voulu observer le comportement de diverses implémentations de MPI dans le cas où des communications réseaux entrent en ligne de compte. Pour ce faire, nous avons utilisé deux nœuds de la grappe et avons lancé quatre processus par nœud (pour les même raisons que précédemment). Dans ce cas, nous avons donc une configuration multiprotocole, car les communications passent par de la mémoire partagée et par le réseau. Les figures 68 et 69 montrent les résultats obtenus pour les réseaux GigaBitEthernet, Myrinet et SCI. Les implémentations utilisées pour les tests sont donc – outre MPICH-Madeleine – MPICH-P4, MPICH-GM et SCI-MPICH. Le cas de SCI-MPICH est problématique car ce logiciel s'est révélé incapable d'exploiter les nœuds avec la technologie HyperThreading (i.e quatre processus par nœud). Nous indiquons néanmoins les résultats obtenus par notre implémentation (c.f la figure 68). Afin de pouvoir effectuer des comparaisons en environnement multi-protocole avec des communications réseaux et par mémoire partagée, nous avons désactivé l'Hyperthreading pour ne lancer que deux processus par nœud et avoir la garantie qu'ils se verraient affecter un processeur physique chacun (c.f la figure 69).

La première remarque que nous tenons à faire, c'est que conformément à ce que nous avions annoncé, HPL est peu sensible à la performance du réseau utilisé : il n'y a que peu de différence entre les résultats obtenus avec MPICH-P4 et MPICH-GM (moins de 20%) au regard de l'écart substantiel entre les réseaux Myrinet et GigaBitEthernet. MPICH-Madeleine donne des résultats légèrement plus élevés que ceux de MPICH-P4 avec environ 3% d'écart en notre faveur et plus bas que ceux de MPICH-GM avec un écart d'environ 4% en faveur

de l'implémentation de Myricom. Dans le cas de SCI, l'écart est minime en notre défaveur et de l'ordre de 1%.

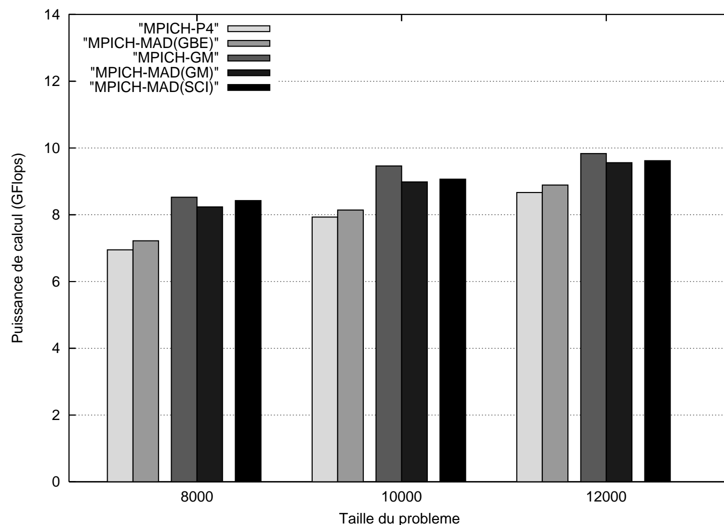


FIG. 68 – Test HPL sur 2 nœuds avec 8 processus (HyperThreading activé)

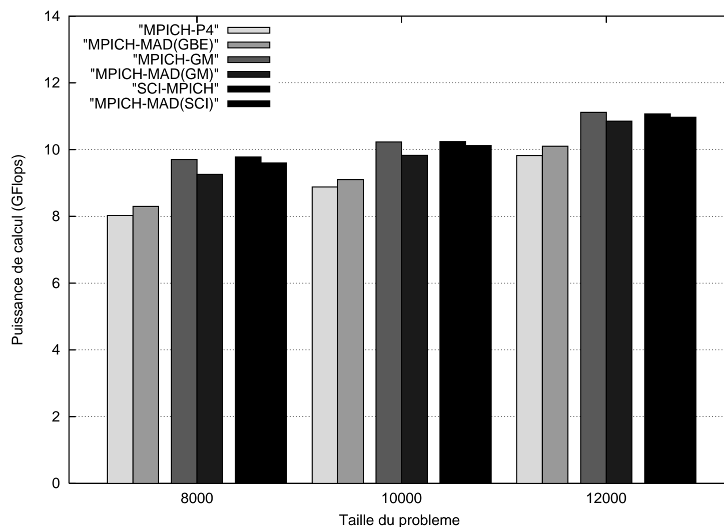


FIG. 69 – Test HPL sur 2 nœuds avec 4 processus (HyperThreading désactivé)

Ces résultats sont satisfaisants car le fait que l'ordonnancement des processus légers Marcel soit préemptif pouvait laisser craindre un ralentissement de l'application : HPL étant très sensible aux BLAS et donc à l'utilisation des caches des processeurs, un tel ordonnancement aurait pu se révéler en pratique contre-productif.

Résultats avec 4 nœuds Enfin la dernière évalution a été faite sur la totalité de la grappe Dalton : nous avons lancé 16 processus sur les quatre nœuds disponibles. Comme nous ne

disposions que de deux nœuds équipés de cartes Myrinet, seul le réseau GigaBitEthernet est utilisé. La figure 70 confirme le comportement de MPICH-Madeleine par rapport à MPICH-P4 : les résultats obtenus sont plus élevés, avec un écart d'environ 4,5% en notre faveur. À titre de référence, nous donnons également les résultats obtenus avec SCI-MPICH dans ce cas de figure où l'HyperThreading est activé.

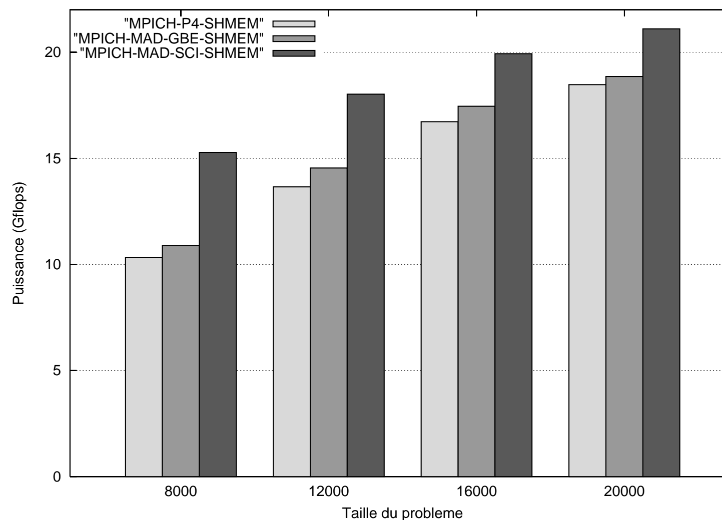


FIG. 70 – Test HPL sur 4 nœuds et 16 processus

5.3.5 Conclusion

Nous donnons le tableau ci-après à titre de rappel, et y avons ajouté les chiffres d'une implémentation de MPI que nous n'avons pas pu tester nous-mêmes : ChaMPIon/Pro. Cependant, nous avons jugé intéressant de fournir ces points de comparaisons car la plate-forme décrite pour obtenir ce résultat était similaire à la nôtre.

Version de MPI	Temps de transfert minimal	Débit maximal
MPICH-P4	$34.2\ \mu$s	$95,2$ Mo/s
MPICH-Madeleine (GBE)	$28.1\ \mu$s	$111,6$ Mo/s
MPICH-GM	$7,5\ \mu$s	$233,5$ Mo/s
chaMPIon/Pro (GM)*	$7,3\ \mu$s	$232,7$ Mo/s
MPICH-Madeleine (GM)	$11,4\ \mu$s	235 Mo/s
SCI-MPICH	$4,3\ \mu$s	$209,8$ Mo/s
MPICH-Madeleine (SCI)	$7,6\ \mu$s	$242,5$ Mo/s

Les comparaisons que nous avons effectuées permettent d'affirmer le caractère générique de notre implémentation de MPI : nos cibles sont les grappes de grappes hétérogènes, mais une utilisation en environnement plus «classique» avec des grappes homogènes de machines multi-processeurs donne d'excellents résultats. En effet, nous nous situons à des niveaux de performances équivalents et souvent supérieurs à ceux des implémentations concurrentes

et optimisées pour une technologie particulière. Les expériences conduites à l'aide de tests réels comme HPL ou plus basiques le montrent.

De plus, les applications utilisant notre implémentation peuvent s'attendre à des gains importants en ce qui concerne les opérations non-bloquantes : cela démontre que notre approche consistant à découpler la progression des communications de celle de l'application est rationnelle.

Ainsi, le second principe qui fonde notre démarche de recherche est validé par ces résultats : l'empilement de couches logicielles n'est pas dommageable pour les performances, quand bien même l'interface de la bibliothèque de communication est générique et masque les caractéristiques des réseaux sous-jacents.

Notre conclusion est que les implémentations actuelles de MPI sont capables de fournir un haut niveau de performances en exploitant pleinement les capacités du matériel pour lequel elles sont destinées.

5.4 Évaluation sur grappes hétérogènes et grappes de grappes

Après avoir comparé notre implémentation avec la bibliothèque de communication sous-jacente et avec des solutions concurrentes multiprotocoles mais destinées à l'exploitation des grappes homogènes de machines multi-processeurs, nous abordons le cas des grappes des grappes. Nous avons sélectionné les deux systèmes qui nous paraissent à l'heure actuelle le plus à même d'être capables d'exploiter ce type de configurations : MPICH-G2 et PACX-MPI. Ce travail a été mené en collaboration avec l'équipe du Professeur Wolfgang Rehm, qui dirige le Projet Coc-Grid et toutes les mesures ont été effectuées sur la grappe Jack.

5.4.1 Comparaisons avec MPICH-G2

Afin de pouvoir réaliser des comparaisons qui aient un sens, nous avons intégré notre implémentation de MPI en tant que *Vendor MPI* pour MPICH-G2 (cf. 3.3.2.1). Ainsi nous pouvons évaluer précisement les différences entre MPICH-Madeleine et MPICH-G2 sans que le résultat soit faussé par celles existantes entre les *Vendor MPI* possible.

Nous avons utilisé la grappe Jack en configuration hétérogène avec deux nœuds pour le réseau SCI et deux autres pour le réseau Myrinet. À ce propos, le pilote GM de Madeleine n'était pas aussi optimisé que dans le cas des évaluations précédentes, la latence minimale obtenue avec MPICH-Madeleine était de 25 μs et le débit maximal était similaire à la version actuelle. Quant au réseau GigaBitEthernet, nous rappelons que la latence minimale obtenue avec MPICH-Madeleine est de 29 μs et le débit maximal de 111 Mo/s. Ces éléments permettent d'interprèter les résultats exposés dans cette section.

5.4.1.1 Comparaison avec MPICH-Madeleine

Nous avons établi des comparaisons pour les communications inter-grappes uniquement car MPICH-G2 utilisant MPICH-Madeleine pour les communications intra-grappes, il suffit de se reporter à 5.3.2.2 et 5.3.2.3 pour avoir une idée du niveau des performances de ces communications avec Myrinet et SCI. Quant à MPICH-Madeleine, nous avons utilisé un canal virtuel englobant trois canaux physiques, avec un canal pour chaque réseau présent. Les performances présentées sont donc celles obtenues avec un canal virtuel car c'est en pratique plus simple à utiliser, mais il est toujours possible de ne créer que des canaux physiques, plus performants, et de gérer au niveau applicatif les différents réseaux disponibles.

Cette configuration de MPICH-Madeleine n'utilise donc pas de passerelles et toutes les communications inter-grappes passent par le réseau GigaBitEthernet. Les figures 71 et 72 montrent donc les performances de MPICH-G2 et de MPICH-Madeleine dans un tel cas. À titre de référence, nous donnons également les performances qu'il est possible d'obtenir avec MPICH-Madeleine en utilisant la retransmission de la bibliothèque de communication Madeleine entre les réseaux Myrinet et SCI.

Les résultats obtenus montrent clairement la supériorité de MPICH-Madeleine pour les communications inter-grappes, aussi bien dans le cas de l'utilisation de GigaBitEthernet que des passerelles haut-débit de Madeleine. Nous rappelons que cela est dû à une exploitation des réseaux rapides disponibles capable d'utiliser les meilleurs protocoles disponibles alors

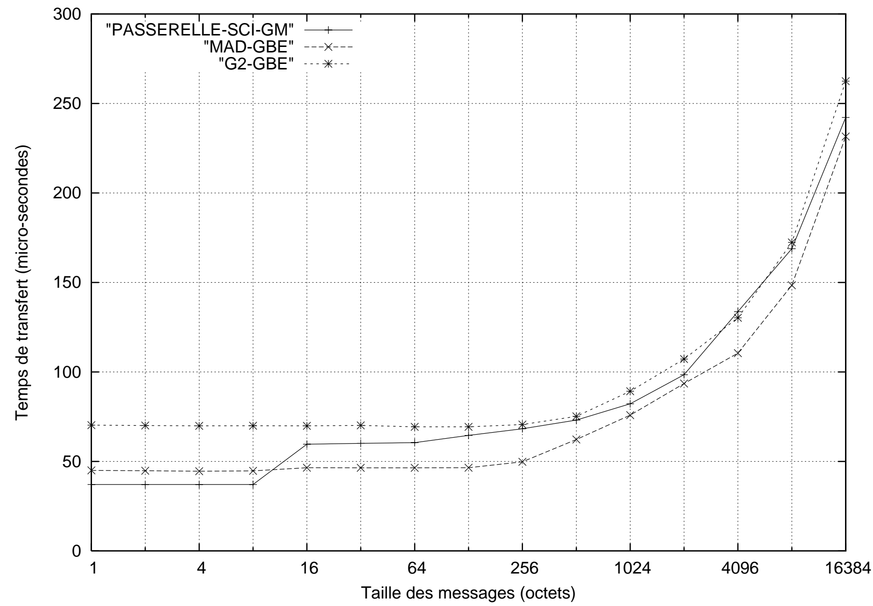


FIG. 71 – Latences des communications inter-grappes pour MPICH-G2 et MPICH-Madeleine

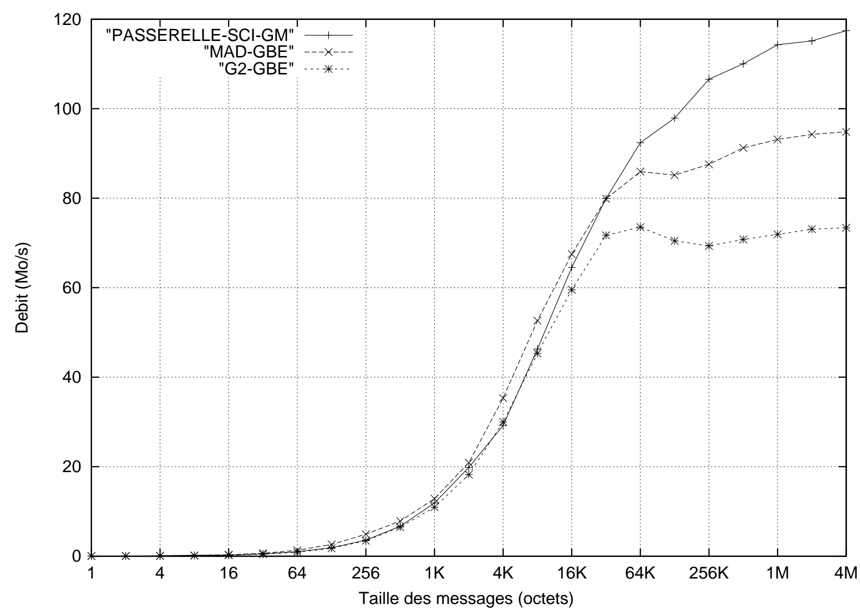


FIG. 72 – Débits des communications inter-grappes pour MPICH-G2 et MPICH-Madeleine

que les implémentations alternatives n'utilisent que TCP.

5.4.1.2 Comparaison de MPICH-Madeleine avec d'autres *Vendor MPI*

Comme nous l'avons expliqué en introduction de cette partie, nous avons intégré MPICH-Madeleine en tant que *Vendor MPI* afin d'éviter de biaiser les résultats en utilisant d'autres implémentations de MPI. Cependant, nous nous sommes posés la question de savoir comment MPICH-G2 fonctionnait avec d'autres versions de MPI. Nous nous sommes placés dans un cadre encore une fois hétérogène, en évaluant les résultats des communications inter-grappes de MPICH-G2 entre deux nœuds appartenant l'un à une grappe Myrinet et l'autre à une grappe GigaBitEthernet.

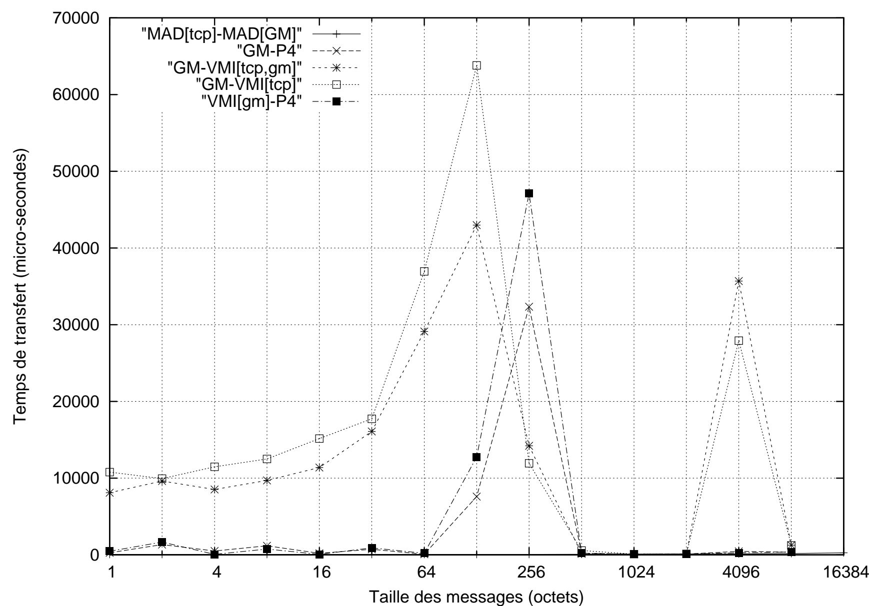


FIG. 73 – Latences de plusieurs *Vendor MPI* pour MPICH-G2

Les *Vendor MPI* utilisés sont donc outre MPICH-Madeleine, MPICH-GM, MPICH-P4 et MPICH-VMI, qui peut fonctionner au-dessus des deux réseaux considérés. Les résultats sont exposés sur les figures 73 et 74 et montrent des comportements que nous ne sommes pas encore parvenus à expliquer.

Ces résultats montrent que le succès de MPICH-G2 est très dépendant des *Vendor MPI* utilisés sur les grappes locales, ce qui est étonnant car ces implémentations de MPI n'interviennent en principe pas pour les communications inter-grappes.

5.4.2 Comparaison avec PACX-MPI

Avec MPICH-G2, nous avons évalué une solution dont le schéma pour les communications inter-grappes est direct c'est-à-dire qu'aucune retransmission n'est effectuée pour émettre un message d'une grappe vers une autre. PACX-MPI fonctionne différemment ainsi que nous avons eu l'occasion de le voir (cf. 2.1.3.3). Nous avons donc cherché à évaluer le mécanisme des passerelles de PACX-MPI et celui utilisé par MPICH-Madeleine.

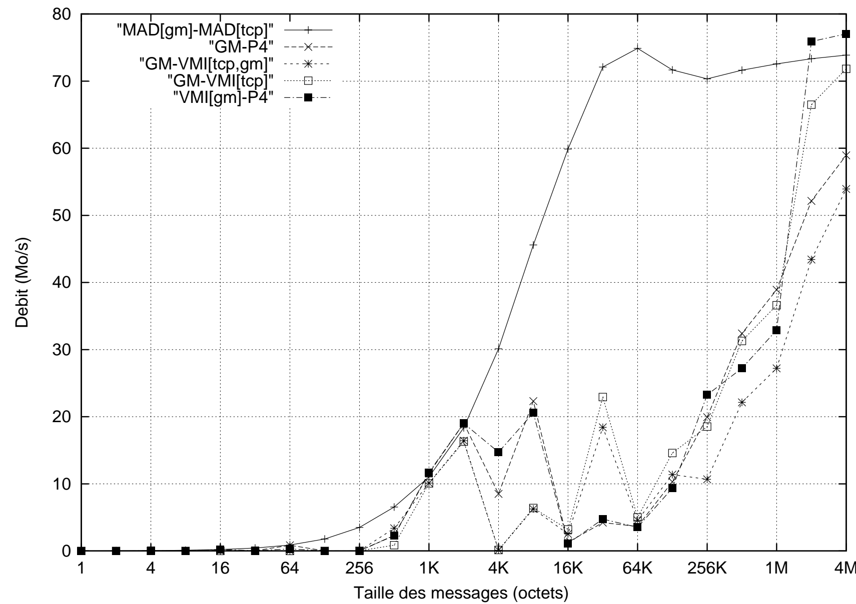


FIG. 74 – Débits de plusieurs *Vendor MPI* pour MPICH-G2

La configuration est un peu différente du cas précédent puisque nous avons considéré deux grappes SCI reliées par un réseau GigaBitEthernet différent et moins efficace que dans le cas des tests avec MPICH-G2 (ceci explique les différences de performances entre les courbes). PACX-MPI utilise SCI-MPICH comme implémentation de MPI pour l'exploitation des grappes locales.

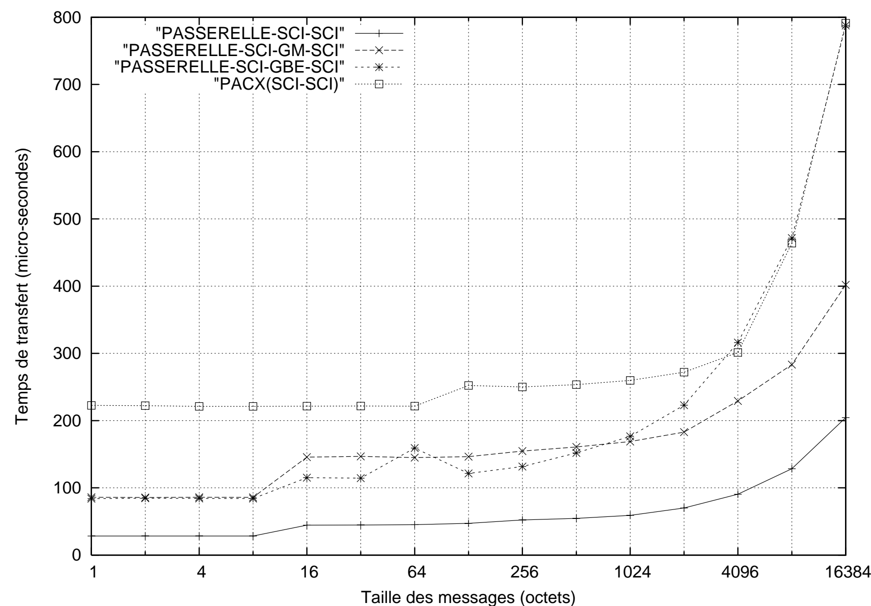


FIG. 75 – Latences des communications inter-grappes entre PACX-MPI et MPICH-Madeleine

Nous n'évaluons que les communications inter-grappes, donc la différence n'a pas d'influence sur les performances. Les figures 75 et 76 montrent les résultats obtenus. Nous avons donné les mesures pour les communications inter-grappes de PACX-MPI et pour les communications inter-grappes de MPICH-Madeleine pour les trois cas suivants :

- l'utilisation d'une unique passerelle entre les deux grappes (SCI-SCI) ;
- l'utilisation de deux passerelles avec un lien haut-débit les reliant (en l'occurence, Myrinet : SCI-GM-SCI) ;
- l'utilisation de deux passerelles avec un lien GigaBitEthernet les reliant (SCI-GBE-SCI).

Les deux derniers cas permettent à MPICH-Madeleine de simuler le comportement de PACX-MPI qui utilise forcément deux passerelles pour ses communications inter-grappes.

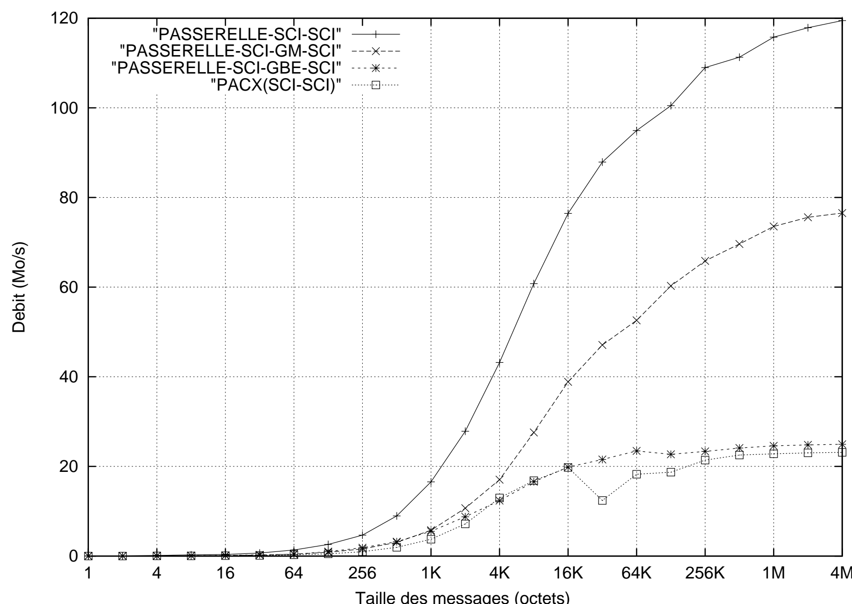


FIG. 76 – Débits des communications inter-grappes entre PACX-MPI et MPICH-Madeleine

Ces résultats nous montrent que si MPICH-Madeleine est plus performant pour les communications inter-grappes, c'est parce que nous sommes capables d'utiliser un nombre variable de passerelles et que dans le cas où nous utiliserions deux retransmissions, notre capacité d'exploitation de protocoles autres que TCP permet d'atteindre de meilleurs débits. Cela prouve également que le principe des retransmissions n'est pas foncièrement un obstacle pour l'obtention de bonnes performances et que le goulot d'étranglement est uniquement dû au protocole TCP. Un rapport technique de l'équipe CoC-Grid est consacré à cette comparaison et plus de détails sont disponibles [2] dans [BTR03].

[2]bien qu'un peu obsolètes en ce qui concerne MPICH-Madeleine

5.4.3 Conclusion

Ces expériences nous permettent d'affirmer que MPICH-Madeleine est tout-à-fait conforme au cahier des charges établi : les performances pour les communications inter-grappes sont plus efficaces que celles des système concurrents considérés dans ce chapitre. À ce titre, nous remarquons que les débits atteints lors de l'utilisation d'un unique passerelle entre deux réseaux haut-débit n'est pas totalement en accord avec ce que nous pouvions attendre étant donnée l'utilisation de pipe-lines logiciels.

MPICH-Madeleine fait également preuve de plus de souplesse, car nous avons pu à chaque fois nous adapter à la solution avec laquelle nous nous comparons en adoptant un schéma de communication (*direct* ou *avec retransmissions*) similaire.

Les expériences établies reposent pour l'instant sur des tests point-à-point mais seront complétées par des évaluations avec HPL, comme dans le cas des grappes homogènes de machines multi-processeurs.

5.5 Conclusion générale

Nous rappelons les différents points que nous avons mis en évidence avec les expériences menées : tout d'abord, MPICH-Madeleine n'introduit qu'un surcoût léger par rapport à la bibliothèque de communication sous-jacente employée (Madeleine). L'essentiel de ce surcoût est dû à la façon dont nos messages sont structurés, avec une série d'au plus deux empaquetages/dépaquetages. Cependant cette organisation, ainsi que nous l'avons vu au chapitre 4, nous est dictée par le mode de fonctionnement de Madeleine. Notre exploitation de cette couche de communication est donc très satisfaisante.

Dans le cas de grappes homogènes de machines multi-processeurs – configuration impliquant un support multiprotocole pour MPI – notre implémentation montre un excellent comportement et offre des performances égales voires supérieures à celles des systèmes concurrents, qui sont uniquement destinés à l'exploitation d'un type de réseau particulier.

Nos objectifs de portabilité des performances sont donc remplis puisqu'en ce qui concerne les plate-formes de type grappes de grappes hétérogènes, les résultats nous sont favorables.

Ainsi, MPICH-Madeleine, qui est la mise en œuvre de l'architecture proposée au chapitre 3, valide tous les choix effectués et les principes retenus dans le cadre de cette thèse. Nous montrons bien que des performances élevées sont encore possibles même dans le cas de configurations complexes et difficiles à maîtriser et qui nécessitent des outils riches en fonctionnalités.

Conclusion et perspectives

Nous donnons nos conclusions sur les travaux de recherches menés durant cette thèse et indiquons les perspectives qui se présentent pour la continuation.

1 Conclusion

1.1 Cadre de travail

Ces travaux se situent dans le contexte du calcul haute-performance et ont été menés dans le cadre du projet INRIA RUNTIME. Ce projet a pour objectif l'étude, la conception et la réalisation d'une nouvelle génération de supports d'exécution destinés à servir de fondation pour l'implémentation d'intergiciels parallèles de haut niveau. Les plate-formes matérielles visées sont de différentes natures : grappes homogènes de machines multiprocesseurs ou NUMA, grappes de grappes hétérogènes ou encore grappes de grande taille. Les systèmes développés par l'équipe RUNTIME offrent donc des services aux couches logicielles de niveau supérieur pour permettre une exploitation optimale du matériel sous-jacent.

Dans une perspective de portabilité des performances, deux axes de recherches font l'objet d'une attention toute particulière : le multithreading et les communications. En effet, ces aspects sont prépondérants pour l'exploitation efficace du matériel et l'obtention des hautes performances requises par les applications de calcul intensif. La difficulté majeure réside dans la nature des plate-formes considérées qui exhibent un caractère hiérarchique et hétérogène et dont la maîtrise est ardue à obtenir au niveau applicatif.

L'objectif de cette thèse était donc d'étudier la façon dont pouvaient être prises en compte cette nature ainsi que les exigences des applications, en particulier au sein du standard actuel de programmation parallèle par passage de messages : *Message Passing Interface*. MPI constitue une cible de choix pour les développements logiciels effectués dans RUNTIME et permet d'atteindre un public plus large, peu enclin à utiliser des outils non standard.

Cette problématique du support des configurations hiérarchiques et hétérogènes ainsi que de la gestion dynamique des processus dans MPI est très sensible et focalise l'attention de nombre d'équipes de recherche à travers le monde. Nous nous plaçons donc dans un contexte très mouvant car les technologies évoluent rapidement et la compétition au niveau international est omniprésente.

Dans un premier temps, nous résumons notre contribution dans ce domaine puis nous donnons des éléments quant à l'évalution et l'utilisation de notre travail.

1.2 Contribution

Notre contribution se situe à deux niveaux : nous avons d'une part proposé une architecture non consensuelle et l'avons mise en œuvre pratiquement, ce qui a débouché sur le logiciel MPICH-Madeleine.

1.2.1 Une proposition d'architecture logicielle originale

La première partie de notre contribution est la proposition d'architecture que nous avons formulée. Le défi majeur que nous cherchions à relever était d'aboutir sur un outil de programmation riche et flexible permettant une exploitation optimale des multiples configurations matérielles possibles. En particulier, nous voulions que les différents réseaux haut-débit disponibles soient effectivement utilisés.

La définition de notre architecture a impliqué de devoir repenser la façon dont la progression des communications est orchestrée dans les implémentations actuelles de MPI et nous avons donc axé notre démarche autour des éléments suivants :

- l'introduction d'un moteur de progression des communications fondé sur des processus légers ;
- l'utilisation d'une interface de bas-niveau générique, offrant un haut niveau de performances ainsi que des services indispensables pour leur conservation dans des environnements complexes comme ceux des grappes de grappes hétérogènes.

Ceci nous a permis d'encapsuler à bas-niveau certains aspects liés à l'hétérogénéité comme la gestion de la scrutation ou encore le routage et la retransmission des messages d'un réseau à un autre.

Les progrès et avantages sont nombreux car l'utilisation des processus légers pour notre moteur permet d'obtenir une progression des communications découplée de l'exécution de l'application en plus de substantiels gains de réactivité.

Les décisions prises au niveau architectonique n'étaient pas triviales car de nombreuses expériences passées ont illustré la difficulté d'obtenir de bons résultats dans un contexte où communications et processus légers cohabitent au sein du même système.

1.2.2 Une implémentation efficace sur de nombreuses plate-formes

Cette architecture a été mise en œuvre avec succès pour donner naissance à une nouvelle implémentation du standard MPI : MPICH-Madeleine. Nous avons utilisé comme base de départ le logiciel MPICH et l'avons profondément modifié pour y intégrer notre architecture. Les aspects liés à l'introduction des processus légers ont fait l'objet d'une attention toute particulière, si bien que notre implémentation est l'une des rares permettant l'utilisation de ces outils au niveau applicatif.

Les réseaux haut-débit actuellement supportés sont GigaBitEthernet, Myrinet et Scalable Coherent Interface et nous avons collaboré activement au développement de Madeleine avec Olivier AUMAGE afin que cette dernière soit optimisée en vue de son utilisation au sein de MPICH. Les communications par mémoire partagée sont également disponibles pour optimiser l'utilisation des nœuds multi-processeurs.

Les différents protocoles de communication utilisent un moteur de progression des communications intelligent qui s'appuye sur la bibiothèque de processus légers Marcel. Nous avons travaillé également avec Vincent DANJEAN sur l'ordonnancement de ces processus afin d'améliorer les performances des communications en environnement hétérogène.

Les objets de base de Madeleine que nous utilisons sont les canaux de communication et nous les avons étendus pour les rendre extensibles, obtenant par cette occasion une implémentation de MPI avec une gestion dynamique des processus pour répondre aux exigences des applications comme le *computational steering*. Nous avons finalement étendu l'interface de MPI avec les nouvelles fonctionnalités introduites.

Cette implémentation constitue donc une plate-forme de validation des idées mises en place dans les divers éléments logiciels développés dans RUNTIME.

1.3 Évalution et utilisation

Les expériences que nous avons menées et les comparaisons établies avec de nombreuses implémentations de MPI concurrentes montrent que notre approche est bien fondée et justifient notre démarche de recherche. Les performances obtenues sur des grappes homogènes de machines multi-processeurs ainsi que sur des grappes de grappes hétérogènes sont de tout premier plan. Notre logiciel est librement disponible et ses sources sont téléchargeables à l'URL suivant :

`http://dept-info.labri.fr/~mercier/mpi.html.`

MPICH-Madeleine a été évalué par une équipe de recherche étrangère indépendante, celle du projet CoC-Grid, dirigée par le Professeur Wolfgang Rehm. Son équipe cherche à mettre au point des solutions logicielles pour l'exploitation des configurations telles que les grappes de grappes et les grilles de calcul. MPICH-Madeleine est l'une des trois solutions retenues avec PACX-MPI et MPICH-G2 et faisant l'objet d'une évaluation par cette équipe. Les résultats préliminaires sont très encourageants et montrent que MPICH-Madeleine est un compétiteur sérieux pour les solutions existantes.

MPICH-Madeleine est de plus utilisé au sein du projet PARIS, dirigé par Thierry PRIOL, comme l'implémentation de MPI pour le logiciel PADICOTM. Les performances sont une fois encore très satisfaisantes et permettent à PADICOTM une exploitation optimale du matériel sous-jacent. Notre logiciel remplit donc le rôle pour lequel il a été conçu.

Enfin, l'équipe de développement de MPICH, qui met au point une implémentation du standard MPI-2 nous a invités à ARGONNE NATIONAL LABORATORY pour faire partager notre expérience acquise dans le domaine de la gestion du multi-protocole dans MPICH. Nos idées y ont reçu un accueil très favorable et le travail entrepris au cours de cette thèse se poursuivra – avec pour cible MPICH2 – dans le cadre de notre post-doctorat au sein de cette équipe américaine.

2 Perspectives

Notre travail a permis de reconsidérer le problème de la gestion de la hiérarchie et de l'hétérogénéité dans MPI, tout en intégrant les aspects de la gestion dynamique des processus. MPICH-Madeleine permet donc d'écrire de nouveaux types d'applications destinées à des configurations très diverses.

2.1 Perspectives à court terme

D'intéressantes suites à ce travail subsistent néanmoins. D'une part, certains aspects sont envisageables à court terme, comme l'optimisation des transferts des types de données non-contigus. Les opérations de communications unilatérales (*one-sided operations*) font également partie des fonctionnalités que nous pouvons mettre en place simplement, car notre architecture est totalement compatible avec le principe de ces communications. Les opérations collectives pourraient également faire l'objet d'un travail d'optimisation de façon à tirer parti le mieux possible de l'organisation de la configuration matérielle sous-jacente.

2.2 Extensions nécessaires

Mais d'autres extensions de notre travail nous semblent plus indispensables. L'implantation de la plate-forme GRID5000 sure le territoire national offre des perspectives excitantes d'évaluation pour notre logiciel. Cependant, afin d'avoir un fonctionnement réellement adéquat avec cette plate-forme, les aspects liés à la dynamicité doivent être complétés, avec notamment un lancement véritablement décentralisé des différents «morceaux» d'application que nous voudrions voir collaborer. Cela devrait permettre de plus d'évaluer les aptitudes d'un logiciel comme MPI pour les communications à plus large échelle. Un second point concerne le routage puisque nous aimerions mettre en place une politique de retransmission plus souple et totalement compatible avec la gestion dynamique des processus. Une idée serait de faire cohabiter des politiques distinctes au sein d'une même application en associant une politique de routage à chacun des communicateurs que nous créeons.

2.3 NUMA, multithreading et communications

Nous avons mis l'accent sur l'adaptabilité de notre logiciel vis-à-vis des différentes plateformes matérielles rencontrées et avons brièvement parlé des machines de type NUMA. Ces configurations nous apportent des perspectives très intéressantes car nous nous posons la question de l'exploitation de grappes de telles machines, qui n'est pas triviale. En effet, comment faudrait-il répartir les processus sur les différents nœuds ? L'exploitation de ces architectures passe-t'elle par l'utilisation de la mémoire partagée ou bien allons nous nous acheminer vers une solution où les processeurs seraient occupés par les différents processus légers du moteur de communication ? Le placement de ces entités est très problématique car l'ordonnanceur actuel est incapable de connaître le rôle des différents processus légers. Il est donc parfois amené à prendre des décisions à l'encontre de ce que nous souhaiterions.

Bibliographie

[ABD+00] Olivier Aumage, Luc Bougé, Alexandre Denis, Lionel Eyraud, Jean-François Méhaut, Guillaume Mercier, Raymond Namyst et Loïc Prylli, *A portable and efficient communication library for high-performance cluster computing*, IEEE International Conf. on Cluster Computing (Cluster 2000) (Chemnitz), novembre 2000, pp. 78–87.

[ABD+02] Olivier Aumage, Luc Bougé, Alexandre Denis, Lionel Eyraud, Jean-François Méhaut, Guillaume Mercier, Raymond Namyst et Loïc Prylli, *High performance computing on heterogeneous clusters with the Madeleine II communication library*, Cluster Computing **5** (2002), 43–54, Special Issue on the Cluster 2000 Conference.

[ABEN02] Olivier Aumage, Luc Bougé, Lionel Eyraud et Raymond Namyst, *Calcul réparti à grande échelle*, ch. Communications efficaces au sein d'une interconnexion hétérogène de grappes : Exemple de mise en oeuvre dans la bibliothèque Madeleine, Hermès Science Paris, 2002, ISBN 2-7462-0472-X.

[ADF+01] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Nicholas T. Karonis, Matei Ripeanu, Edward Seidel et Brian Toonen, *Supporting efficient execution in heterogeneous distributed computing environments with cactus and globus*, Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM) (Denver, Colorado), ACM Press, 2001, p. 52.

[AEN01] Olivier Aumage, Lionel Eyraud et Raymond Namyst, *Efficient Inter-Device Data-Forwarding in the Madeleine Communication Library*, Proc. 15th Intl. Parallel and Distributed Processing Symposium, 10th Heterogeneous Computing Workshop (HCW 2001) (San Francisco), Held in conjunction with IPDPS 2001, 2001, Extended proceedings in electronic form only, p. 86.

[AF99] Adnan M. Agbaria et Roy Friedman, *Starfish : Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations*, 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

[AL03] Ravi Reddy Alexey Lastovetsky, *HMPI : Towards a Message-Passing Library for Heterogeneous Networks of Computers*, Proceedings of the International Parallel and Distributed Processing Symposium, 2003, pp. 102 – 117.

[AM03] Olivier Aumage et Guillaume Mercier, *MPICH/MadIII : a Cluster of Clusters Enabled MPI Implementation*, Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003) (Tokyo), Held in conjunction with IEEE Computer Society and ACM, mai 2003, pp. 26–35.

[AMN01] Olivier Aumage, Guillaume Mercier et Raymond Namyst, *MPICH/Madeleine : a True Multi-Protocol MPI for High-Performance Networks*, Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001) (San Francisco), Held

in conjunction with IEEE Computer Society and ACM, Avril 2001, Extended proceedings in electronic form only., p. 51.

[Aum02a] Olivier Aumage, *Heterogeneous multi-cluster networking with the Madeleine III communication library*, Proc. 16th Intl. Parallel and Distributed Processing Symposium, 11th Heterogeneous Computing Workshop (HCW 2002) (Fort Lauderdale), Held in conjunction with IPDPS 2002, Avril 2002, 12 pages. Extended proceedings in electronic form only.

[Aum02b] Olivier Aumage, *Madeleine : une interface de communication performante et portable pour exploiter les interconnexions hétérogènes de grappes.*, Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, Septembre 2002, 154 pp.

[BBC+02] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cécile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Neri et Anton Selikhov, *MPICH-V : Toward a Scalable Fault Tolerant MPI for Volatile Nodes*, Proceedings of the 2002 ACM/IEEE conference on Supercomputing (Baltimore, Maryland), IEEE Computer Society Press, 2002, pp. 1–18.

[BCH+03] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier et Frédéric Magniette, *MPICH-V2 : a Fault-Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging*, Proceedings of the 2003 ACM/IEEE conference on Supercomputing (Phoenix, Arizona), IEEE Computer Society Press, Novembre 2003.

[BSC+01] Rajanikanth Batchu, Anthony Skjellum, Zhenqian Cui, Murali Beddhu, Jothi P. Neelamegam, Yoginder Dandass et Manoj Apte, *MPI/FT(TM) : Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing*, Proceedings of the 1st International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2001, p. 26.

[BTR03] Daniel Balkanski, Mario Trams et Wolfgang Rehm, *Heterogeneous Computing with MPICH-Madeleine and PACX-MPI ; a Critical Comparison*, Proceedings of the 3rd IEEE International Conference on Cluster Computing (CLUSTER'03), 2003, `http://www-user.tu-chemnitz.de/~danib/cluster-benchmarks/index.html`.

[CC97] G. Chiola et G. Ciaccio, *GAMMA : a Low-cost Network of Workstations Based on Active Messages*, Proceedings of the 5th EUROMICRO workshop on Parallel and Distributed Processing (PDP'97) (Londres), janvier 1997.

[CE00] Franck Cappello et Daniel Etiemble, *MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks*, Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM) (Dallas, Texas, United States), IEEE Computer Society, 2000, p. 12.

[CFP+01] Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie et Leonid Gurvits, *Using Multirail Networks in High-Performance Clusters*, Proceedings of the 3rd IEEE International Conference on Cluster Computing, IEEE Computer Society, 2001, pp. 15–24.

[CHAa] *ChaMPIon/Pro*, `http://www.spscicomp.org/ScicomP8/Presentations/ChaMPIonPro.ppt/`.

[CHAb] *CHARM++*, `http://charm.cs.uiuc.edu/research/charm/`.

[CoC] *Clusters of Clusters-Grid Project*, `http://www.tu-chemnitz.de/informatik/RA/cocgrid/`.

[Con] *Condor*, `http://www.cs.wisc.edu/condor/`.

[Dan00] Vincent Danjean, *LinuxActivations : un support système performant pour les applications de calcul multithreads*, Actes des Rencontres francophones du parallélisme (RenPar 12) (LIB, Univ. Besançon), 2000, pp. 87–92.

[Dan04] Vincent Danjean, *De la réactivité des threads*, Thèse de doctorat, spécialité informatique, École normale supérieure de Lyon, 46, allée d'Italie, 69364 Lyon cedex 07, France, Décembre 2004.

[DAT] *Direct Access Transport*, `http://www.datcollaborative.org/`.

[Dem97] Erik D. Demaine, *A Threads-Only MPI Implementation for the Development of Parallel Programs*, Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97) (Winnipeg, Manitoba, Canada), juillet 1997, pp. 153–163.

[Den03] Alexandre Denis, *Contribution à la conception d'une plate-forme haute performance d'intégration d'exécutifs communicants pour la programmation des grilles de calcul*, Thèse de doctorat, spécialité informatique, Université de Rennes 1, Décembre 2003, 180 pp.

[DHLO+96] Jack Dongarra, Steven Huss-Lederman, Steve Otto, Marc Snir et David Walker, *MPI : The Complete Reference*, The MIT Press, 1996.

[DM98] L. Dagum et R. Menon, *OpenMP : An industry-based API for shared-memory programming*, IEEE Computational Science and Engineering **5** (1998), no. 1, 46–55.

[DN03] Vincent Danjean et Raymond Namyst, *Controling Kernel Scheduling from User Space : an Approach to Enhancing Applications' Reactivity to I/O Events*, Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03) (Hyderabad, India), Lect. Notes in Comp. Science, vol. 2913, Held in conjunction with IEEE Computer Society and ACM, Springer-Verlag, 2003, 10 pages, pp. 490–499.

[dSK99] B. de Supinski et N. Karonis, *Accurately Measuring MPI Broadcasts in a Computational Grid*, Proc. 8th IEEE Symp. on High Performance Distributed Computing (Redondo Beach, Californie), août 1999, pp. 29–37.

[ED96] Graham E.Fagg et Jack J. Dongarra, *PVMPI : An Integration of the PVM and MPI Systems*, Tech. report, 1996.

[ED97] Graham E.Fagg et Jack J. Dongarra, *Heterogeneous MPI Application Interoperation and Process Management under PVMPI*, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 4th European PVM/MPI Users' Group Meeting (Craovie, Pologne), Lecture Notes in Computer Science, vol. 1332, Springer, novembre 1997, pp. 91–98.

[FBD01] Graham E. Fagg, Antonin Bukovsky et Jack Dongarra, *Fault Tolerant MPI for the HARNESS Meta-computing System*, Proceedings of the International Conference on Computational Sciences-Part I, Springer-Verlag, 2001, pp. 355–366.

[FD00] Graham E. Fagg et Jack Dongarra, *FT-MPI : Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World*, Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, 2000, pp. 346–353.

[FGG^{+}98] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thruvathukal et S. Tuecke, *Wide-Area Implementation of the Message Passing Interface*, Parallel Computing, vol. 24, 1998, pp. 1735–1749.

[FGT] Ian Foster, Jonathan Geisler et Steven Tuecke, *MPI on the I-WAY : A Wide-Area, Multimethod Implementation of the Message Passing Interface*, Tech. report, Argonne National Laboratory.

[FK98] Ian Foster et Nicholas T. Karonis, *A grid-enabled MPI : message passing in heterogeneous distributed computing systems*, Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM) (San Jose, Californie), IEEE Computer Society, 1998, pp. 1–11.

[GCC99] F. García, A. Calderón et J. Carretero, *MiMPI : A Multithread-Safe Implementation of MPI*, Recent Advances in PVM and MPI. 6th PVM/MPI European User's Group Meeting (Barcelone), Lecture Notes in Computer Science, vol. 1697, Springer-Verlag, septembre 1999, pp. 207–214.

[GKP98] G.A. Geist, J.A. Kohl et P.M. Papadopoulos, *PVM and MPI : a Comparison of Features*, Tech. report, septembre 1998.

[GL94] W. Gropp et E. Lusk, *The MPI communication library : its design and a portable implementation*, Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi (1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA), IEEE Computer Society Press, 1994, pp. 160–165.

[GLDA96] W. Gropp, E. Lusk, N. Doss et A.Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing **22** (1996), no. 6, 789–828.

[GM] *Glen's Message*, `http://www.myricom.com/scs/index.html`.

[GPT99] Patrick Geoffray, Loïc Prylli et Bernard Tourancheau, *BIP-SMP : High Performance Message Passing over a Cluster of Commodity SMP's*, Supercomputing (SC'99), novembre 1999.

[GRBK98] Edgar Gabriel, Michael Resch, Thomas Beisel et Rainer Keller, *Distributed Computing in a Heterogeneous Computing Environment*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (Vassil Alexandrov et Jack Dongarra, eds.), Lecture Notes in Computer Sciences, Springer, 1998, pp. 180–188.

[HH98] Parry Husbands et James C. Hoe, *MPI-StarT : delivering network performance to numerical applications*, Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM) (San Jose, Californie), IEEE Computer Society, 1998, pp. 1–15.

[HPL] *Linpack*, `http://www.netlib.org/benchmark/hpl/`.

[HR99] Hermann Hellwagner et Alexander Reinefeld (eds.), *SCI : Scalable Coherent Interface, Architecture and Software for High-Performance Computer Clusters*, Lecture Notes in Computer Science, vol. 1734, Springer, 1999.

[IMK^{+}03] Yutaka Ishikawa, Motohiko Matsuda, Tomohiro Kudoh, Hiroshi Tezuka et Satoshi Sekiguchi, *GridMPI : The Design of a Latency-aware MPI Communication Libary*, Tech. report, Cluster Computing Research Center, 2003, *(NB : cet article n'est disponible qu'en langue japonaise)*.

[IMP00] IMPI Steering Committee, *IMPI - Interoperable Message Passing Interface*, janvier 2000.

[ITKT00] Toshiyuki Imamura, Yuichi Tsujita, Hiroshi Koide et Hiroshi Takemiya, *An Architecture of stampi : MPI Library on a Cluster of Parallel Computers*, Proceedings of EuroPVM/MPI2000, Lecture Notes in Computer Science, vol. 1908, Springer-Verlag, 2000, pp. 200–207.

[KTF03] N. Karonis, B. Toonen et I. Foster, *MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface*, Journal of Parallel and Distributed Computing, vol. 63, mai 2003, pp. 551–563.

[Lac01] Sébastien Lacour, *MPICH-G2 collective operations : performance evaluation, optimizations*, Internship report, Magistère d'informatique et modélisation (MIM), ENS Lyon, MCS Division, Argonne Natl. Labs, USA, septembre 2001.

[LAM] *Los Alamos MPI*, `http://public.lanl.gov/lampi/`.

[LBK02] Orion Lawlor, Milind Bhandarkar et L. V. Kale, *Adaptative MPI*, 2002, `http://charm.cs.uiuc.edu/papers/AmpiSC02.ps`.

[LCW+03] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff et D. K. Panda, *Performance Comparison of MPI implementations over Infiniband, Myrinet and Quadrics*, Proceedings of the SuperComputing Conference (SC), 2003.

[LG] Ewing Lusk et William Gropp, *MPICH Working Note : the implementation of the second generation ADI*, Tech. report, Argonne National Laboratory.

[LG94] Ewing Lusk et Willam Gropp, *Dynamic Process Management in an MPI Setting*, Tech. report, 1994.

[LG96] Ewing Lusk et William Gropp, *MPICH Working Note : The Second-Generation ADI for the MPICH Implementation of MPI*, Tech. report, Argonne National Laboratory, 1996.

[LG98] Ewing Lusk et Willam Gropp, *Why PVM and MPI Are Completely Different*, Tech. report, septembre 1998.

[LG02] Ewing Lusk et Willam Gropp, *Fault Tolerance in MPI Programs*, Tech. report, 2002.

[LJW+04] Jiuxing Liu, Weihang Jiang, Pete Wyckoff, Dhabalewar K. Panda, David Ashton, Darius Buntinas, William Gropp et Brian Toonen, *Design and implementation of MPICH2 over Infiniband with RDMA Support*, Proceedings of the 4th International Parallel and Distributed Processing Symposium (IPDPS 04), 2004.

[LNLE00] Soulla Louca, Neophytos Neophytou, Arianos Lachanas et Paraskevas Evripidou, *MPI-FT : Portable Fault Tolerance Scheme for MPI*, Parallel Processing letters, vol. 10, Scientific Publishing Company, 2000, pp. 371–382.

[Mer00] Guillaume Mercier, *MPICH/Madeleine : un support efficace pour l'hétérogénéité des réseaux dans MPI*, Rapport de stage de DEA DEA2000-01, LIP, ENS Lyon, Lyon, France, 2000, DEA d'informatique fondamentale, Univ. Claude Bernard, Lyon 1. Disponible en anglais comme [AMN01].

[MPIa] *MPI-Connect*, `http://icl.cs.utk.edu/projects/mpi-connect/`.

[MPIb] *MPICH-GM*, `http://www.myri.com/scs/`.

[MVA] *MVAPICH*, `http://nowlab.cis.ohio-state.edu/projects/mpi-iba/`.

[MYR] *Myrinet*, `http://www.myricom.com`.

[NM95] Raymond Namyst et Jean-François Méhaut, *PM2 : Parallel multithreaded machine. a computing environment for distributed architectures*, Parallel Computing (ParCo '95), Elsevier Science Publishers, Septembre 1995, pp. 279–285.

[NOW] *YAMPII*, `http://now.cs.berkeley.edu/`.

[OHT+96] Francis B. O'Carroll, Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa et Satoshi Matsuoka, *Implementing MPI in a High-Performance, Multithreaded Language MPC++*, Tech. report, Real World Computing Partnership, 1996.

[OPE] *Open MPI*, `http://www.open-mpi.org/`.

[PB98] Sundeep Prakash et Rajive L. Bagrodia, *MPI-SIM : using parallel simulation to evaluate MPI programs*, Proceedings of the 30th conference on Winter simulation (Washington, D.C., United States), IEEE Computer Society Press, 1998, pp. 467–474.

[PBH99] Aske Plaat, Henri E. Bal et Rutger F. H. Hofman, *Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects*, Proceedings of the The Fifth International Symposium on High Performance Computer Architecture, IEEE Computer Society, 1999, pp. 244–253.

[PD96] Boris Protopopov et Rossen Dimitrov, *Implementing high-performance multi-device (MPI) for clusters of workstations interconnected with gigabit LAN's*, Tech. report, Mississippi State University, 1996.

[PKC97] Scott Pakin, Vijay Karamcheti et Andrew A. Chien, *Fast Messages : Efficient, Portable Communication for Workstation Clusters and MPPs*, IEEE Parallel Distrib. Technol. **5** (1997), no. 2, 60–73.

[PP] Scott Pakin et Avneesh Pant, *VMI 2.0 : A Dynamically Reconfigurable Messaging Layer for Availability, Usability and Management.*

[Pro96] Boris Protopopov, *Concurrency, multi-threading and message passing*, Master thesis, Mississippi State University, décembre 1996.

[PS98] Boris Protopopov et Anthony Skjellum, *A multi-threaded message passing interface (MPI) architecture : performance and program issues*, Tech. report, Mississippi State University, 1998.

[PSB03] Martin Poeppe, Silke Schuch et Thomas Bemmerl, *A Message Passing Interface Library for Inhomogeneous Coupled Clusters*, Proceedings of ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop for Communication Architecture in Clusters(CAC 03) (Nice, France), avril 2003, `http://www.lfbs.rwth-aachen.de/papers/papers_pdf/poeppe_metampich.pdf`.

[PSH96] Boris Protopopov, Anthony Skjellum et Shane Hebert, *A Thread Taxonomy for (MPI)*, Proceedings of the Second MPI Developpers Conference (Notre Dame, Indiana), IEEE Computer Society Press, 1996, pp. 50–57.

[PT98] Loïc Prylli et Bernard Tourancheau, *BIP : a new protocol designed for high performance networking on myrinet*, Parallel and Distributed Processing, IPPS/SPDP'98, Lecture Notes in Computer Science, vol. 1388, Springer-Verlag, avril 1998, pp. 472–485.

[PTW99] Loïc Prylli, Bernard Tourancheau et Roland Westrelin, *The design for a high-performance MPI implementation on the Myrinet network*, PVM/MPI'99, Lecture Notes in Computer Science, vol. 1697, Springer, 1999, pp. 223–230.

[Rab98] Rolf Rabenseifner, *MPI-GLUE : Interoperable high-performance MPI combining different vendor's MPI worlds*, Tech. report, avril 1998.

[Rad97] Thomas Radke, *More Message Passing Performance with the Multithreaded MPICH Device*, Tech. report, University of Tachnology Chemnitz, 1997.

[SCA] *MPIConnect*, `http://www.scali.com`.

[SCI] *SCI-MPICH*, `http://www.lfbs.rwth-aachen.de/~joachim/SCI-MPICH/`.

[Squ03] Jeffrey M. Squyres, *A Component Architecture for LAM/MPI*, Proceedings of the 10th EuroPVM/MPI Users Group conference, Lect. Notes in Comp. Science, ACM Press, 2003, pp. 1–9.

[STA] *Start-x*, `http://www.ece.cmu.edu/~jhoe/StarT-X/`.

[Ste96] Georg Stellner, *CoCheck : Checkpointing and Process Migration for MPI*, Proceedings of the 10th International Parallel Processing Symposium (IPPS'96) (Honolulu, Hawaï), IEEE CS Press, Avril 1996, pp. 526–531.

[TIYT03] Yuichi Tsujita, Toshiyuki Imamura, Nobuhiro Yamagashi et Hiroshi Takemiya, *MPI-2 Support in Heterogenous Computing Environment Using an SCore Cluster System*, Proceedings of ISPA 2003, Lecture Notes in Computer Science, vol. 2745, Springer-Verlag, 2003, pp. 139–144.

[TOP] *Top 500*, `http://www.top500.org`.

[Tra02] Mario Trams, *Feasibility of PACX-MPI for use in a Cluster-of-Clusters Environment*, Tech. Report RA-TR-2002-03, Université de Chemnitz, Département d'informatique, Chemnitz, Allemagne, 2002.

[TSH+00] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada et Yutaka Ishikawa, *PMv2 : a High Performance Communication Middleware for Heterogeneous Network Environments*, Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM) (Dallas, Texas, United States), IEEE Computer Society, 2000, pp. 52–67.

[TY01] Hong Tang et Tao Yang, *Optimizing threaded MPI execution on SMP clusters*, Proceedings of the 15th international conference on Supercomputing (Sorrento, Italy), ACM Press, 2001, pp. 381–392.

[vCGS92] T. von Eicken, D. Culler, S. Goldstein et K. Schauser, *Active Messages : a mechanism for integrated communication and computation.*, Proc. 19th Int'l Symposium on Computer Architecture, mai 1992.

[VSRC95] Paula L. Vaughan, Anthony Skjellum, Donna S. Reese et Fei Chen Cheng, *Migrating from PVM to MPI, part I : the Unify system*, 5th Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginie) (IEEE Computer Society Technical Committee on Computer Architecture, ed.), IEEE Computer Society Press, Février 1995, pp. 488–495.

[vV98] T. von Eicken et W. Vogels, *Evolution of the Virtual Interface Architecture*, IEEE Computer **31** (1998), no. 11, 61–68.

[YAM] *YAMPII*, `http://www.il.is.s.u-tokyo.ac.jp/yampii/`.

Communications à hautes performances portables en environnements hiérarchiques, hétérogènes et dynamiques

Résumé : Cette thèse a pour cadre les communications dans les machines para llèles dans une optique de calcul haute-performance. Les évolutions du matériel ont rendu nécessaire les adaptations des logiciels destinés à exploiter les machines parallèles. En effet, les architectures de type 'grappes" sont maintenant très répandues et l'apparition des grilles de calcul complique encore plus la situation car l'obtention des hautes performances passe par une exploitation des différents réseaux rapides disponibles et une prise en compte de la hiérarchie intrinsèque des configurations considérées. Au niveau applicatif, de nouvelles exigences émergent comme la dynamicité. Or, ces aspects sont trop souvent partiellement traités, en particulier dans les implémentations du standard de programmation par passage de messages MPI. Les solutions existantes se concentrent sur la hiérarchie et l'hétérogénéité ou la dynamicité, exceptionnellement les deux. En ce qui concerne les premiers aspects, des simplifications conduisent à une exploitation suboptimale du matériel potentiellement disponible.
Nous avons analysé des implémentations existantes de MPI et avons proposé une architecture répondant aux besoins formulés. Cette architecture repose sur une forte interaction entre communications et processus légers et son cœur est constitué par un moteur de progression des communications qui permet d'améliorer substantiellement les mécanismes existants. Les deux éléments logiciels fondamentaux sont une bibliothèque de processus légers (Marcel) ainsi qu'une couche générique de communication (Madeleine). L'implémentation de cette architecture a débouché sur le logiciel MPICH-Madeleine, utilisé ou évalué par plusieurs équipes et projets de recherche en France comme à l'étranger. L'évalution des performances (comparaisons avec Madeleine, mesures des opérations point-à-point, noyaux applicatifs) menée avec plusieurs réseaux haut-débit sur des grappes homogènes de machines multi-processeurs et les comparaisons avec MPICH-G2 ou PACX-MPI en environnement hétérogène démontrent que MPICH-Madeleine atteint des résultats de niveau similaire voire supérieur à ceux d'implémentations spécialisées de MPI.

Mots-clef : Grappes de PC, MPI, réseaux rapides, hiérarchie, hétérogénéité, dynamicité, haute-performance

High-Performance Portable Communication in Hierarchical, Heterogeneous and Dynamical Environments

Abstract : This thesis targets communication within parallel computers with an emphasis on high-performance computing. The software exploiting parallel computers had to adapt to their evolutions. Indeed, architectures such as PC clusters are now widespread and the emergence of grids tends to add new levels of complexity since high-performance can be obtained through exploitating the different high-speed networks available as well as taking into account the inherent hierarchy of the configurations. And as far as applications are concerned, new functionalities are also required, such as dynamicity. Those aspects are far too often neglected or partially tackled in existing implementations of the message passing standard, that is, MPI. Current solutions do focus on hierarchy and heterogeneity or on dynamicity, rarely both and regarding the first aspects, some simplifications do not lead to a full exploitation of the underlying hardware.
We have analyzed existing MPI implementations and have proposed an architecture that answers the needs we pointed out. This architecture relies on a strong interaction between threads and communication and its core is build above a progression engine that improves existing mechanisms. The two key elements used are a user-level thread library (Marcel) and generic communication library (Madeleine). The implementation of this architecture, MPICH-MAdeleine, is used or evaluated by several research groups, both french and foreign. The performance assessment carried out with several high-speed networks in both homogenous and heterogenous environments shows that MPICH-Madeleine's performance level is equal or superior to that of the software it challenges.

Keywords : PC Clusters, MPI, high-speed networks, hierarchy, heterogeneity, dynamicity, high-performance