Évolutions du passage de messages face aux défis de la gestion des topologies matérielles hiérarchiques.

Guillaume Mercier

Mémoire
présenté par

Guillaume Mercier

Maître de conférence – Bordeaux INP/ENSEIRB-Matméca
Laboratoire Bordelais de Recherche en Informatique
Équipe-projet commune TADaaM

en vue de l'obtention du diplôme d'

Habilitation à Diriger des Recherches
de l'Université de Bordeaux
Spécialité : Informatique

Évolutions du passage de messages face aux défis de la gestion des topologies matérielles hiérarchiques.

Soutenue le : 4 Décembre 2019

Devant le jury composé de :

Président :	Denis Barthou	Professeur des Universités à Bordeaux-INP
Rapporteurs :	Christine Morin	Directrice de Recherche à Inria
	Christian Perez	Directeur de Recherche à Inria
Examinateurs :	Yutaka Ishikawa	Responsable de l'équipe "System Software Research" au RIKEN
	Jesper Larsson Träff	Professeur des Universités à Tü Wien
Garant :	Emmanuel Jeannot	Directeur de Recherche à Inria

Après avis de :

Rapporteurs :	Franck Cappello	Senior Computer Scientist à Argonne National Laboratory
	Christine Morin	Directrice de Recherche à Inria
	Christian Perez	Directeur de Recherche à Inria

Pour Gabriel

Remerciements

Ainsi donc, la voilà, cette fameuse page qui sera sans doute la seule lue de ce document. J'espère donc n'oublier personne et ménager les susceptibilités.

En premier lieu, je remercie mes rapporteurs : Franck Cappello, Christine Morin et Christian Perez, qui ont très gentiment accepté de participé à l'aventure et surtout de lire ce document sans doute un peu trop long (tout comme le standard MPI lui-même finalement). Leurs commentaires pertinents auront permis d'améliorer substantiellement la qualité de ce manuscript. Je remercie ensuite les membres du jury pour leur participation et leurs commentaires ainsi que pour la soutenance qui restera un bon souvenir et tout particulièrement Jesper Larsson Träff et Ishikawa-sensei avec lequel l'aventure du parallélisme à commencé il a vingt ans (tout ceci ne nous rajeunit pas !).

Je profite également de cette occasion pour remercier une nouvelle fois Raymond Namyst pour plein de choses : les cours de système à l'ENS, la thèse, le recrutement et Runtime. Bref, tout ce qui m'a permis de démarrer dans la carrière. Je remercie également les nouveaux chefs de Storm et TADaaM, Denis Barthou et Emmanuel Jeannot, aussi bien pour les aspects professionnels que personnels.

Plus généralement, je remercie toutes les personnes qui, d'une façon ou d'une autre, ont travaillé avec moi et m'ont permis *in fine* d'écrire ce document et soutenir cette habilitation. En vrac donc : tous mes coauteurs (à travers les âges), les gens sympas d'Argonne, en particulier Darius Buntinas et David Goodell avec lesquels j'ai eu la chance de travailler sereinement et sans pression et plus récemment, je remercie Andrea Piacentini et Éric Maisonnave du CERFACS pour être des fervents défenseurs de Hsplit et pour le travail autour de Hippo.

Évidemment, je pense à tous les forçats de l'open space, passés, présents et à venir : Jérôme Clet-Ortega et François Tessier (dit le Félon, mais nous sommes quittes depuis ma mystification à Nosferatu), Cyril Bordage, Bertrand Putigny, Nicolas Denoyelle (pour avoir fait péter ParcoursSup mais surtout pour sa *gentillesse*), Paul-Antoine Arras (pour les blind tests), Christopher Haine, Clément Foyer, Terry Cojean, Sylvain Henry et Farouk Mansouri (sympathique, mais ignare en dattes et variables globales). Plus largement, je fais une spéciale dédicace à tous les doctorants qui ont le bon goût de finir ce qu'ils ont commencé.

Tout ceci n'aurait pas été possible sans un environnement de travail propice, aussi il me paraît important de remercier l'ENSEIRB-Matméca et tous mes collègues qui font de cet endroit un lieu agréable de travail. Inria n'est pas en reste, bien entendu, et j'ai une pensée émue pour toutes celles et tous ceux qui, au détour d'un couloir m'ont manifesté leur sympathie dans les moments plus difficiles aussi bien professionnellement que personnellement : Brice Goglin, mon co-bureau, qui m'a appris à reconnaître les Ardennais à l'oreille et l'existence de Woinic (aaah, le pourcentage culturel ...), et qui sera ravi d'apprendre ma bonne résolution de remanger à nouveau le midi dans le bureau, Nathalie Furmento, qui prouve quotidiennement qu'être sympathique et membre du CNRS n'est pas mutuellement exculsif, les Plafrim Boys (notamment François Rué et Julien Lelaurain), auprès desquels je m'excuse de ne pas avoir fait mention de Plafrim dans ma présentation.

Une partie non négligeable de ce document n'aurait pas pu être écrite sans des échanges nourris avec les membres du Forum MPI : qu'ils en soient vivement remerciés aujourd'hui : Julien Jaeger du CEA, Jean-Baptiste Besnard et Julien Adam de Paratools, Guillaume Papauré d'Atos/Bull mais aussi Jeff Squyres, Daniel Holmes ou encore Rolf Rabenseifner et bien entendu Martin Schulz.

J'en profite pour remercier Sergio Leone, qui m'a aidé à comprendre que le monde se divise en deux catégories, tout comme les messages dans MPI.

J'ai également une pensée pour toute ma famille que je remercie de son soutien inconditionnel et sans faille, et sans laquelle je n'aurais pas réussi cette entreprise.

Enfin, et surtout, je dédie cette modeste contribution à mon fils Gabriel, qui, sans s'en rendre toujours compte, aura été lui aussi un indéfectible soutien. S'il est trop jeune encore pour comprendre tout ceci, j'espère qu'il lira un beau jour ces quelques mots et se rendra compte que jamais père n'aura été plus fier de son fils. Puisses-tu, en grandissant, conserver cette bienveillance, cette gentillesse et ce souci d'autrui qui font de toi une extraordinaire personne. Je t'aime, tout simplement.

Résumés

Résumé

Les besoins en puissance de calcul croissent d'année en année et ce dans des domaines variés, qui ne concernent plus uniquement les applications scientifiques, même si ces dernières restent de grosses consommatrices. Les seules machines capables de fournir cette puissance sont à l'heure actuelle les machines parallèles (supercalculateurs et grappes), ce qui implique que les applications doivent être parallélisées pour en tirer parti. Cette parallélisation n'est pas triviale et des paradigmes et environnements de programmation adaptés sont de rigueur. L'un des modèles de programmation le plus répandu depuis environ vingt-cinq ans est le passage de messages et il existe un standard, appelé MPI (pour *Message Passing Interface*) qui permet de garantir à la fois performances et portabilité. Cependant, les architectures matérielles évoluent depuis une quinzaine d'années et les grappes de nœuds multiprocesseurs avec une hiérarchie mémoire complexe et exhibant des effets NUMA sont de plus en plus répandues. Afin de maintenir un niveau élevé de performances, les caractéristiques du matériel doivent de plus en plus être prises en compte au niveau applicatif, ce qui impacte négativement la portabilité. De plus, le standard MPI étant indépendant du matériel sous-jacent, il ne fournit que peu de mécanismes et d'abstractions pour exploiter de façon portable et efficace ces architectures matérielles hiérarchiques complexes.

Dans ce document, je présente mes travaux concernant l'exploitation des topologies matérielles complexes dans les applications parallèles utilisant MPI (et donc le paradigme du passage de messages), que ce soit de façon pure ou bien hybride avec l'adjonction d'un autre modèle comme le multithreading. Ces travaux s'articulent autour de trois phases successives : dans un premier temps, je montre comment le standard MPI et ses implémentations gèrent (ou pas) les architectures matérielles hiérarchiques complexes et comment, grâce à des optimisations poussées dans ces mêmes implémentations, les applications peuvent exploiter implicitement le matériel sous-jacent. Dans une deuxième phase, je montre comment la gestion de la localité dans les applications permet d'améliorer plus avant les performances, en utilisant des mécanismes connexes à MPI et appartenant à l'ensemble de l'écosystème HPC. Cependant, ces mécanismes, bien qu'efficaces, ne présentent pas toujours des garanties suffisantes de portabilité, ce qui conduit à la dernière phase où des modifications du standard MPI lui-même me semblent nécessaires pour atteindre les buts recherchés.

Abstract

The needs for computing power are increasing from year to year in various fields and not only for scientific applications, despite the fact that they remain big consumers. The only computers able to deliver such power are currently parallel computers (supercomputers and clusters alike) which means that applications need to be parallelized to take advantage of it. Such parallelization step is by no mean a trivial undertaking and programming paradigms and environments tailored for such work are mandatory. One of the most widespread programming model for the last 25 years is called message passing and a dedicated standard called *Message Passing Interface* (MPI) ensures both performance and portability. However, hardware architectures have undergone tremendous changes for the last 15 years and clusters of multiprocessor

nodes featuring a complex memory hierarchy with NUMA effects are more and more widespread. In order to sustain a high level of performance, hardware characteristics and details need to be taken into account at the application level, which impacts negatively portability. Moreover, the MPI standard being hardware-agnostic, very few mechanisms and abstractions are offered to the user so as to leverage these hierarchical and complex hardware architectures in a portable yet efficient way.

In this document, I present my research dealing with the exploitation of complex hardware topologies in parallel applications using MPI in a pure or in a hybrid fashion, that is, with the adjunct use of another programming model such as multithreading for instance. This work hinges around three successive phases : first, I show how the MPI standard and its implementations handle (or not) complex hierarchical hardware architectures and how, through sophisticated optimizations in the implementations, applications are able to implicitly exploit the underlying hardware. In a second phase, I show how locality management in applications allow to further improve performance by using mechanisms related to MPI and belonging to the HPC ecosystem. However such mechanisms, despite their efficiency, do not guarantee portability which leads to the last phase where extensions to the MPI standard itself are in my opinion necessary to reach the goals.

Table des matières

En-tête i

Remerciements v

Résumés vii

Table des matières ix

Table des figures xiii

Liste des tableaux xv

1 Introduction 1
- 1.1 Domaine des travaux : le calcul hautes-performances . . . 1
- 1.2 Positionnement et évolution des travaux . . . 4
- 1.3 Organisation du document . . . 5

2 Passage de messages et topologies matérielles complexes 7
- 2.1 *Message Passing Interface* : un standard, de multiples mises en œuvre . . . 8
  - 2.1.1 Genèse du standard *Message Passing Interface* . . . 8
  - 2.1.2 La première génération d'implémentations . . . 8
  - 2.1.3 La seconde génération d'implémentations . . . 9
- 2.2 Discussions autour des topologies matérielles . . . 9
  - 2.2.1 Évolutions des architectures matérielles . . . 9
  - 2.2.2 Topologie matérielle vs. hiérarchie . . . 12
- 2.3 Le standard MPI et les topologies matérielles . . . 14
  - 2.3.1 De l'indépendance de MPI vis-à-vis du matériel . . . 14
  - 2.3.2 Une confusion malheureuse . . . 15
  - 2.3.3 Un modèle de programmation trop simple (voire simpliste) ? . . . 15
  - 2.3.4 Cachez cette topologie que je ne saurais voir … . . . 16
  - 2.3.5 Les implémentations à la rescousse . . . 16
- 2.4 Les implémentations de MPI et les topologies matérielles . . . 17
  - 2.4.1 Différences de générations, différences d'approches . . . 17
  - 2.4.2 Cas des communications point-à-point inter-nœuds . . . 18
  - 2.4.3 Cas des communications point-à-point intra-nœud . . . 20
  - 2.4.4 Cas des communications collectives . . . 22
  - 2.4.5 Cas des topologies virtuelles . . . 23
- 2.5 Communications à hautes-performances en environnements hiérarchiques . . . 24
  - 2.5.1 Vue d'ensemble de MPICH2 et NEMESIS . . . 25
  - 2.5.2 Mise en œuvre des communications intra-nœud dans NEMESIS . . . 30
  - 2.5.3 Support des communications inter-nœuds : les module réseaux . . . 37
  - 2.5.4 Performances de MPICH2-NEMESIS . . . 42
  - 2.5.5 Bilan critique de MPICH2-NEMESIS . . . 50
- 2.6 Conclusion . . . 52

3 Vers une meilleure prise en compte de la localité dans l'écosystème HPC 55
3.1 Localité de données et placement de processus applicatifs . . . 56
3.1.1 Localité et placement : même combat ? . . . 56
3.1.2 Illustration de la problématique par l'exemple . . . 57
3.1.3 Une formalisation simple du problème . . . 62
3.1.4 Stratégies et solutions algorithmiques pour le placement . . . 63
3.1.5 Mise en œuvre pratique du placement : un état (partiel) de l'art . . . 64
3.1.6 Impact de mes premiers travaux consacrés au placement . . . 69
3.2 Le placement de processus dans MPI (et ses applications) . . . 69
3.2.1 Placement ou renumérotation de processus ? . . . 70
3.2.2 MPI et la prise en compte de la localité . . . 73
3.2.3 Un problème non trivial . . . 75
3.3 Un environnement de déploiement et d'exécution d'applications *locality-aware* . . . 77
3.3.1 Étape 1 : récupération des informations de topologie virtuelle . . . 77
3.3.2 Étape 2 : récolte des informations de topologie matérielle . . . 84
3.3.3 Étape 3 : appariement de la topologie virtuelle avec la topologie matérielle 89
3.3.4 Étape 4 : application effective du résultat de la permutation σ . . . 94
3.4 Au-delà du déploiement : localité et allocation de ressources de calcul . . . 100
3.4.1 Allocation de ressources dans les machines parallèles . . . 100
3.4.2 Choix de métrique et approche retenue. . . . 101
3.4.3 Topologies virtuelles et matérielles dans les gestionnaires de ressources . 101
3.4.4 Allocation de ressources *virtual topology-aware* . . . 103
3.4.5 Un exemple pratique . . . 104
3.5 Conclusion . . . 105

4 De nouvelles abstractions pour exploiter la localité dans les applications 107
4.1 Mécanismes disponibles et évolutions de l'écosystème MPI . . . 108
4.1.1 Topologies virtuelles et renumérotation . . . 108
4.1.2 Support du partage de l'espace d'adressage . . . 111
4.1.3 Sessions MPI et ensembles de processus . . . 113
4.1.4 Gestionnaires de processus/supports exécutifs . . . 114
4.1.5 Solutions *ad-hoc* . . . 115
4.2 Vers une exploitation de la localité avec les communicateurs MPI . . . 116
4.2.1 Représentation d'une topologie matérielle avec des communicateurs MPI 116
4.2.2 Extensions proposées du standard MPI . . . 118
4.2.3 Fonctionnement des communicateurs topologiques par l'exemple . . . 122
4.2.4 Discussion sur l'approche proposée . . . 127
4.2.5 Implémentations existantes . . . 129
4.2.6 Une solution portable pour le placement dynamique d'applications hybrides ? . . . 134
4.3 Le groupe de travail *Hardware Topologies* du MPI Forum . . . 138
4.3.1 Création d'un groupe de travail dédié aux topologie matérielles . . . 139
4.3.2 Organisation et objectifs du groupe de travail . . . 139
4.3.3 Propositions du groupe de travail . . . 140
4.4 Conclusion . . . 141

5 Conclusion générale **143**
5.1 Bilan des travaux présentés . . . 143
5.2 Visibilité et impact des travaux réalisés . . . 144
5.2.1 MPICH2-NEMESIS : une implémentation MPI de référence . . . 144
5.2.2 hwloc et sa concurrence : combien de divisions ? . . . 145
5.2.3 TREEMATCH : une référence pour le calcul de placement . . . 145
5.2.4 Le pari Hsplit/Hippo . . . 145
5.3 Perspectives . . . 146
5.3.1 Amélioration et extension des travaux actuels . . . 146
5.3.2 Poursuivre les efforts de standardisation, au sens large . . . 148
5.3.3 Repenser MPI et sa place dans le paysage du HPC (et au-delà) . . . 149

Liste des publications **151**

Bibliographie **155**

Table des figures

2.1 Topologies NUMA de plates-formes multiprocesseurs généralistes. P désigne les processeurs, M la mémoire, et I/O les bus d'entrées-sorties *(Courtoisie de Brice Goglin)*. . . . 11
2.2 Performances comparées selon le placement des processus dans une grappe Infiniband . . . 13
2.3 Architecture «classique» bimodulaire d'une implémentation MPI. . . . 17
2.4 Architecture avec module extérieur de communication pour les échanges inter-grappes. . . . 19
2.5 Architecture récursive multi-MPI. . . . 19
2.6 Architecture logicielle de MPICH2. . . . 25
2.7 Une émission/réception en mémoire partagée avec NEMESIS . . . 27
2.8 Principe de fonctionnement de NEMESIS avec trois processus. . . . 29
2.9 Algorithme *Lock-free*, utilisant les opérations *swap* atomique (SWAP) et *compare-and-swap* (CAS). . . . 31
2.10 Schéma d'un protocole de type rendez-vous. . . . 32
2.11 Architecture de MPICH2-Nemesis avec le LMT Knem. . . . 37
2.12 Pile logicielle de MPICH avec et sans *Tag Matching* . . . 41
2.13 Imbrication indésirable de protocoles rendez-vous. . . . 42
2.14 Performance des communications intra-nœuds pour différentes implémentations de MPI. . . . 44
2.15 Performances point-à-point avec le réseau Infiniband . . . 47
2.16 Performances point-à-point avec le réseau Myrinet/MX . . . 49
2.17 NAS Parallel Benchmarks sur Infiniband . . . 50
2.18 NAS Parallel Benchmarks sur MX/Myrinet . . . 51

3.1 Schéma de communication pour quatre anneaux avec quatre processus chacun (les cercles grisés sont les *leaders*). . . . 58
3.2 Comparaison de schémas de communication : kernels NAS et échange de jeton . 59
3.3 Temps de circulation du jeton en fonction de sa taille et de la politique de placement. . . . 61
3.4 Placement et renumérotation de processus . . . 71
3.5 Extrait d'un schéma de communication. . . . 80
3.6 Schémas de communication pour le noyau NAS IS (Classe D, 64 processus) . . . 80
3.7 Extrait de données d'analyse : communications inter-nœuds et total des communications . . . 81
3.8 Extrait de traces d'analyse : temps passé dans MPI et ratio calcul/communication 82
3.9 Extrait de données d'analyse : compteurs matériels . . . 83
3.10 Un processeur Opteron. . . . 84
3.11 Représentation qualitative du processeur Opteron de la figure 3.10. *Machine(i,j)* représente la vitesse des communications entre les cœurs C_i and C_j. . . . 85
3.12 Une machine quadriprocesseurs Intel Itanium 2 *Montecito* 9040 bicœurs. . . . 85
3.13 Informations de topologie matérielle sous forme graphique (mécanisme de découverte de topologie de PM^2 . . . 87

3.14 Organisation arborescente d'une topologie matérielle dans hwloc. 88
3.15 Données en entrée de l'algorithme TREEMATCH 90
3.16 Ensemble indépendant de poids minimal 92
3.17 Évolution de la matrice de communication aux différentes étapes de l'algorithme TREEMATCH. 93
3.18 Ensemble indépendant de poids minimal (profondeur 2) 94
3.19 Intégration de `MPI_Dist_graph_create` dans un code MPI pré-existant. . . . 98
3.20 Schémas de communication pour RSA 768. 99
3.21 Arbre de la topologie de 6 nœuds (2 unités physiques d'exécution chacun) avec le nœud n3 non disponible 104

4.1 Les recommandations du NERSC en cas d'utilisation d'un modèle hybride de programmation. 112
4.2 Un exemple d'application MPI avec plusieurs sessions simultanées (figure extraite de la propostion soumise au MPI Forum pour les sessions). 114
4.3 Un exemple de nœud de calcul hiérarchique. 123
4.4 Exemple d'affectation hétérogène de processus sur des unités de calcul 126
4.5 Cas d'utilisation du CERFACS 135
4.6 Temps d'exécution de l'interpolateur SCRIP. 137

Liste des tableaux

2.1 Instructions nécessaires pour envoyer/recevoir un message de 8 octets. 46
2.2 Ratio de communications réseaux des noyaux NAS en fonction de la classe et du nombre de processus (politique de placement *Round-Robin*). 48

3.1 RSA-768 : résultats pour cent itérations 99
3.2 Matrice d'affinité des 8 processus (4 paires) : les nombres correspondent à la quantité de données ou au nombre de messages échangés entre les processus applicatifs. 105

4.1 Extensions topologiques de Fujitsu pour le réseau toroïdal Tofu. 115

Chapitre 1
Introduction

Ce document présente la majorité des travaux menés après l'obtention de ma thèse, effectuée au LaBRI et soutenue en 2004 à l'université de Bordeaux. Une partie concerne mon séjour post-doctoral à l'Argonne National Laboratory (Chicago, États-Unis) jusqu'en Septembre 2006, date à laquelle j'ai été recruté Maître de Conférences à l'ENSEIRB-Matméca [1]. Ce manuscript est donc consacré à mes activités de recherche au sein de l'Équipe-Projet Commune (EPC) Runtime, puis, suite à sa recomposition, au sein de l'EPC TADaaM. La période couverte (2005 – 2019) étant assez large, j'ai essayé de remettre en perspective les travaux et recherches afin que le lecteur puisse appréhender au mieux le contexte mais surtout la pertinence des directions prises et des choix effectués. Une partie de ce qui sera présenté ici pourra sembler à l'heure actuelle presque *banal* car bon nombre d'idées et de préconisations sont désormais intégrées par les utilisateurs et les développeurs d'applications, car considérées comme *évidentes* (par exemple, le placement de processus), mais elles n'allaient pas forcément de soi au moment de leur invention ou de leur formulation. Que cela soit devenu actuellement le cas me conforte dans les directions suivies, les approches retenues et les outils réalisés et diffusés.

1.1 Domaine des travaux : le calcul hautes-performances

Les travaux présentés dans ce document (et plus généralement depuis ma thèse) se situent dans le contexte du calcul hautes-performances (*High-Performance Computing* ou HPC). Il s'agit d'un domaine extrêmement concurrentiel dans le paysage de la recherche (en informatique). Les raisons sont multiples et les considérations scientifiques ne sont pas les seules à rentrer en ligne de compte, bien que prépondérantes. En effet, s'il est clair que l'existence de machines aux performances hors normes permet de relever des défis scientifiques majeurs, elle autorise également une démonstration de force d'un point de vue politique car la puissance de calcul d'une nation lui permet de s'affirmer en tant que puissance scientifique sur la scène internationale («*HPC is part of a global race*»). C'est ainsi que sont apparus dans le paysage du calcul hautes-performances de nouveaux acteurs (Russie, Chine, Brésil, Inde) aux côtés des traditionnels États-Unis, Japon et Europe (où coexistent les stratégies individuelles de certains états-membres avec une coordination à l'échelon européen comme en attestent les programmes H2020 [2] ou PRACE [3]).

Course vers l'*exascale* et parallélisme

La préoccupation actuelle dans le monde du HPC consiste à atteindre l'*exascale*, c'est-à-dire construire une machine capable d'effectuer 10^{18} opérations flottantes par seconde [4]. Pour

1. École ayant depuis intégré le groupement de l'Institut Polytechnique de Bordeaux.
2. https://ec.europa.eu/programmes/horizon2020/en/h2020-section/high-performance-computing-hpc
3. http://www.prace-ri.eu/prace-in-a-few-words/
4. Un milliard de milliards d'opérations de calcul à virgule flottante par seconde.

atteindre cet ambitieux objectif, les machines construites possèdent une architecture dite *parallèle*, c'est-à-dire agrégeant un grand nombre de machines (ou nœuds de calcul) à l'aide d'un réseau rapide et ces nœuds de calcul sont eux-mêmes composés de multiples processeurs. Ce matériel doit bien entendu être à la pointe et délivrer des performances de tout premier plan. Il est parfois complémenté par des accélérateurs plus spécialisés qui peuvent se substituer aux processeurs génériques dans certains cas. Ces accélérateurs (comme les GPU) ont émergé depuis une douzaine d'années environ, mais les architectures hétérogènes résultant sont toutefois plus difficiles à programmer et demandent d'adapter les applications pour en tirer parti, ce qui peut poser problème pour une adoption plus large par les développeurs des applications pouvant bénéficier de ce type de matériel. De plus, la question énergétique devient un défi d'importance à relever car il est impératif d'être en capacité de fournir la puissance de calcul requise mais avec un bugdet énergétique limitant les solutions envisageables [5].

Pour quelles applications, et avec quelles conséquences ?

Les grandes capacités de calcul exhibées par de telles machines répondent en fait à des besoins applicatifs de domaines scientifiques variés (dont l'impact sociétal n'est pas négligeable) : sciences du climat et de l'environnement, énergie nucléaire, énergies renouvelables ou encore sciences des matériaux sont notamment concernées. À ces demandeurs «tradtitionnels» viennent s'ajouter de façon plus récente les applications de *Big Data* et de *Data Science* qui traitent des jeux de données de plus en plus volumineux, ce qui requiert bien évidemment d'importantes capacités de calcul. Toutes ces applications ont besoin de plus de puissance de calcul parce que les problèmes considérés sont toujours plus ambitieux ou parce que le niveau de précision demandé croît significativement. Or ces besoins ne font que s'accroître avec le temps, entraînant la course évoquée précédemment. Par conséquent, si les seules architectures capables de fournir les ressources nécessaires (i.e. la puissance de calcul) sont les machines parallèles, les applications doivent adapter leur exécution et passer d'un mode d'exécution séquentiel à un mode d'exécution parallèle, justement. Dans un tel mode, l'application de départ est découpée en «morceaux» (des tâches) qui seront exécutées simultanément sur les différents ressources de calcul de la machine, ce qui permet une réduction drastique du temps de calcul séquentiel original.

Affres de la parallélisation

Néanmoins, un tel fonctionnement pose deux problèmes : d'une part, l'opération consistant à découper l'application d'origine en un ensemble de tâches doit être à la fois réalisable et optimisée, c'est-à-dire que l'ensemble de ces tâches pouvant s'exécuter simultanément doit être le plus grand possible pour obtenir un important degré de parallélisme. Cette partie du travail (la parallélisation et son optimisation) incombe aux développeurs d'applications et aux personnes qui travaillent sur les algorithmes parallèles. D'autre part, l'application parallèle obtenue doit être écrite de telle façon que l'architecture-cible sous-jacente soit exploitée à son maximum. Ce travail est difficile car il requiert de prendre en compte plusieurs facteurs : les caractéristiques de l'application, celles du matériel considéré, mais aussi l'évolutivité des architectures parallèles. En effet, il est important que les performances des applications puissent être conservées quand bien même le matériel viendrait à changer (chose fréquente dans le domaine du HPC, où les machines deviennent obsolètes après quelques années seulement).

5. La question environnementale s'invite, ce qui n'était pas forcément le cas auparavant.

Standards et modèles de programmation parallèle

L'écriture d'une application parallèle passe tout d'abord par l'emploi d'un modèle de programmation adapté, modèle qui sera instancié sous la forme d'un support exécutif ou d'une bibliothèque. Il existe actuellement un ensemble de modèles permettant l'écriture d'une application parallèle : langages basé sur le partitionnement d'un espace d'adressage global (PGAS), graphes de tâches, passage de messages ou encore multithreading. Le choix du modèle (et *in fine* du langage/logiciel/support d'exécution) va dépendre surtout du programmeur, de l'application et dans une moindre mesure de l'architecture-cible. Or, afin de conserver un niveau satisfaisant de performances, il est était courant par le passé de devoir réécrire les applications pour exploiter à son maximum l'architecture-cible. La portabilité était donc sacrifiée sur l'autel de la performance. Cette approche entraînant des pertes de temps et des coûts importants, des standards de programmation parallèle ont été proposés et sont utilisés en pratique pour écrire les applications scientifiques. Cela permet de régler les problèmes de portabilité, mais des problèmes de performance peuvent alors apparaître. En effet, les interfaces définies sont souvent des abstractions de haut-niveau et ne permettent pas toujours de rentrer dans les détails du matériel, unique manière d'obtenir des optimisations poussées. Un des objectifs des *implémentations* de ces standards est donc de réduire cet écart entre les performances du matériel et celles délivrées au niveau applicatif. Également, ces mêmes standards doivent parfois être révisés afin de mieux appréhender la complexité du matériel et les méthodes de programmation, qui sont aussi susceptibles d'évolutions.

Place du modèle du passage de messages

Un des standards de programmation parallèle le plus utilisé s'appelle *Message Passing Interface* (MPI). Il repose sur le modèle du passage de messages où les différents processus applicatifs (les tâches vues précédemment) s'envoient de façon explicite des messages pour s'échanger des données ou des informations sur l'exécution de l'application. Le passage de message est souvent considéré comme le paradigme de programmation le plus intuitif ou naturel pour écrire des applications parallèles. Le standard MPI a été défini dans sa première version en 1994 par le Forum MPI, un organisme regroupant à la fois des industriels du HPC et des équipes universitaires travaillant dans le domaine du calcul parallèle. MPI s'est rapidement imposé comme un outil incontournable pour le développement d'applications parallèles, et les constructeurs de supercalculateurs se doivent de fournir une version de MPI avec leur matériel : ceci est considéré comme un logiciel de base du système, au même titre qu'un compilateur, un éditeur de lien, etc.

Évolutions conjointes des standards et du matériel

Les évolutions des standards de programmation (et notamment MPI, mais pas uniquement lui) tiennent compte non seulement des changements dans la pratique de développement des applications parallèles mais également des transformations profondes que connaissent les processeurs et architectures parallèles depuis bientôt une vingtaine d'années. L'augmentation du nombre de cœurs est une tendance très marquée et il est désormais courant de trouvers des nœuds de calcul avec plusieurs dizaines de cœurs et surtout une complexification substantielle de la hiérarchie mémoire de ces mêmes nœuds. Cette notion de **hiérarchie** est essentielle désormais car il faut veiller à la façon dont les processus (les tâches) de l'application vont être déployés et éxécutés sur les nœuds de calcul afin que les ressources de calcul soient exploitées au mieux pour obtenir des performances optimales. La manière dont les données sont réparties et accédées est également un sujet de préoccupation majeure. Cette problématique, connue

sous le nom de *localité* est devenue un enjeu essentiel dans le calcul hautes-performances et concerne l'écosystème HPC dans son ensemble.

1.2 Positionnement et évolution des travaux

C'est dans ce contexte que se placent mes travaux, puisque je travaille sur cette notion de hiérarchie depuis ma thèse. Dans les chapitres qui composent ce document, je détaillerai mes contributions dans le domaine du HPC pour que la localité soit mieux prise en compte dans les applications parallèles afin d'exploiter les architectures hiérarchiques complexes que l'on trouve dans toutes les machines. J'ai plus particulièrement travaillé sur le standard MPI et ses implémentations, car comme indiqué précédemment, MPI tient une place prépondérante dans le paysage du HPC. Sa place centrale a certes été régulièrement remise en cause par d'autres solutions mais à l'heure actuelle, aucune n'a été sérieusement capable de prendre le relais. Imputer cette situation à la seule inertie des développeurs d'applications qui seraient peu enclins à changer leurs habitudes constituerait selon moi une erreur de diagnostic. MPI a su résister aux changements et aux évolutions des pratiques et du matériel car il a non seulement fait la preuve de ses performances mais également parce qu'il a su se renouveler pour s'adapter et fournir les fonctionnalités permettant de réduire toujours plus cet écart de performances entre le matériel et les applications. MPI tient encore et toujours une place clef au sein de l'écosystème HPC. Mes travaux peuvent être décomposés en trois phases distinctes qui montrent des façons différentes d'adresser le problème de l'évolution de MPI pour une meilleure adéquation de ce modèle/standard/bibliothèque avec les architectures matérielles apparues depuis une quinzaine d'années.

la première phase concerne tous les travaux qui ont consisté à améliorer les *implémentations* du standard MPI. Ce faisant, le développeur n'a à fournir aucun effort de conception dans son application pour bénéficier de ces améliorations et optimisations, qui ne concernent toutefois qu'une implémentation donnée. Il est possible d'optimiser très finement une implémentation de MPI, mais cette dernière n'est pas toujours en capacité d'atteindre les meilleures performances car elle est parfois limitée dans son champ d'intervention (certaines choses ne sont pas couvertes par MPI) ;

la deuxième phase concerne les travaux menés non pas *au sein* des implémentations de MPI mais plutôt *autour* de ces implémentations, ce que j'appelle *l'écosystème MPI*. Ainsi, il est possible d'offrir des solutions pertinentes au problème de la localité et de la gestion des architectures matérielles hiérarchiques complexes qui vont complémenter les mécanismes proposés *en sus* par les implémentations. Lors de cette phase,j'ai même cherché à aller au-delà de cet écosystème MPI pour proposer des solutions concernant l'écosystème HPC plus globalement. Le problème des solutions proposées dans ce cadre réside dans leur caractère particulier, c'est-à-dire non standard. Ainsi, une application changeant d'environnement pourrait potentiellement perdre les bénéfices apportés par ces solutions, quand bien même ces dernières serait indépendantes des bibliothèques ou logiciels utilisés pour les mettre en application ;

la troisième phase consiste donc à chercher à intégrer *dans le standard MPI lui-même* des mécanismes pertinents afin que ces derniers soient disponibles quelle que soit la version de la bibliothèque MPI choisie. L'introduction dans le standard permet également de montrer aux développeurs le caractère pérenne des solutions proposées, ce qui est de nature à en accélérer l'adoption.

1.3 Organisation du document

Ce document est organisé en quatre chapitres. Le premier chapitre concerne la première phase décrite précédemment avec un première partie qui dresse une sorte de chronologie/état de l'art sur MPI et les différentes solutions proposées pour la gestion des topologies matérielles. Ma contribution au niveau des implémentations est détaillée à partir de la section 2.5 avec le système de communication NEMESIS. Je montre comment j'ai réussi à mettre au point une implémentation de MPI performante et capable d'exploiter efficacement des grappes de machines hiérarchiques complexes. NEMESIS a été un élément-clef de la pile logicielle HPC pendant de nombreuses années avec un nombre substantiel d'utilisateurs.

La deuxième phase est abordée par le chapitre 3. J'y montre les problèmes que j'ai contribués à mettre en évidence et les multiples solutions que j'ai proposées pour les résoudre. Ainsi que nous le verrons, l'écosystème HPC est concerné en bien des endroits par ces contributions. La mise en perspective de mes travaux s'appuiera une nouvelle fois sur un état de l'art qui montrera leur place et leur influence dans le domaine.

Le chapitre 4 sera quant à lui dédié à la troisème phase (la plus récente) qui m'occupe le plus à l'heure actuelle. J'y détaillerai des évolutions notables de MPI qui selon moi vont dans la bonne direction mais sont encore en-deçà de ce qu'il serait nécessaire. Je montrerai les propositions que nous défendons et que nous souhaiterions voir acceptées dans le standard MPI. Ce chapitre décrit par ailleurs le travail fourni au sein du Forum MPI afin que la gestion des architectures matérielles soit enfin examinée au niveau du standard MPI proprement dit.

Enfin, le chapitre 5 concluera ce document et donnera des perspectives aussi bien au niveau des travaux que du domaine de recherche plus globalement.

Chapitre 2
Passage de messages et topologies matérielles complexes

Dans ce chapitre, nous allons aborder les travaux effectués concernant l'amélioration des performances dans les implémentations de MPI. Tout d'abord il est important de rappeler qu'au cours des quinze dernières années, le paysage du calcul hautes-performances a considérablement évolué. Si les recherches concernant les implémentations de MPI sont toujours d'actualité encore aujourd'hui, d'autres voies sont explorées afin de réduire l'écart entre les performances obtenues pratiquement au niveau applicatif et celles offertes théoriquement par l'architecture matérielle cible. De plus, la place prépondérante occupée par MPI jusqu'à présent est remise en question par des paradigmes et solutions alternatives. Pour autant, il serait prématuré de considérer ce modèle de programmation comme obsolète car, comme nous le verrons ultérieurement (cf. chapitre 4), il demeure un élément avec lequel il faut et il faudra encore compter dans le futur. Ce n'est d'ailleurs pas la première fois que cela arrive : par le passé, MPI a été à de maintes fois donné pour «mort» mais il s'est toujours maintenu dans le paysage du HPC, par un travail constant mené au sein de la communauté et du *Forum MPI* en particulier.

Il est donc essentiel de remettre dans le contexte les travaux présentés dans ce chapitre. Au début des années 2000, MPI a moins de dix années d'existence et les implémentations sont loin d'avoir atteint le niveau de sophistication qu'elles peuvent exhiber aujourd'hui. Il était donc assez naturel de commencer par effectuer un certain travail pour les améliorer avant d'explorer d'autres pistes, comme celles que nous détaillerons dans le chapitre suivant (cf. chapitre 3). Durant mes travaux de thèse, les machines multiprocesseurs [1] commençaient à être relativement répandues et si nombre d'implémentations de MPI supportaient déjà des matériels de ce type, cette fonctionnalité avait fait plus ou moins l'objet de travaux spécifiques. À l'évidence, les machines hiérarchiques complexes (de type grappes de nœuds NUMA avec de multiples niveaux de hiérarchie mémoire) n'étaient pas encore les cibles priviligées qu'elles allaient devenir par la suite.

Dans la section 2.1, nous commencerons par effectuer quelques rappels concernant la création du standard MPI ainsi que de son implémentation qui permettront justement de remettre dans le contexte les travaux presentés. La section 2.2 sera consacrée aux architectures matérielles et plus particulièrement leur évolution. La relation entre hiérarchie et topologies matérielles sera d'ailleurs explicité. La section 2.3 détaillera les liens entre les topologies matérielles et le standard MPI. Nous ferons à cette occasion un point pour éclaircir, voire démystifier la croyance largement répandue qui voudrait que MPI soit *totalement* indépendant du matériel sous-jacent. Nous définirons ce que cette notion d'indépendance recouvre et montrerons que dès le début, MPI ne l'était pas complètement. La section 2.4 nous donnera l'occasion de faire un état de l'art des travaux concernant le support des topologies matérielles dans les implémentations de MPI. Nous montrerons comment ces dernières ont progressivement évolué afin de mieux supporter les architectures hiérarchiques complexes. Ensuite, nous détaillerons en section 2.5 une de nos contributions dans ce domaine, c'est-à-dire NEMESIS, un système de

1. bi-processeurs, plus exactement

communication qui a été un élément-clef de l'implémentation de référence MPICH pendant une décennie. Outre NEMESIS, les travaux connexes qui ont été réalisés dans ce cadre seront présentés, aussi bien au niveau intra-nœud qu'inter-nœuds. Enfin, nous dresserons un bilan et ferons une analyse de ce chapitre.

2.1 *Message Passing Interface* : un standard, de multiples mises en œuvre

2.1.1 Genèse du standard *Message Passing Interface*

Le standard *Message Passing Interface* a été adopté dans sa première mouture au mitan des années 90 [2] et a été un élément majeur dans l'adoption du parallélisme car il permettait aux utilisateurs de pouvoir écrire des applications portables sur une large gamme de machines. Cela n'était pas forcément possible auparavant car chaque constructeur de machine parallèle (supercalculateurs surtout) proposait sa propre solution dédiée à son propre matériel. Cette garantie de portabilité a été un premier élément qui a facilité l'adoption de MPI. Un deuxième élément déterminant est la façon dont MPI a été défini et standardisé. En effet, le succès est également dû au fait que l'instance de standardisation, c'est-à-dire le Forum MPI, regroupait un important panel d'acteurs majeurs du paysage du HPC : des constructeurs/ vendeurs de machines parallèles, des universités et centres de recherche travaillant sur cette thématique ou bien utilisant le parallélisme pour mener leurs travaux. Le Forum MPI a donc réuni un large spectre de compétences et de visions ce qui a permis de déboucher sur un compromis : le première version de MPI était née. Cette unité a donc rassuré les utilisateurs sur la pérénité de la proposition. Enfin, un autre élément déterminant et non des moindres a été la disponibilité quasi immédiate d'implémentations de MPI.

Ceci s'explique par le fait que lors de la création du *Forum MPI*, des équipes travaillaient sur des thématiques similaires et avaient très souvent développé leur propre support ou bibliothèque de communication. Un certain nombre de concepts, d'idées ou de fonctionnalités disponibles dans ces bibliothèques *proto-MPI* ont naturellement trouvé leur place dans ce nouveau standard et dès lors, la conversion de ces bibliothèques en implémentations de MPI s'en est trouvée facilitée. La standardisation de MPI est partie d'éléments pré-existants (Intel NX, Express, Zipcode, PARMACS, IBM EUI/CCL, PVM, P4, Occam, Linda, etc.), ce qui a eu pour résultat une disponibilité immédiate de logiciels avec lesquels il était possible de développer des applications parallèles. Une approche opposée (i.e. standardiser des fonctionnalités peu, voire pas du tout, implémentées) n'auraient sans doute pas produit le même résultat (cf. l'exemple de VIA [41]). Néanmoins, pendant quelques années, MPI va être en compétition avec *Parallel Virtual Machine* (PVM) et ce n'est que vers l'an 2000 que la base d'utilisateurs de MPI va dépasser celle de PVM et en faire un standard *de facto*.

2.1.2 La première génération d'implémentations

Il est possible de distinguer deux générations d'implémentations de MPI. Ces deux générations ne sont pas totalement correlées avec l'évolution du standard proprement dit, c'est-à-dire les transitions de MPI 1 vers MPI 2 puis vers MPI 3, et ainsi de suite. La première génération d'implémentations ont été disponibles durant une période d'une dizaine d'années environ : du milieu des années 90 au milieu des années 2000. D'un point de vue du standard, cela correspond essentiellement à MPI 1 puis MPI 2, même si le support de MPI 2 s'est révélé quelque peu problématique, notamment avec l'introduction du multithreading (e.g. le support du niveau

2. 1994 pour être précis.

`MPI_THREAD_MULTIPLE`) et de la gestion dynamique des processus. Il existait à l'époque un nombre important d'implémentations de MPI, tant libres que commerciales, développées aussi bien par des acteurs académiques qu'industriels. Il est à noter qu'à cette époque déjà, deux projets libres émergent : MPI Chameleon [130] d'un côté (MPICH en abrégé) et Local Area Multicomputer MPI (LAM/MPI) [187] de l'autre. Un certain nombre d'implémentations seront des dérivées de ces deux logiciels libres. En effet, implémenter MPI est un travail assez conséquent et il n'est donc pas déraisonnable de repartir d'une base pré-existante, d'autant plus que le nombre de fonctions dans MPI ne cesse de croître au fur et à mesure que le numéro de version augmente [3]. Les architectures en couches de ces implémentations favorisent d'ailleurs une telle démarche. Il «suffit» de remplacer les couches de plus bas-niveau par des éléments spécifiques ou propriétaires afin d'obtenir une implémentation complète des spécifications de MPI. Enfin, il a existé à cette époque (i.e. la fin des années 90) un certain nombre d'implémentations utilisant des threads plutôt que des processus UNIX classiques et se destinant à des machines à mémoire partagée. C'est le cas de ToMPI [49], de MultiThreaded Device [170] [4], de MiMPI [67] ou encore de threadMPI [191].

2.1.3 La seconde génération d'implémentations

À partir du milieu des années 2000, le nombre d'implémentations libres va considérablement chuter et deux pôles principaux vont se constituer autour de la nouvelle version de MPICH d'une part (MPICH2 [76]) et du successeur de LAM/MPI d'autre part (Open MPI [65]). En ce qui concerne Open MPI, un certain nombre de projets avec des thèmes bien précis, comme le support de la tolérance aux pannes ou bien le support de configurations multi-réseaux vont rejoindre ce consortium en y apportant de fait des fonctionnalités particulières. Les premiers articles présentant Open MPI datent de 2004 et décrivent son architecture modulaire. Du côté de MPICH, les premières publications remontent à 2002 mais le projet ne va véritablement décoller qu'à partir de 2005 avec l'attribution du *R&D100 Award*.

D'autres équipes essaieront de leur côté de créer *ex nihilo* une nouvelle implémentation de MPI, comme par exemple YAMPII [210] mais le succès sera limité (le Japon essentiellement). À noter également que le foisonnement d'implémentations basées sur des threads plutôt que sur des processus traditionnels va se tarir, avec la notable exception d'Adaptive MPI [91] et MPC [159], utilisé pour des questions de programmation hybride mais également pour des considérations d'empreinte mémoire de l'implémentation afin d'assurer un certain passage à l'échelle des applications visées. Cependant, la perspective de l'*exascale* remet ce type d'approche en selle et de nouveaux travaux émergent. C'est cette seconde génération d'implémentations qui va être confrontée au problème du support des machines hiérarchiques complexes avec la généralisation massive des processeurs multicœurs, là où la précédente n'avait eu qu'à gérer les machines SMP (biprocesseur pour l'essentiel). Nous allons détailler ce point dans les sections suivantes.

2.2 Discussions autour des topologies matérielles

2.2.1 Évolutions des architectures matérielles

Il est difficile de comprendre les évolutions qu'ont subi les implémentations de MPI sans se pencher sur les changements survenus au niveau des architectures parallèles au cours des vingt-cinq dernières années. Jusqu'au milieu des années 90, le paysage du calcul parallèle est

3. Le nombre de fonctions a quasiment doublé entre MPI 1 et MPI 2 par exemple !
4. Qui n'est pas à proprement parler une implémentation de MPI.

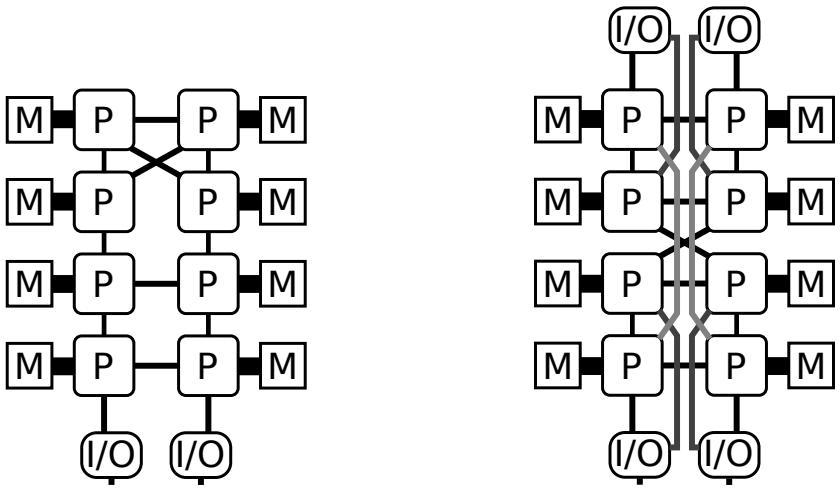
dominé par les figures de la machine massivement parallèle et du supercalculateur. Il s'agit de machines coûteuses et réservées à des laboratoires et centres de calcul de taille conséquente, et dont l'accès n'est pas forcément simple. La possibilité de créer des réseaux de stations de travail (*Network of Workstations*) avec des environnements de programmation comme PVM va changer la donne. En effet, il est désormais possible pour les laboratoires avec des moyens plus limités d'utiliser leurs stations de travail (pendant les périodes d'inactivité par exemple) pour en faire une machine parallèle.

Dans le même temps, plusieurs technologies de réseaux rapides d'interconnexion vont faire leur apparition, comme Myrinet ou *Scalable Coherent Interface* avec des couches de communications bas-niveau très performantes car situées en espace utilisateur et contournant à dessein le noyau du système d'exploitation. Ces réseaux rapides vont donc être des cibles de choix pour construire ce qui sera appelé des grappes (*clusters*). Les stations de travail vont assez rapidement être supplantées par des machines plus grand public et dont les processeurs atteignent de très bonnes performances. L'arrivée de ces *clusters* va contribuer également à démocratiser le parallélisme. La disponibilité de ces architectures va aussi permettre aux recherches consacrées aux supports exécutifs et environnements de programmation parallèle de connaître un essor important.

L'architecture de ces premiers *clusters* est donc relativement simple : ce sont des machines uniprocesseurs reliées entre elles par un réseau rapide d'interconnexion. Évidemment, la topologie physique d'un tel réseau va dépendre du nombre de machines à relier ainsi que de la technologie utilisée. Une telle organisation du matériel correspond alors très bien au modèle de programmation «plat» de MPI (c.f. section 2.3.3). La vision du matériel est relativement simple puisque les utilisateurs sont conscients de la présence d'entités distinctes (les nœuds) reliées par le réseau et possédant chacune leur propre mémoire. Comme ces machines ne disposent que d'un unique processeur, il est dès lors facile de faire l'amalgame entre un *nœud* et un *processeur*. Quant au réseau, il est rare que les utilisateurs s'en préoccupent et l'approche «boîte noire» est dans ce cas pleinement assumée. Il faut cependant noter que cette vision simple n'est pas nécessairement partagée par les gens qui implémentent MPI à l'époque. Cependant, si des travaux sont menés, il sont essentiellement destinés aux réseaux d'interconnexion des machines parallèles plutôt qu'aux grappes (e.g. [81]).

Un nouveau type de nœud de calcul se démocratise vers la fin des années 90 : les machines SMP (*Symmetric Multi Processors*). Ces machines ne possèdent plus un unique processeur mais plusieurs (deux pour commencer ...), qui sont tous reliés à la mémoire centrale via le même bus partagé. De plus, ces machines sont de type UMA (*Uniform Memory Access*), ce qui signifie que le temps pris par un processus pour accéder à ses données résidant en mémoire est le même, quel que soit le processeur sur lequel il s'exécute. Cette première évolution de l'architecture des nœuds ne va pas bouleverser les habitudes des utilisateurs de MPI car les implémentations s'en chargeront selon un principe que nous évoquerons ultérieurement en section 2.3.5 et que nous détaillerons dans ce cas précis (cf. section 2.4.3).

Dès lors, cette tendance ne va aller qu'en s'accentuant, avec une multiplication du nombre de processeurs par nœud. Elle est de plus conjugée à la remise en cause de la loi de Dennard [50], mais pas encore de celle de Moore [144]. La conséquence est l'augmentation continue du nombre de cœurs et/ou de processeurs par puce. Ainsi, la plupart des machines actuelles – ce qui inclut celles vendues au grand public – sont des (petites) machines parallèles. Au niveau des machines de calcul et des grappes, les liens entre les unités logiques d'exécution des programmes (cœurs, CPU, etc.) et la mémoire ne sont plus aussi simples que dans le cas des machines SMP. En effet, il n'est plus possible de garantir des temps d'accès uniformes à la mémoire, d'autant plus que cette dernière est physiquement répartie en plusieurs bancs disséminés dans la machine. Ce phénomène est désigné sous le nom d'effet NUMA (*Non Uniform*



(a) Nœud à 8 processeurs AMD Opteron 800 ou 8000 (2004-2010).

(b) Nœud à 8 processeurs Intel Xeon 7500 ou E7-8800 (2010-2012).

FIGURE 2.1 – Topologies NUMA de plates-formes multiprocesseurs généralistes. P désigne les processeurs, M la mémoire, et I/O les bus d'entrées-sorties *(Courtoisie de Brice Goglin)*.

Memory Access).

L'apparition des processeurs multicœurs avec effets NUMA va de concert avec une autre tendance : la complexification de la hiérarchie mémoire des machines. L'utilisation de caches pour accélérer les accès à la mémoire centrale est rentrée dans les habitudes depuis longtemps et constitue quelque chose d'assez simple dans le cas uniprocesseur et UMA. Or, les processeurs sont devenus bien plus complexes à cause de la présence de multiples niveaux de caches (L1, L2, L3, etc.). De plus, ces caches ne sont pas forcément tous partagés entre les différents cœurs du processeur. Cependant, une utilisation sophistiquée de ces ressources permet une amélioration des performances des applications. Les caches, qui faisaient déjà l'objet de préoccupations pour les bibliothèques utilisées en calcul hautes-performances (e.g. les implémentations des BLAS) sont désormais la cible d'optimisations par les applications elles-mêmes. Fort heureusement, la cohérence des caches est assurée (la plupart du temps) : on parle alors de machines ccNUMA (*cache coherent Non Uniform Memory Access*).

Enfin, cette tendance à la diversification n'est pas prête de s'arrêter car des évolutions technologiques annoncées depuis plusieurs années commencent à être disponibles au niveau des mémoires centrales. L'apparition de mémoires non volatiles (NVRAM) ou de mémoires à haut débit (*High-Bandwith Memory*), telle que la MCDRAM existante dans les processeurs KNL d'Intel®, brouille la limite entre mémoire cache et mémoire centrale. De plus, les technologies de disques durs de type *Solid State* (SSD) permettent de rajouter des niveaux intermédiaires entre mémoire centrale et mémoire de stockage. Au final, les nouvelles hiérarchies mémoires qui se profilent vont être caractérisées d'un côté par une augmentation du nombre de niveaux et surtout par des niveaux de natures potentiellement différentes avec des caractéristiques propres (en termes de performances ou cas d'utilisation). D'après une étude menée en 2017 dans le cadre du projet américain *Exascale Computing Project* [13], 79% des développeurs d'applications s'attendent à devoir gérer eux-mêmes la hiérarchie mémoire explicitement (e.g. les mouvements de données), ce qui montre bien la préoccupation que cela représente. Pour plus de précisions sur les subtilités de ces architectures et des processeurs et mémoires qui les composent, le lecteur pourra se reporter à l'habilitation à diriger les recherches de Brice GOGLIN [70]. La gestion des ces nouvelles ressources va devenir un enjeu et un défi déterminants,

aussi bien au niveau des applications parallèles que des bibliothèques qu'elles utilisent, ce qui inclut bien évidemment MPI et ses implémentations.

2.2.2 Topologie matérielle vs. hiérarchie

Dans ce document, les termes *topologies matérielles* et *hiérarchie* seront beaucoup employés. Ils ne sont néanmoins pas interchangeables et il convient par conséquent de préciser leur sens et leurs relations mutuelles.

La topologie matérielle désigne l'organisation physique des ressources de calcul d'une machine (parallèle, dans notre cas). Cela inclut par exemple :

- la topologie des réseaux d'interconnexion des différents nœuds de calcul (e.g ; *Fat Tree*, Tore, etc.) ;
- la présence éventuelle de multiples réseaux d'interconnexion ;
- l'interconnexion des *switches* dans le réseau ainsi que leur nombre ;
- l'organisation physique des nœuds de calcul, avec le nombre de cœurs, leurs interconnexions via le bus mémoire ;
- l'organisation et disposition des différents types de mémoires présentes dans un nœud (mémoire centrale, caches, etc.).

Les donneés quantitatives numériques caractérisant ces éléments, telles que la latence, le débit, la taille (capacité) des mémoires ou bien encore la fréquence des processeurs constituent des éléments d'appréciation qui peuvent être pris en compte, mais pas obligatoirement. Ainsi nous désignerons par **quantitatives** les approches qui prennent en compte ces données numériques pour proposer des solutions au problème de la gestion des topologies matérielles. Les approches qui n'utilisent que les informations de *structure* du matériel seront quant à elles qualifiées de **qualitatives**. Les deux approches ne sont pas mutuellement exclusives et il est possible de les mélanger.

La hiérarchie désigne également une forme d'organisation du matériel mais avec des critères supplémentaires précisant un peu plus ce dernier, notamment :

- l'inclusion physique de certaines ressources dans d'autres. Par exemple, un processeur appartient à un nœud de calcul, lui-même appartenant à une grappe (*cluster*) ;
- les caractéristiques des ressources permettent de la classer. Par exemple, on parle naturellement de hiérarchie mémoire, car d'un point de vue des vitesses d'accès (opérations *load*/*store*), on a :

$$\text{registres CPU} > \text{cache L1} > \text{cache L2} > \text{cache L3} > \text{mémoire centrale (RAM)} > \text{mémoire de stockage (e.g. disque dur)}$$

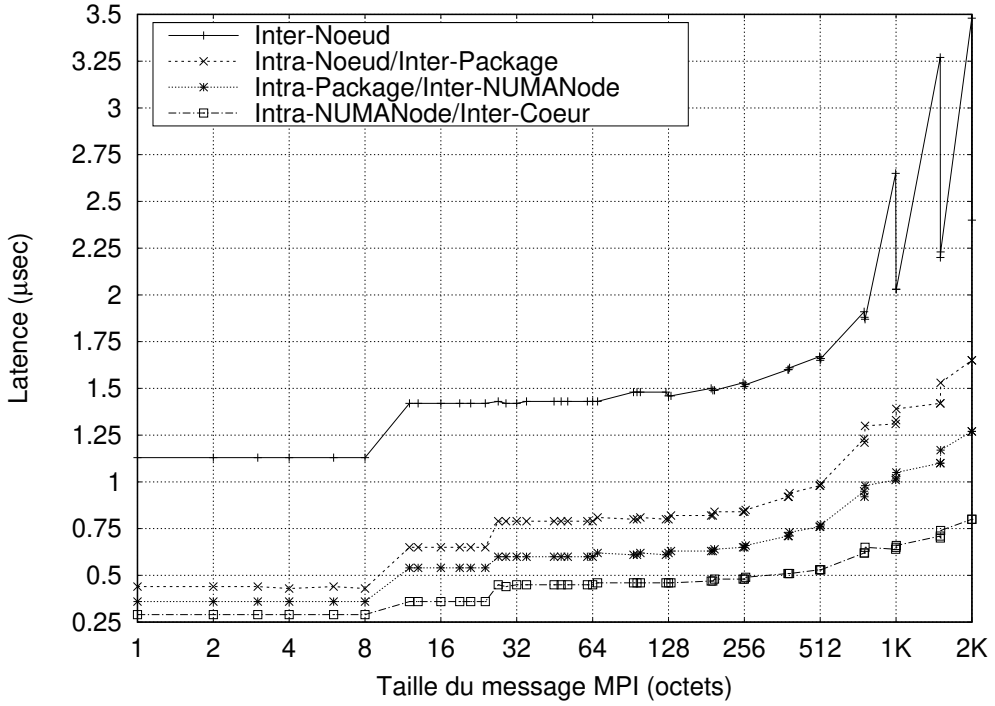
Dans le monde du calcul parallèle et du passage de messages en particulier, il est fréquent d'établir une hiérarchie concernant la vitesse de transmission d'un message entre deux processus applicatifs. Cette hiérarchie comporte le plus souvent seulement deux niveaux :

- le niveau **intra-nœud** où les communications utilisent des mécanismes de mémoire partagée ;
- le niveau **inter-nœuds** où les communications utilisent le(s) réseau(x) d'interconnexion.

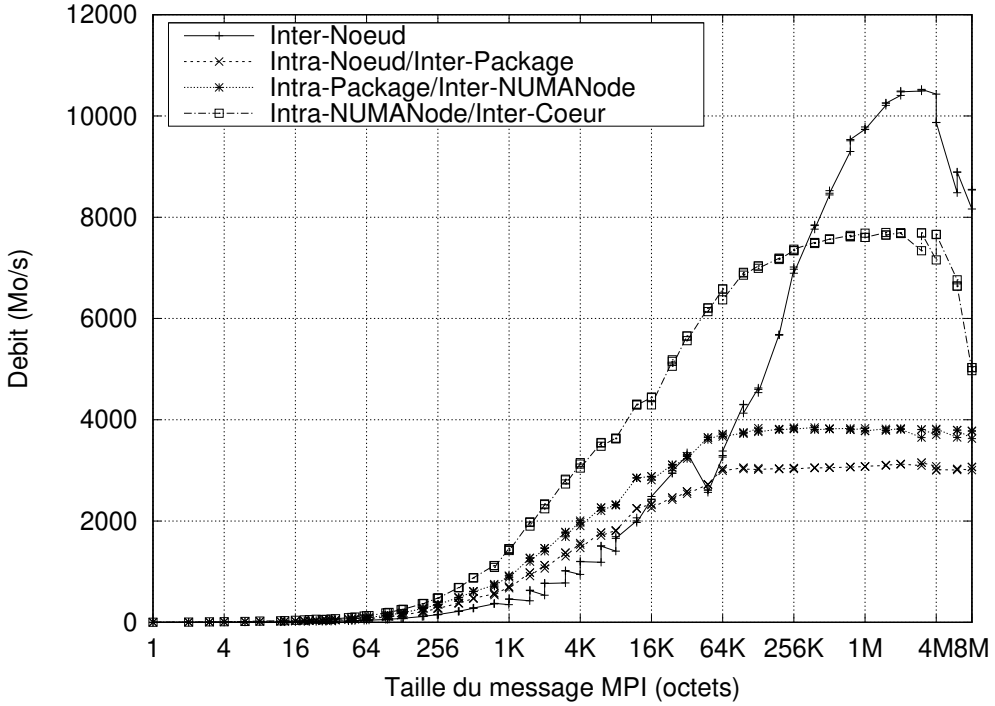
L'hypothèse le plus souvent formulée est la suivante :

$$\text{vitesse de communication}_{intra-noeud} > \text{vitesse de communication}_{inter-noeud}$$

À titre d'exemple, la figure 2.2 montre les performances à la fois dans le cas inter-nœuds et dans le cas intra-nœud dans une grappe équipée d'un réseau Infiniband. Un nœud de calcul



(a) Latence



(b) Débit

FIGURE 2.2 – Performances comparées selon le placement des processus dans une grappe Infiniband

de cette grappe possède deux *packages* [5] avec deux NUMANodes par *package*. Chaque NUMANode possède son propre cache L3 ainsi que six cœurs physiques. Chaque cœur possède ses propres caches L2 et L1. Les communications réseaux sont moins efficaces que les communications intra-nœuds et dans ce cas, les communications inter-packages sont plus lentes que les communications inter-NUMANodes, elles-mêmes plus lentes que les communications inter-cœurs (mais intra-NUMANodes) à cause du partage de cache L3. Évidemment, ces mesures point-à-point ne prennent pas en compte la congestion possible de la carte réseau ni du bus mémoire [6].

Il convient de noter que certaines topologies ne sont pas hiérarchiques. C'est par exemple le cas des réseaux en anneaux, toroïdaux, ou encore de type *Dragonfly*. Il est cependant possible d'extraire une hiérarchie dans ces cas, ce qui simplifie la vision du matériel au prix d'une évidente perte d'information. Mais la question pertinente est de savoir s'il est préférable de disposer et d'exploiter une information partielle ou bien s'il vaut mieux ne rien faire du tout ?

Ainsi que nous le verrons, de nombreuses solutions proposée par le passé comme d'autres plus actuelles ont fait le choix de s'attaquer au problème de la gestion et de l'exploitation des topologies matérielles en considérant ces dernières comme hiérarchiques. Cette vision, certes simplificatrice, n'en recouvre pas moins une réalité physique et reste très pertinente dans de nombreux cas (surtout au niveau des nœuds NUMA). Certaines approches récentes s'appuyent même sur la hiérarchie simple à deux niveaux (intra- et inter-nœuds) pour leur gestion des topologies matérielles.

Cette vision hiérarchique est très présente dans les implémentations de MPI, qui l'utilisent abondemment pour optimiser les communications dans les grappes de nœuds NUMA, mais elle est moins présente dans le standard MPI lui-même. Cependant, la problématique de l'*exascale* [180] a changé la donne et pousse MPI à introduire ce genre de concepts afin d'assurer un passage à l'échelle. La notion de voisinage commence à faire son apparition, que ce soit avec des abstractions comme les fenêtres de mémoire partagée (*Shared Memory Windows*) ou bien avec les opérations collectives de voisinage (*Neigborhood Collectives*). Bref, le but recherché est d'encourager et de respecter davantage la *localité* dans les échanges inter-processus afin de raffiner le modèle de programmation (cf.2.3.3).

2.3 Le standard MPI et les topologies matérielles

2.3.1 De l'indépendance de MPI vis-à-vis du matériel

Le succès de MPI n'est pas seulement dû à une conjonction de circonstances favorables car cela n'aurait sans doute pas permis à cette interface de se maintenir au cours de ces vingt-cinq dernières années. Certaines caractéristiques intrinsèques de MPI expliquent aussi cette réussite : MPI assure une portabilité des codes qui l'utilisent, ainsi qu'un bon niveau de performances car les différentes implémentations ont un objectif commun, à savoir la réduction de l'écart entre les performances au niveau du matériel et celles obtenus au niveau applicatif. D'autres caractéristiques comme la simplicité et la symmétrie [75] [7] peuvent également expliquer le succès rencontré par MPI.

5. Connus auparavant sour le terme de *socket* qui n'est plus utilisé par les constructeurs.

6. Ce qui peut donner l'impression que les communications inter-nœuds sont plus rapides que les communications intra-nœuds pour des messages de taille 256 kilo-octets et plus. Si l'on utilisait le réseau pour envoyer et recevoir des données entre processus sur une même machine, le bus mémoire devrait être partagé et les performances seraient moindre. Il n'y a pas de partage dans le cas point-à-point inter-nœuds d'où les performances affichées.

7. Du moins dans les premières versions du standard.

Outre ces facteurs, il en existe un autre qui revient assez régulièrement et qui concerne l'indépendance de MPI vis-à-vis du matériel sous-jacent. En effet, MPI est considéré comme étant *hardware-agnostic*, c'est-à-dire ne formulant aucune espèce d'hypothèse quant au matériel qui sera employé pour exécuter une application l'utilisant en tant que bibliothèque de communication. S'il est clair que le paradigme du passage de messages est applicable quelle que soit l'architecture matérielle, il est perçu comme particulièrement pertinent dans un contexte de mémoire physiquement distribuée (e.g. dans des grappes) mais moins adéquat pour des machines parallèles à mémoire partagée. C'est d'ailleurs ce qui explique les réticences historiques de certains constructeurs vis-à-vis de MPI ainsi que la mise en avant de modèles de programmation hybrides (MPI+X) pour les grappes de machines multicœeurs [8].

2.3.2 Une confusion malheureuse

Cependant, cette indépendance, plutôt bienvenue, fait parfois l'objet de méprises car certains l'interprètent comme une interdiction consubstantielle à MPI de fournir aux applications des éléments d'information concernant le matériel sous-jacent. Cette interprétation restrictive du standard constitue à mon sens une erreur : au contraire, il est totalement possible de pouvoir exploiter des informations matérielles au niveau applicatif via MPI, à condition de proposer et d'employer des abstractions et mécanismes idoines. De plus, cette prise en compte du matériel a toujours existé de fait, de façon plus ou moins implicite. En effet, dans le cadre d'une utilisation dans une grappe de nœuds multicœurs (cas très fréquent), les utilisateurs sont habitués [9] à répartir leurs processus applicatifs sur ces différents nœuds via un fichier appelé le *machinefile* ou *ranks file*. Ce fichier est pris en argument par la commande *mpiexec* ou *mpirun* afin que le support exécutif de la bibliothèque MPI puisse procéder au lancement, à la répartition et à l'éxecution des processus applicatifs. Concrètement, ceci est *déjà* une prise en compte de la topologie matérielle de la machine cible mais n'est pas toujours perçu comme tel car cette répartition des moyens de calcul en *nœuds* est considérée comme *naturelle*. Néanmoins, on pourra objecter que ceci est connexe à MPI et pas directement au cœur du modèle de programmation car concernant les aspects déploiement, répartition et lancement des processus de l'application.

2.3.3 Un modèle de programmation trop simple (voire simpliste) ?

Dans un univers de nœuds uniprocesseurs, on obtient alors une vision relativement «plate» de l'environnement d'exécution avec des processeurs reliés par un réseau d'interconnexion. La topologie de ce réseau n'est d'ailleurs pas nécessairement connue ni même exploitée au niveau applicatif, car, comme nous le verrons, cet aspect est délégué à l'implémentation (c.f section 2.4). Au final, le modèle de programmation plat de MPI, qui considère une application comme un ensemble de processus distribués et complètement interconnectés a totalement été intégré par les utilisateurs. Ce modèle de programmation va perdurer jusqu'à aujourd'hui, quand bien même il a été quelque peu malmené par l'arrivée et la généralisation massive des nœuds multicœurs. Ces derniers, par les multiples niveaux de hiérarchie mémoire qu'ils possèdent, introduisent une certaine complexité dans une vision qui était jusqu'alors relativement simple. Là encore, plutôt que le modèle lui-même, ce sont les implémentations qui vont se charger d'absorber ce choc technologique (c.f section 2.4). Le modèle de programmation commence cependant à connaître des évolutions bienvenues depuis ces dernières années, impulsées par

8. Constat qu'il conviendra de mitiger dans une section ultérieure (cf. 3.1.5.5).
9. Formatés pourrait-on presque dire.

la problématique de l'*exascale* (c.f le principe de voisinage introduit dans les opérations de communication collectives, par exemple).

2.3.4 Cachez cette topologie que je ne saurais voir ...

Le fait que MPI soit globalement considéré comme indépendant du matériel n'empêche en rien ce dernier d'être évoqué régulièrement dans le texte du standard lui-même. En effet, il existe dans MPI la notion de topologie virtuelle, qui permet d'organiser les différents processus MPI selon des critères applicatifs. Il est parfois possible de renuméroter ces processus selon leur rôle au sein du motif de communication de l'application (notion de *rank reordering*) et potentiellement leur localisation au sein de la machine cible. Il est également possible de créer des fenêtres de mémoire partagée afin de réduire le surcoût induit par MPI lors de communications inter-processus.

Il serait tentant alors d'introduire des critères prenant en compte non seulement ce schéma de communication mais également les caractéristiques des architectures matérielles. C'est une possibilité théorique, mais le standard ne franchit pas ce pas et laisse les implémentations faire comme bon leur semble. Ceci n'est pas la conséquence du respect de la règle d'indépendance de MPI vis-à-vis du matériel : MPI reste une interface et déborderait de ses prérogatives en dictant aux implémentations la façon de mettre en œuvre telle ou telle fonction. Cependant, ce point de vue ne fait pas l'unanimité au sein du Forum MPI. En revanche, la trop stricte interprétation de cette indépendance (c.f 2.3.2) a conduit à une situation qui n'est plus satisfaisante depuis l'avènement des machines multicœurs. Il existe cependant quelques mécanismes qui pourraient permettre dans une certaine mesure de répondre aux besoins, mais ils sont assez limités. Nous les détaillerons en section 4.1.

2.3.5 Les implémentations à la rescousse

Si la notion de topologie matérielle n'est pas clairement abordée dans le standard MPI *stricto sensu*, il n'en n'est pas de même au niveau de ses multiples implémentations, le plus souvent via la notion de *hiérarchie*. En effet, la tendance naturelle depuis l'avènement de MPI a toujours été de faire absorber les innovations et changements (chocs ?) technologiques par les implémentations et non par le standard MPI ou le modèle de programmation directement. Il a néanmoins toujours existé des exceptions à cela, quand l'impact sur les performances sont trop négatifs (e.g. le support des opérations de type *Remote Memory Access* à l'aide des communications unilatérales ou *one-sided communications*). Dans ce cas, des ajouts à MPI sont effectués, proposant de nouvelles fonctions/interfaces venant élargir la palette de choix pour résoudre un problème donné (mais au prix de cette fameuse inflation du nombre de fonctions, comme évoqué en 2.1.2).

L'avantage de cette démarche est que l'utilisateur n'a pas à gérer ces changements dans ses applications : il continue d'utiliser sa version préférée de MPI puisque le contrat concernant la minimisation de l'écart de performance entre le matériel et son application est (en théorie) respecté. Cette démarche n'est d'ailleurs pas propre à MPI puisque pendant des années, les applications ont bénéficié des gains de performance offerts par des générations successives de processeurs toujours plus rapides sans que leurs codes aient à être modifiés. En ce qui concerne le passage de messages, l'idée selon laquelle il était possible de bénéficier sans effort des changements opérés au niveau du matériel est restée valable durant un certain temps, jusqu'à aux machines multicœurs complexes. La situation est un peu différente dans le sens où l'utilisateur ne modifie pas son application, mais l'implémentation de MPI doit en revanche tenir compte de ces évolutions du matériel.

Il convient cependant de clarifier le fait que ce support des topologies matérielles est fait de façon interne dans l'implémentation de MPI proprement dite. Il s'agit donc d'une approche opaque et indirecte pour les applications, qui ne disposent pas vraiment de moyens pour exploiter les architectures matérielles à leur niveau. Nous verrons par la suite qu'il existe quelques exceptions qui dérogent à cette règle. C'est cette absence d'abstractions utilisables au niveau applicatif qui nous a poussé à proposer des extensions de l'interface MPI (voir la section 4.2) et à créer un groupe de travail dédié au sein du *Forum MPI* (c.f section 4.3).

2.4 Les implémentations de MPI et les topologies matérielles

2.4.1 Différences de générations, différences d'approches

Nous allons maintenant détailler les efforts faits au niveau des implémentations de MPI pour prendre en compte les spécificités du matériel, le plus souvent avec une approche hiérarchique de ce dernier. De façon très schématique, l'architecture d'une implémentation de MPI suit le principe décrit par la figure 2.3 avec deux sous-systèmes de communication :

— un sous-système destiné aux communications intra-nœud, basé sur de la mémoire partagée ;
— un sous-système destiné aux communications inter-nœuds et utilisant le ou les réseaux d'interconnexion disponibles.

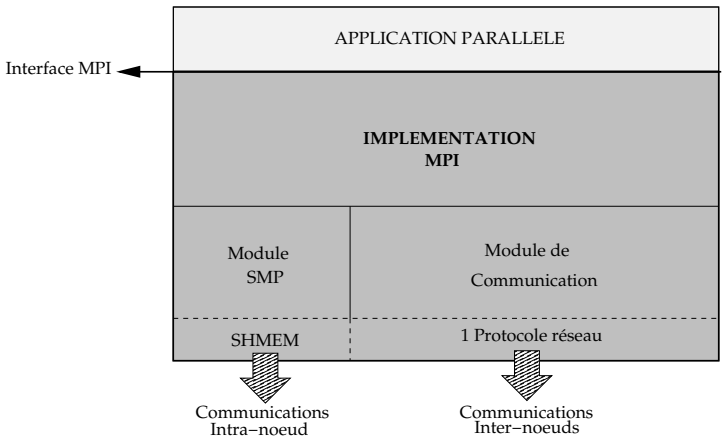


FIGURE 2.3 – Architecture «classique» bimodulaire d'une implémentation MPI.

Selon la génération d'implémentations de MPI, les efforts ne se sont pas concentrés sur les mêmes niveaux hiérarchiques de la topologie matérielle :

— **Les implémentations de première génération** comportaient à l'origine uniquement un sous-système inter-nœuds. Avec l'arrivée des machines SMP, les implémentations ont d'abord traité les communications intra-nœud avec l'interface *loopback* des cartes réseau Ethernet puis un sous-système intra-nœud dédié a été rajouté afin d'optimiser ces communications. Il a donc fallu rajouter des éléments dans des architectures logicielles qui n'avaient pas forcément été prévues pour cela à l'origine. De plus, les sous-systèmes inter-nœuds ont fait l'objet de beaucoup de travaux car le nombre important de technologies de réseaux rapides (faibles latences et hauts débits) disponibles à cette époque poussait les implémentations à suivre le mouvement. Également, le *Grid Computing* était une thématique très en vogue à cette époque (fin des années 90 – milieu des années 2000), ce qui a favorisé l'émergence de solutions MPI adaptées pour ce type d'environnements de calcul parallèle résolument orientés inter-nœuds.

— **Les implémentations de seconde génération** ont été conçues alors que les grappes de machines SMP, puis multicœurs connaissaient un essor considérable. Il était donc vital que les deux niveaux hiérarchiques soient *a minima* supportés dans les implémentations dès le départ. Le paradigme de conception a même parfois été inversé (ainsi que nous l'expliquerons dans la section 2.5 détaillant nos travaux dans ce domaine).

De façon générale, il est possible de constater que le support des machines hiérarchiques complexes a toujours fait l'objet de nombreux travaux et recherches. Nous allons maintenant détailler les différentes approches pour le support des communications point-point inter-nœuds et intra-nœud ainsi que les cas particuliers des communications collectives et des topologies virtuelles.

2.4.2 Cas des communications point-à-point inter-nœuds

La première génération d'implémentations de MPI devait être capable d'exploiter efficacement des machines massivement parallèles ou des réseaux (puis grappes) de machines uniprocesseurs. Cela passait bien évidemment par un support perfectionné du réseau interconnectant les différents processeurs. À ce niveau, il est possible de distinguer deux types d'approches :

— d'une part celle qui consistait à prendre en compte de façon très précise la topologie du réseau d'interconnexion. Cela concernait surtout des classes de machines bien particulières, pour lesquelles les constructeurs/vendeurs souhaitaient obtenir les optimisations les plus sophistiquées.

— d'autre part l'approche qui consistait à simplifier la situation en se basant sur la présence de niveaux inter- et intra-nœuds et en essayant d'exploiter au mieux leurs différences de performances existantes sans forcément se soucier des topologies physiques proprement dites (au niveau du ou des réseaux d'interconnexion).

Cette seconde approche a notamment permis **le support des architectures multi-réseaux**, c'est-à-dire les grappes possédant plusieurs réseaux d'interconnexion, les grappes de grappes et les grilles de calcul (*Grid computing*). Dans ce cas, l'implémentation ne reconnaît pas juste un niveau inter-nœuds, mais plusieurs. Les niveaux intra-nœud et inter-nœuds sont inclus dans un niveau intra-grappe et un niveau inter-grappes fait de plus son apparition. La prise en charge de ces différents niveaux peut se faire de plusieurs façons [10] :

— une unique implémentation de MPI est capable de gérer ces différents niveaux et/ou réseaux. C'est le cas de MPIConnect [147], MPI/I-WAY [61], MPICH-VMI [155], LAM/MPI [187], chaMPIon/Pro [34] ou encore MPICH/Madeleine [CI16], qui est une implémentation de MPI multiprotocoles que j'ai développée pendant ma thèse [Th1] et qui permet aux applications d'accéder aux différents réseaux physiques via des objets MPI appelés des **communicateurs**.

— une implémentation de MPI se charge du niveau intra-grappe et une autre bibliothèque de communication est chargée des échanges inter-grappes. Une surcouche logicielle est fournie qui permet malgré tout de n'avoir besoin que de MPI au niveau applicatif, ainsi que le montre la figure 2.4. Un certain nombre de projets ont suivi cette approche : Unify [202], PVMPI [58] et MPI-Connect [146] [11] utilisaient PVM pour les échanges inter-grappes. StaMPI [95], MPI-GLUE[168], PACX-MPI[64] et Interoperable MPI [68] (IMPI) utilisaient pour leur part TCP/Ethernet pour ces échanges.

— les échanges inter-grappes sont gérés par une implémentation MPI qui utilise en interne une *autre* implémentation de MPI pour gérer les échanges intra-grappes. Cette ap-

10. On trouvera dans ma thèse [Th1] plus de détails concernant toutes ces solutions ainsi qu'une classification basée selon leurs différences architecturales.

11. Á ne pas confondre avec MPIConnect !

proche, que l'on pourrait presque qualifier de «récursive» est illustrée par la figure 2.5. Cette approche a été adoptée par des projets dont les cibles étaient les grilles de calcul, notamment MPICH-G2 [104], MPI/I-WAY [61], MetaMPICH [164] ou encore GridMPI [98]. Il est à noter que cette dernière avait une approche quantitative puisque les latences des différents réseaux étaient prises en compte pour choisir le bon canal d'échange.

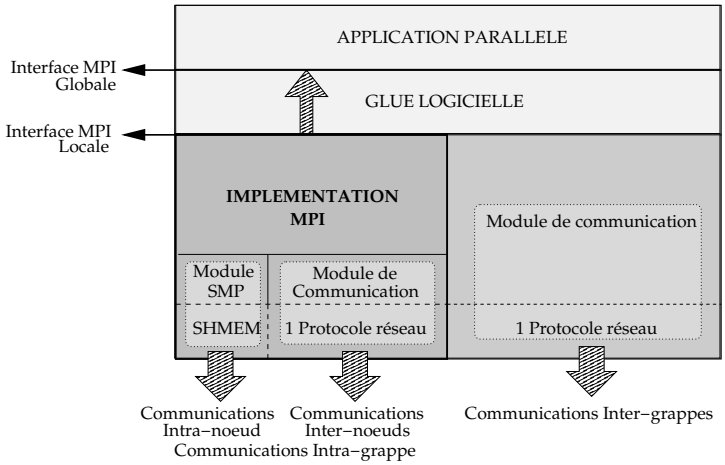


FIGURE 2.4 – Architecture avec module extérieur de communication pour les échanges inter-grappes.

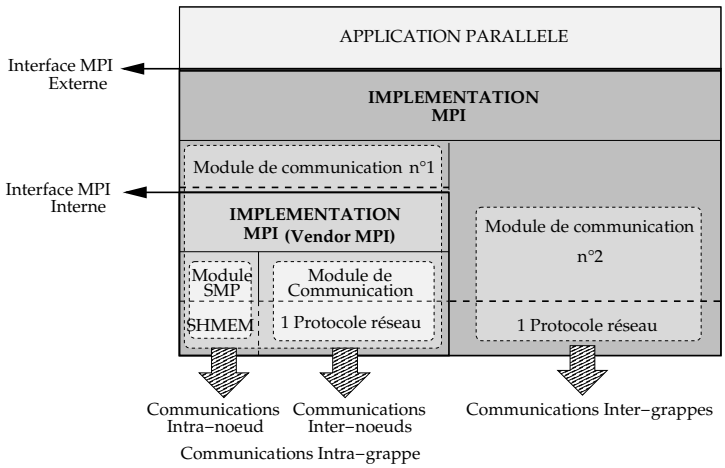


FIGURE 2.5 – Architecture récursive multi-MPI.

Le cas de MPICH-G2 mérite que l'on s'y penche de plus près car cette implémentation possède en réalité une hiérarchie à quatre niveaux :

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

MPICH-G2 formule le même type d'hypothèses que celles enoncéées en section 2.2.2, c'est-à-dire que plus un niveau est haut dans la hiérarchie, moins ses performances sont bonnes (i.e WAN-TCP $<$ LAN-TCP $<$ intra-machine TCP $<$ vendor-supplied MPI). Enfin, l'ensemble de

ces niveaux est accessible par les applications via des **communicateurs MPI dédiés**.

Les grilles de calcul ont été abordées également par d'autres travaux visant non pas les implémentations directement mais plutôt les applications parallèles MPI, notamment ceux de López *et al.* [127], ou Coti *et al.* ([43] et [11]). Là encore, l'aspect hiérarchique de la topologie est utilisé. Certains travaux ont cherché à optimiser les collectives dans ce contexte particulier, comme ceux de Gupta *et al.* [78].

Il est à noter enfin l'existence d'une approche radicalement différente : HMPI [115]. HMPI n'est pas une implémentation de MPI à proprement parler. Il s'agit d'un préprocesseur associé à une surcouche logicielle de MPI qui permet aux applications de se structurer selon une approche quantitative (c'est-à-dire que les latences et débits des différents réseaux sont prises en considération) et en utilisant là encore des **communicateurs MPI**. Cette approche évoluera par la suite en HeteroMPI [116].

Enfin, toutes les implémentations de MPI évoquées dans cette section sont de première génération et sont basées sur une approche hiérarchique simplificatrice sans rentrer finement dans le détail des topologies physiques des réseaux. Il existe néanmoins une exception car Fujitsu possède une implémentation de seconde génération avec un série d'extensions permettant aux applications de pouvoir exploiter finement la topologie de Tore à 6 dimensions de leur réseau d'interconnexion Tofu [63].

Au final, on peut constater que le problème de la gestion des topologies matérielles complexes dans MPI a été posé quasiment depuis la naissance de ce standard, mais dans un contexte différent de celui qui nous connaissons actuellement. Cependant, certaines réponses et méthodes apportées sont encore pertinentes malgré ce changement de contexte. Il convient de noter cependant que la grande majorité de ces solutions ne permet pas aux applications de contrôler et d'exploiter directement les spécificités du matériel et que souvent, c'est une approche opaque qui est choisie. Les applications peuvent donc exploiter ce matériel indirectement, parce que l'implémentation est capable de le faire. Un éventuel changement d'implémentation dans la pile logicielle peut donc potentiellement causer la perte de ce bénéfice. Ainsi que nous le verrons, cette approche, sans doute pertinente pendant un temps, connaît vite ses limites et ne permet pas d'exploiter de façon pleinement satisfaisante le potentiel du matériel. Ce reproche est formulable également en ce qui concerne le support intra-nœud que nous allons détailler maintenant.

2.4.3 Cas des communications point-à-point intra-nœud

L'arrivée des machines SMP est un changement technologique qui va être pris en charge plus ou moins bien par les implémentations MPI. Au tout début, certaines ne vont même pas changer leur architecture et vont utiliser l'interface réseau *loopback* présente dans le système d'exploitation pour gérer les communications intra-nœud. Cependant, une telle solution n'est pas satisfaisante et des solutions spécifiques vont être mises en place dans les implémentations de première génération. Un support efficace des machines SMP sera d'ailleurs un argument commercial pour certaines implémentations fournies par des vendeurs/constructeurs !

La réponse apportée est assez simple conceptuellement : comme les communications inter-nœuds sont gérées par un sous-système de communication dédié, on va rajouter un nouveau sous-système dédié aux communications intra-nœud qui vont utiliser la mémoire partagée. Cette approche, qui peut paraître simple, masque en réalité des problèmes au niveau de la gestion des communications puisque désormais, le moteur de progression [12] de ces dernières

12. Le moteur de progression est une partie cruciale d'une implémentation de MPI qui est chargé de faire progresser les communications : poster des requêtes en émission, vérifier les arrivées de messages et les files de réception, etc.

doit gérer plusieurs sources potentielles d'arrivée de messages : le réseau et la zone de mémoire partagée dédiée aux communications intra-nœud.

La plupart des implémentations «classiques» de première génération va suivre un tel schéma. En effet, pour les implémentations basées sur des threads plutôt que sur des processus traditionnels, le partage de l'espace d'adressage entre les processus MPI (des threads en réalité) fait que les communications intra-nœud sont intrinsèquement très rapides. Parmi les implémentations «classiques», on peut citer par exemple MPICH-P4 (réseaux Ethernet/TCP), MPICH-GM (réseau Myrinet avec le protocole GM), MPI-BIP (réseau Myrinet avec le protocole BIP), MPICH-PM (dérivé de MPICH avec SCore comme bibliothèque bas-niveau de communication), SCI-MPICH (version de MPICH destinée aux réseaux *Scalable Coherent Interface*), LAM/MPI ou encore MPI_StarT. On remarque que nombre d'entre-elles sont des versions dérivées de MPICH qui possédaient une architecture *a priori* évolutive permettant l'intégration de nouveaux modules des communication, mais la gestion de ces multiples modules était relativement peu sophistiquée.

C'est d'ailleurs ce constat qui m'a poussé à recréer un moteur de progression des communications totalement repensé dans MPICH/Madeleine, basé sur des threads ([CI16] et [Th1]). En ce qui concerne le support des machines SMP, c'est un élément que j'ai rajouté en fin de thèse. En effet, je me suis d'abord concentré sur les communications inter-nœuds étant donné que la cible de mon travail était à l'époque les grappes de grappes hétérogènes. Dans la conclusion de ma thèse, j'émettais l'idée que différents niveaux hiérarchiques intra-nœud pourraient être gérés via de multiples canaux de communication dédiés. La situation actuelle montre que cette solution n'est pas techniquement envisageable et que si elle l'avait été, les performances n'auraient sans doute pas été au rendez-vous.

Il y a donc peu de travaux, et c'est assez compréhensible, avant l'arrivée des machines multicœurs hiérarchiques complexes. L'accroissement du nombre de cœurs de calcul dans ces nœuds a poussé les implémentations à revoir leurs priorités et c'est un facteur non négligeable dans la transition vers la seconde génération d'implémentations de MPI. Ainsi que nous l'avons déjà écrit (cf. section 2.4.1), le paradigme de conception des implémentations de seconde génération s'est inversé avec la priorité mise sur les communications par mémoire partagée. Du côté d'Open MPI, on va trouver une BTL (*Byte Transfer Layer*) appelée SM (pour *Shared Memory*. Tandis que du côté de MPICH2, l'*Abstract Device Interface* CH version 3 voit trois *channels* dédiés : shm (*shared memory*), ssm (*socket and shared memory*) et sshm (*scalable shared memory*).

Telle était la situation de MPICH2 à mon arrivée dans ce projet à l'occasion de mon séjour post-doctoral à l'*Argonne National Laboratory*. Je vais alors travailler sur un nouveau channel pour CH3 appelé NEMESIS, qui deviendra le cœur de la gestion des communications dans MPICH2 pendant près de dix années. La section 2.5 est entièrement consacrée à cette contribution. Avec Darius BUNTINAS, nous avons d'abord effectué une étude des différentes méthodes de communication en mémoire partagée [CI12] dans les machines SMP (pas encore véritablement multicœurs) avant de proposer l'architecture de NEMESIS ([CI13], [CI14],[RI4]).

Par la suite, de nombreuses solutions vont faire leur apparition pour améliorer ce type de communications, avec en règle générale un support spécifique du noyau du système d'exploitation. On peut citer KNEM ([CI9], [145], [73]), qui prolonge les travaux effectués dans NEMESIS, mais également LiMIC ([101], [102], [33]) ou encore SMARTMAP [27], Cross Memory Attach [39] et enfin XPMEM qui est une extension du noyau. L'habilitation de Brice GOGLIN ([70], chapitre 3) sera d'une lecture précieuse pour celui ou celle cherchant plus de détails quant à ces solutions, KNEM en particulier. Il est à noter que certaines de ces solutions ont été utilisées également pour améliorer les performances des opérations unilatérales (*one-sided communications*) en intra-nœud, comme LiMIC [113] et KNEM, qui a été aussi utilisé pour les

communications collectives [134] (cf. section 2.4.4).

Enfin, il existe dans MPI (depuis la version 3.1 du standard) des fonctions pouvant améliorer les communications intra-nœud en permettant l'utilisation d'un autre paradigme de programmation, voire encore plus simplement en évitant le recours à la bibliothèque MPI. En effet, bien que les communications intra-nœud soient très rapides dans les implémentations, la bibliothèque induit un surcoût alors qu'au final on pourrait se contenter de faire des `memcpy` (des opérations de type *load* et *store*). Le fait que ce type de constructions soit désormais possible dans MPI montre bien l'enjeu que constituent les méthodes de **programmation hybrides** basées sur le passage de messages pour l'inter-nœuds et sur un autre paradigme pour l'intra-nœud. Qu'une telle chose soit désormais possible dans MPI pour les utilisateurs montre que le temps de l'approche opaque est révolue où l'application s'en remettait intégralement à une implémentation prenant toutes les décisions.

2.4.4 Cas des communications collectives

Les communications collectives forment un cas particulier pour deux raisons. D'une part, il s'agit d'une partie du standard très sensible et qui fait l'objet de nombreux travaux, et ce depuis les débuts de MPI. Il est donc évident qu'une prise en compte efficace des topologies matérielles est quelque chose d'incontournable pour ce type de communications. D'autre part, s'il est aisé de catégoriser les communications point-à-point selon le niveau hiérarchique tel que nous l'avons fait dans les sections précédentes, cela est impossible pour les communications collectives, qui ont pour vocation naturelle à franchir les limites des différents niveaux hiérarchiques et/ou topologiques.

Nous avons déjà évoqué deux approches différentes pour les communications point-à-point inter-nœuds en section 2.4.2. Cette différence d'approche peut également se retrouver dans le traitement des communications collectives dans les implémentations de MPI. La première approche, qui consiste à considérer de façon très précise la topologie du réseau d'interconnexion n'est pas utilisée, ou très peu et pour des cas bien particuliers : celui des *implémentations matérielles* de MPI. En effet, MPI étant un paradigme/bibliothèque incontournable en programmation parallèle et HPC, des constructeurs de matériel réseau se sont résolus à proposer certaines opérations collectives directement au niveau de leur matériel. C'est par exemple le cas de Quadrics avec son réseau QSNet qui proposait des opérations comme `MPI_Barrier` directement au niveau matériel, pour des performances de tout premier plan. Il est donc évident que dans de tels cas de figure, la topologie matérielle *doit* être prise en compte.

Dans la majorité des cas, c'est donc la seconde approche (on considère les écarts de performances entre les niveaux hiérarchiques) qui est la plus répandue car elle s'adresse à un éventail plus large d'architectures et est de plus bien adaptée aux grappes de machines SMP (puis multicœurs). Pour la première génération d'implémentations, il existe des travaux mais peu nombreux comparativement à la seconde génération d'implémentations, ce qui est compréhensible au regard du contexte architectural. On pourra citer notamment MagPIe [107] (qui n'est pas une implémentation de MPI mais une bibliothèque d'opérations collectives au-dessus de MPI), MPICH1 [105], MPI-StarT [94], chaMPIon/Pro [34] ou encore certains travaux de Träff [195]. D'autres travaux utilisent plus de deux niveaux hiérarchiques, comme MPICH-G2 [104] qui exploite ses quatre niveaux ou encore Matsuda *et al.* [138] qui prend en compte la latence des WAN ainsi que les différences de débits propres à chaque niveau.

À la différence de la première génération d'implémentations, la seconde va exploiter plus finement les caractéristiques du niveau intra-nœud, afin de mitiger les effets NUMA ou bien d'améliorer l'exploitation des caches partagés entre certains cœurs de calcul. L'idée est d'améliorer la localité des processus applicatifs impliqués dans ces communications collectives. On

trouve de tels travaux aussi bien dans MPICH2 ([219]) que dans ses dérivés comme MVAPICH2 ([136]) ou IntelMPI ([96] et [97]). Open MPI n'est pas en reste avec de nombreux travaux prenant en compte la topologie interne des nœuds (Graham *et al.* [74], Ma *et al.* [132], [133], [134] et [135]).

Il existe enfin des travaux plus génériques consacrés à ce sujet comme ceux de Zhang *et al.* [218] qui décompose les collectives en opération point-à-point puis optimise ces dernières grâce à un placement adéquat des processus MPI, ceux de Li *et al.* [124], de Träff et Rougier [198], ou de Pickartz *et al.* ([161] et [162]) qui s'attaquent au problème des topologies reconfigurables.

Cependant, tout comme les optimisations des opérations point-à-point, on remarquera que les améliorations fournies pour les communications collectives ne permettent pas aux applications de pouvoir manipuler les topologies physiques. Elles sont donc incapables de se structurer pour en tirer parti directement à leur niveau. On retombe sur une approche opaque avec les contrat de confiance entre l'application et l'implémentation MPI portant sur les performances fournies.

2.4.5 Cas des topologies virtuelles

Il existe enfin une catégorie de fonctions bien particulières dans MPI qui pourraient [13] faire l'objet d'optimisations très pertinentes basées sur une exploitation de la structure du matériel sous-jacent : les fonctions de topologies virtuelles dont il existe deux types dans MPI :

— les **topologies cartésiennes**, qui sont un cas particulier, mais très utilisé et qui justifie qu'une interface dédiée soit fournie par MPI. La topologie est décrite par ses dimensions et leurs périodicités.

— les **topologies génériques** qui sont décrites par un graphe quelconque. L'interface des topologies génériques a été étendue depuis MPI 2.2 en 2009 pour des questions de passage à l'échelle [88]. La notion de **graphe distribué** a été introduite et permet de pallier le problème de la description centralisée du schéma de communication applicatif. Les auteurs de cette nouvelle interface se sont notamment intéressés aux travaux que j'ai menés à l'époque concernant les politiques de placement de processus dans les nœuds multicœurs et leur influence sur les performances [CI10].

Les topologies virtuelles dans MPI servent à réorganiser les différents processus applicatifs au sein d'un nouveau communicateur MPI dans lequel leur numéro de rang sera modifié afin de mieux refléter le schéma de communication de l'application. On appelle cette opération de renumérotation le ***rank reordering*** et nous y reviendrons en détails dans la section 4.1.1. Le schéma de communication est quant à lui décrit sous forme de graphe et peut être exploité par l'implémentation à sa discrétion. Il serait donc naturel de prendre en compte la topologie matérielle sous-jacente de façon à faire une renumérotation qui mettrait en correspondance un graphe la représentant [14] avec le graphe des communications. Or, en pratique, presque rien n'a été fait dans le domaine.

Il existe cependant des travaux précurseurs très importants concernant cette optimisation des topologies virtuelles dans MPI : ceux d'Hatazaki [81] visant les machines HP Exemplar-X ou encore ceux de Träff [194] ciblant les machines NEC séries SX [15]. Ces travaux, portant sur la première génération d'implémentations, suivent la première approche décrite en section 2.4.2. Ils ne réussiront pas à enclencher un cercle vertueux, sans doute parce qu'ils étaient destinés à des classes de machines bien particulières et que le contexte ne s'y prêtait pas. Nous retrouvons

13. Le conditionnel employé ici n'est pas anodin.

14. Par exemple, des sommets pour les processeurs et des arêtes pour indiquer les coûts de communication associés entre ces derniers.

15. Travaux qui sont une référence incontournable dans le domaine.

là le problème typique de la poule et de l'œuf : les applications hésitent à utiliser les topologies virtuelles car les gains de performances ne sont pas satisfaisants. Dès lors, pourquoi optimiser une fonctionnalité qui est peu employée ?

Il faudra attendre 2011 avec les travaux de Rashti *et al.* [172] ainsi que les miens [CI6], débouchant sur une implémentation efficace de la fonction `MPI_Dist_graph_create` donnant des résultats intéressants dans le contexte des grappes de machines multicœurs hiérarchiques complexes pour que cette thématique soit de nouveau abordée. Ces travaux seront détaillés ultérieurement en section 3.3.4. Les topologies cartésiennes ont également fait l'objet de travaux en 2018 (Gropp [77] et Niethammer *et al.* [152]). À noter que les travaux de Rabenseifner *et al.* sont en discussion au sein du Forum MPI dans le groupe de travail dédié que j'ai créé (cf. section 4.3) afin de fournir une nouvelle série de fonctions prenant en compte explicitement les topologies matérielles. L'espoir n'est donc pas perdu que dans un avenir proche, les bibliothèques MPI existantes proposent des implémentations des topologies virtuelles un peu plus sophistiquées que celles actuellement disponibles.

2.5 Communications à hautes-performances en environnements hiérarchiques

Dans les sections précédentes, nous avons dressé une chronologie des implémentations de MPI et avons mis en évidence les diverses manières dont les topologies matérielles y étaient (sont) prises en charge. En effet, c'est d'abord *au sein même* de ces implémentations que les première solutions sont trouvées, afin de continuer à exploiter au maximum les performances des nouvelles générations de matériels. Cette démarche, satisfaisante de prime abord, va connaître ses limites au bout de quelques années et il faudra alors explorer d'autres voies. C'est notamment ce que nous verrons dans les chapitres 3 et 4.

Cette section va quant à elle couvrir les travaux que j'ai menés concernant les communications dans les implémentations de seconde génération destinées aux grappes de nœuds SMP et multicœurs. Ces travaux ont duré pendant plusieurs années et ont été menés sur une période couvrant les années 2005 – 2012. Ils ont pour base la mise en œuvre et l'implémentation d'un nouveau système de communication à hautes-performances dénommé NEMESIS. Ce système (qui a été au cœur de MPICH2 pendant environ une décennie) a également été fourni par un certain nombre de vendeurs dans leurs implémentations de MPI destinées à leurs architectures spécifiques : c'est notamment le cas d'Intel® ou encore de Cray. De plus NEMESIS a été le véhicule de travaux menés par d'autres personnes et notamment des membres de l'Équipe-Projet Inria Runtime plus particulièrement dans le cadre d'une Équipe-Associée Inria dont j'ai été le responsable de 2007 à 2011. Nous aurons l'occasion de revenir sur tous ces éléments ultérieurement.

Lors de mes travaux de thèse, j'avais développé une implémentation appelée MPICH/Madeleine et dans laquelle j'avais intégré vers la fin un système de communication pour gérer de façon efficace les communications par mémoire partagée. Lors de mon arrivée à l'*Argonne National Laboratory* dans l'équipe Radix reponsable des travaux concernant MPICH2, j'ai eu pour tâche, avec Darius BUNTINAS de remettre à plat l'architecture existante de cette implémentation de MPI. En effet, les performances n'en n'étaient pas satisfaisantes et il était clair que les communications intra-nœud devaient faire l'objet d'une attention particulière. Le leitmotiv était d'ailleurs : ***Shared Memory is first-class citizen***. De plus, le constructeur Cray affirmait à cette époque qu'il était impossible pour une implémentation de MPI de descendre en-dessous de 1000 instructions machine pour effectuer un échange de message court (8 octets). La tâche nous est alors incombée à Darius ainsi qu'à moi-même de démontrer le contraire.

2.5.1 Vue d'ensemble de MPICH2 et NEMESIS

2.5.1.1 Intégration de NEMESIS dans MPICH2

En 2005 [16], la transition de MPICH vers MPICH2 est effective depuis plusieurs mois mais cette implémentation n'est pas encore complètement établie dans le paysage du HPC et sa consœur de première génération est encore largement utilisée. Pourtant, des développements conséquents ont déjà eu lieu, ce qui a conduit MPICH2 à recevoir le *R&D100 Award* au cours de l'année 2004. L'architecture de MPICH2 suit globalement le même schéma que celle de MPICH et elle est donnée par la figure 2.6. Une couche logicielle appelée *Abstract Device Interface* implémente la plupart des fonctions de MPI. Au-dessus de cette couche se trouvent les structures de données générales telles que celles représentant les communicateurs, les groupes, etc. L'ADI implémente quant à elle les files de messages pour la gestion des communication : la file d'émission et les deux files de réception, une pour les messages postés/attendus (*posted receive queue*) et une seconde pour les messages inattendus (*unexpected receive queue*). La gestion bas-niveau des communications (*byte-transfer layer*) est quant à elle dévolue à un ensemble de sous-couches appelées des *channels*. La différence entre MPICH et MPICH2 réside dans l'implémentation complètement revue de ces couches : en particulier, l'ADI2 de MPICH cède la place à l'ADI3 de MPICH2 (implémentée en tant que CH3) et les couches supérieures sont totalement réécrites ainsi que la gestion des files de messages.

Cette architecture en couche permet aux implémentations dérivées de choisir les parties qu'elles souhaitent garder et celles qu'elles souhaitent réécrire. Par exemple, pour supporter une technologie réseau particulière, il est possible de ne fournir qu'une implémentation d'un *channel* dédié et de garder l'ADI3/CH3, générique. Mais il est aussi possible de réécrire complètement cette couche et de ne garder que les objets et fonctions génériques définis au-dessus. L'avantage est que l'ADI générique peut-être remplacée par une implémentation plus spécialisée (et donc potentiellement plus performante) mais au prix d'un travail de développement bien plus conséquent. C'est par exemple le choix qui sera effectué pour les réseau QSNet de Quadrics ou bien pour les réseau Myrinet.

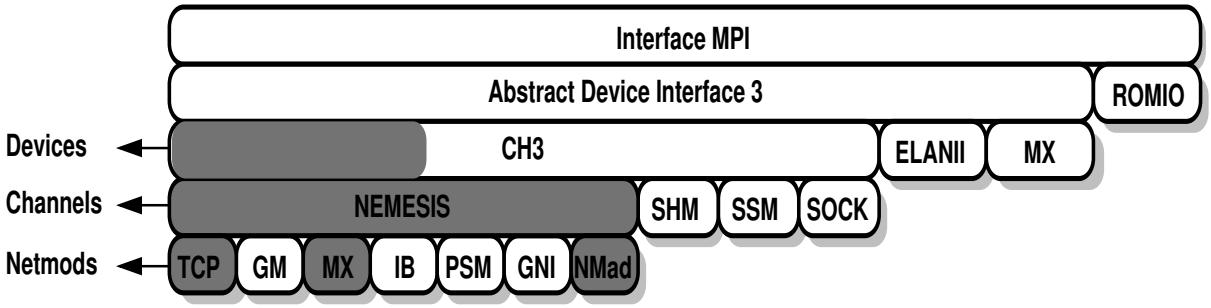


FIGURE 2.6 – Architecture logicielle de MPICH2.

De façon générale, cette refonte logicielle de MPICH2 avait pour but de proposer une implémentation efficace de MPI2 destinée aux machines massivement parallèles, aux machines SMP ainsi qu'aux grappes (de petite ou plus large échelle). Ainsi, un certain nombre de *channels* existaient déjà lors de mon arrivée :

— le *channel* sock, destiné aux communications inter-nœuds utilisant le réseau Ethernet/TCP ;
— le *channel* shm (*shared memory*), destiné aux communications intra-nœud utilisant de la mémoire partagée ;

16. Année de mon arrivée dans l'équipe de MPICH2.

— le *channel* sshm (*scalable shared memory*), destiné aux communications intra-nœud utilisant de la mémoire partagée ;
— le *channel* ssm (*sockets and shared memory*), destiné aux deux types de communications.

Avec trois *channels* sur quatre gérant les communications intra-nœud, il était clair que ce type de communications était considéré comme un objectif prioritaire. Cependant, il y avait des problèmes de performance et de plus certains constructeurs remettaient en cause le paradigme même du passage de messages, le taxant d'intrinsèquement inefficace et incapable de délivrer de hautes-performances.

Une analyse poussée du fonctionnement et du code des *channels* existants dans MPICH2 nous a convaincus que les performances globales pouvaient être drastiquement améliorées. Après une étude des diverses méthodes de communication intra-nœud possibles [CI12], nous avons proposé l'architecture d'un nouveau *channel* appelé NEMESIS et l'avons implémenté au sein de MPICH2. Les buts de NEMESIS étaient de résoudre les problèmes de passage à échelle mais aussi de supporter nativement les communications utilisant divers supports bas-niveau (aussi bien au niveau intra-nœud qu'inter-nœuds). À l'origine, NEMESIS était prévu pour être une interface générique de communication autonome et pouvant être intégrée dans de multiples constructions logicielles. Bien que NEMESIS a été intégré dans MPICH2 en tant que *channel* de l'ADI3, l'ambition a plus long terme était de remplacer cette dernière. Néanmoins, afin d'atteindre un excellent niveau de performances, nous avons modifié l'ADI3 en vue d'optimiser NEMESIS de façon plus poussée[17] (cf. section 2.5.2.5). Finalement NEMESIS deviendra indissociable de MPICH2 au point d'en constituer le cœur (du point de vue des communications) pendant de nombreuses années.

2.5.1.2 Principes généraux de conception de NEMESIS

La bibliothèque NEMESIS a été conçue afin de fournir à l'implémentation MPICH2 des communications extensibles et performantes, au niveau intra-nœud et inter-nœuds avec dans ce cas une prise en charge de multiples technologies d'interconnexion. Du point de vue de la conception, les priorités ont été ordonnées comme suit :

1. le passage à l'échelle (extensibilité) ;
2. des communications intra-nœud performantes ;
3. des communications inter-nœuds performantes ;
4. le support du multi-réseau

La conséquence de ce choix est que le surcoût de la bibliothèque dans le cas des communications intra-nœud doit être minimisé, même si cela doit se faire aux dépens de la performance des communications inter-nœuds.

Une approche basée sur des files de messages Afin d'atteindre ces objectifs d'extensibilité et de faible surcoût en intra-nœud, l'approche retenue a été d'utiliser des files de messages sans verrous (*lock-free queues*) résidant en mémoire partagée. Ainsi, chaque processus impliqué dans des communications n'a besoin que d'une unique file de réception dans laquelle les processus émetteurs situés dans le même nœud de calcul peuvent y insérer leurs messages sans le surcoût de l'acquisition d'un verrou pour synchroniser les accès. D'autres schémas de conception auraient pu employer une paire de files par paire de processus ou bien une unique file protégée par un verrou. Sur une grosse machine SMP ou multicœur, ces alternatives ne passeraient pas à l'échelle à cause du nombre de files nécessaires ($\mathcal{O}(N^2)$) ou de la contention sur le verrou.

17. Les parties grisées sur la figure 2.6 montrent les couches logicielles que j'ai implémentées ou modifiées.

Elles seraient de plus inefficaces en raison du surcoût causé par la scrutation des multiples files de réception et par l'acquisition du verrou.

Nous avons considéré deux types de modèles de mémoire partagée :

— un modèle à petite échelle (machine SMP sous Linux par exemple) caractérisé par un faible nombre de processeurs et où les processus doivent créer et attacher à leur espace d'adressage des zones de mémoire dédiées pour échanger des informations ;

— un modèle à large échelle (une machine ccNUMA par exemple) caractérisé par un nombre important d'unités de calcul et où les processus peuvent accéder à tous les espaces d'adressage sans avoir à créer de zone particulière ni besoin de s'y attacher.

Chaque processus possède donc une **file de réception** et une **file d'élements (buffers) libres** (ou *file libre* dans la suite du document) qui pourra être locale à chaque processus ou bien globale selon la mise en œuvre retenue (cf. section 2.5.1.2). La figure 2.7 montre le principe d'émission et de réception d'un message dans NEMESIS dans le cas de la file libre locale à chaque processus émetteur.

La séquence d'opérations est la suivante :

1. le processus émetteur retire un élément (un *buffer*) de la file libre ;
2. il copie ensuite le message à envoyer dans cet élément ;
3. cet élément est alors inséré dans la file de réception du processus destinataire ;
4. le processus destinataire retire l'élément de sa file de réception ;
5. il traite alors le message : copie à l'adresse finale (cas d'un message attendu) ou bien dans un tampon intermédiaire (cas d'un message inattendu) ;
6. le processus destinataire insère alors l'élément dans la file libre du processus émetteur (file d'où l'élement a été extrait au départ).

Dans le cas du modèle de mémoire à petite échelle, les files libres et de réception – ou plutôt les éléments qui les composent – doivent résider dans une zone de mémoire partagée accessible par l'ensemble des processus. L'ajout dynamique de files et d'éléments est donc difficile, ce qui a implique que ces objets doivent tous être alloués à l'initialisation des processus (`MPI_Init`). Cependant dans le cas du modèle à plus large échelle, comme il n'est pas nécessaire pour les processus d'attacher une zone spécifique de mémoire dans leur espace d'adressage, ces objets peuvent être ajoutés dynamiquement.

Mises en œuvre possibles pour la file de *buffers* libres

Plusieurs solutions étaient possibles en ce qui concerne l'emplacement de la *file libre*. La pre-

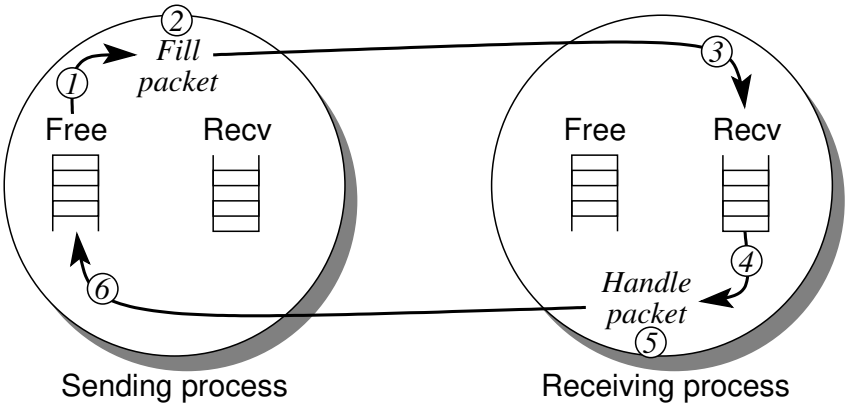


FIGURE 2.7 – Une émission/réception en mémoire partagée avec NEMESIS

mière solution est basée sur une file globale, la deuxième solutions utilise une file locale pour chaque processus récepteur dans laquelle les processus émetteurs retirent des éléments pour envoyer des messages tandis que dans la troisième solution, le processus émetteur retire un élément de sa propre file libre. Les avantages et inconvénients de ces approches sont détaillés ci-dessous.

– Solution avec file globale : La première approche permet une meilleure utilisation des zones de mémoire partagée dans le cas des systèmes avec un modèle de mémoire à petite échelle. Avec les autres approches, les éléments des files libres sont distribués parmi tous les processus émetteurs ou récepteurs et si certains ne communiquent pas, les éléments correspondants ne sont pas utilisés (mais néanmoins alloués). Une file globale corrige ce problème car elle est accessible par tous les processus. Cependant, des problèmes d'équité se posent car un processus peut potentiellement utiliser tous les éléments libres de la file. Un processus prêt à émettre un message peut donc se retrouver dans l'impossibilité de l'envoyer faute d'élément d'envoi disponible. Dans le cas de machines NUMA, la file peut également être localisée dans une partie de la mémoire qui n'est locale ni au processus émetteur et ni au processus récepteur, ce qui induit un surcoût d'accès.

– Solution avec file locale côté récepteur : La deuxième approche utilise un file libre locale, située dans la mémoire du processus récepteur. Pour les machines NUMA, le bénéfice est que seul le processus émetteur doit accéder à de la mémoire distante quand il faut modifier un élément de la file libre. De plus, comme ces éléments ne peuvent être utilisés que pour l'envoi de messages à un unique destinataire, le processus récepteur ne subit pas de famine : si aucun élément libre n'est disponible, c'est que le récepteur est occupé et donc reçoit déjà d'autres messages. Des problèmes d'équité demeurent cependant : un processus pourrait envoyer une grande quantité de messages à tous les autres et utiliser tous les éléments libres dans leurs files respectives. Un autre désavantage de cette approche réside dans la sous-utilisation des éléments alloués aux processus qui ne reçoivent pas ou peu de messages.

– Solution avec file locale côté émetteur : La troisième approche utilise également un file libre locale, mais située du côté de l'émetteur, ainsi que le montre la figure 2.7. Cette solution présente des avantages similaires à la précédente pour les machines NUMA, sauf que les processus récepteurs doivent accéder à de la mémoire distante. L'avantage supplémentaire est qu'un unique processus n'est pas en capacité d'utiliser tous les éléments libres dans une machine ce qui empêcherait les autre processus d'émettre des messages. Cependant, si un processus doit envoyer des messages à de multiples destinataires et que certains d'entre-eux ne remettent que lentement des éléments dans les files libres alors le processus émetteur est ralenti dans ses envois. Ce phénomène peut cependant être vu comme un mécanisme de régulation et de contrôle du flux d'émission, assurant une certaine équité car un processus particulier ne peut surcharger les files de réception des processus d'une machine entière. Enfin, cette solution présente l'inconvénient d'une sous-utilisation des éléments des files libres des processus n'envoyant que peu voire pas de messages.

– Discussion : Différentes approches impliquent différents type de files *lock-free*. Dans tous les cas, la file de réception doit être en capacité de gérer de multiples insertions concurrentes (les processus émetteurs) mais un unique retrait (le processus destinataire). Les différences concernent la file libre : dans la première approche, la file libre (unique et globale) doit permettre des insertions concurrentes ainsi que des retraits concurrents. Dans la seconde approche (file libre locale au processus récepteur) de multiples retraits concurrents mais une unique insertion doivent être gérés. Dans la troisième approche (file libre locale au processus émetteur) la file libre est comme la file de réception et doit donc être capable de supporter de multiples insertions concurrentes et un unique retrait. Le problème posé par les files sans verrous avec de multiples retraits concurrents est que si plusieurs processus scrutent une file vide, alors

quand un nouvel éléments est rajouté, il y aura de nombreuses transactions mémoire car des lignes de caches dans les processus scrutateurs seront invalidées puis rechargées. Une solution plus extensible consiste en la mise en place d'une liste chaînée de processus en attente pour que chaque processus puisse scruter une variable dans une ligne de cache qui lui est propre, mais établir une telle liste chaînée a pour conséquence que les processus ne peuvent rien faire d'autre en attendant qu'un élément devienne disponible sinon les processus suivants dans la liste pourraient être retardés. Ainsi que nous le verrons en section 2.5.2, c'est la troisième approche que nous avons retenue dans NEMESIS.

Les modules réseaux

Un ensemble de modules réseaux (les *netmods*) a été rajouté, modules qui s'interfacent avec le système de files de messages, ce qui permet une unification des méthodes d'envoi et de réception des messages. Le paradigme de conception est ici inversé : les communications inter-nœuds doivent rentrer dans le cadre des communications intra-nœud et non l'inverse, dans le but d'obtenir les meilleures performances. À l'arrivée d'un message sur une carte réseau, la fonction de scrutation dédiée du module réseau correspondant enfile ce message dans la file de réception du processus. Chaque module réseau possède en outre sa propre file d'émission. Du point de vue de l'émission d'un message le mécanisme mis en œuvre est identique : dans le cas d'une communication intra-nœud, on insère ce message directement dans la queue de réception du processus destinataire tandis que dans le cas inter-nœud, le processus émetteur insère ce message dans sa propre file d'émission. Le chemin critique d'émission s'en trouve simplifié puisqu'il n'y a plus de discrimination à faire entre intra- et inter-nœuds. Le multi-réseaux est supporté par l'ajout de modules réseaux supplémentaires. La section 2.5.3.3 présente certains des modules réseaux existants dans NEMESIS. La figure 2.8 montre le principe de fonctionnement de NEMESIS avec trois processus qui communiquent avec chacun sa file de réception sans verrou. Les modules réseaux permettent les communications inter-nœuds et les messages reçus depuis le réseau sont mis dans la file de réception sans verrou du processeur destinataire. Ainsi, un processus n'a besoin de scruter qu'une seule file quelle que soit la source du message à recevoir.

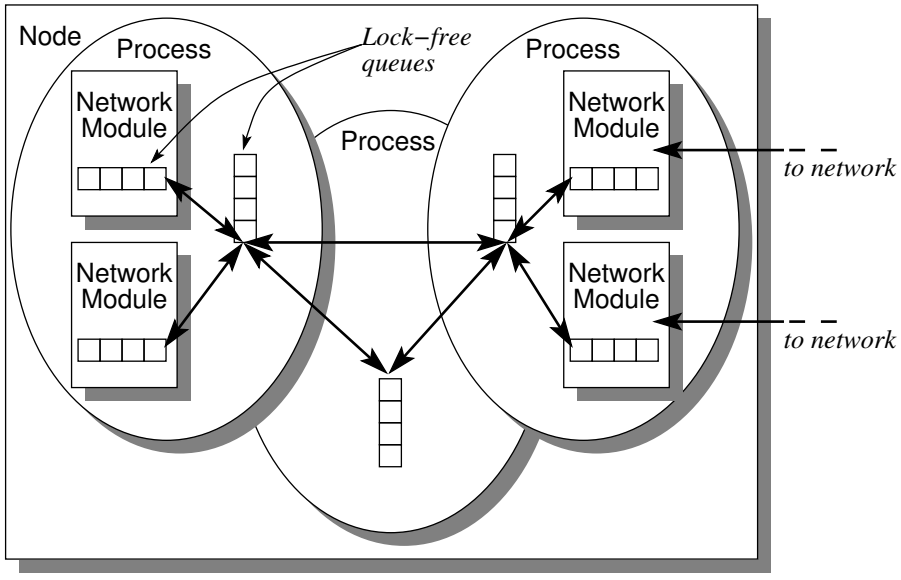


FIGURE 2.8 – Principe de fonctionnement de NEMESIS avec trois processus.

2.5.2 Mise en œuvre des communications intra-nœud dans NEMESIS

Dans l'implémentation de NEMESIS un seul processus par nœud alloue une zone de mémoire contenant la file libre et la file de réception de tous les autres processus qui projettent cette zone dans leur propre espace d'adressage. Chaque processus possède un tableau de pointeurs sur les files libres et de réception des autre processus. Étant donné que les adresses auxquelles ces zones sont projetées peuvent diverger d'un processus à l'autre, les pointeurs sont en réalité exprimés comme des décalages plutôt que comme des adresses absolues. Il y a donc *a priori* un surcoût puisque l'accès à un pointeur requiert un calcul supplémentaire de décalage, qui en pratique se révèle négligeable pour les performances.

2.5.2.1 Interactions entre la couche CH3 et le *channel* NEMESIS

Pour l'envoi d'un message, la couche CH3 appelle une fonction d'émission (*send*) implémentée par le *channel* choisi (NEMESIS en l'occurence) avec un pointeur sur la description des données à émettre dans un en-tête de message MPI ainsi qu'un autre pointeur sur un objet *requête* MPI. Cette description consiste en un *iovec*, c'est-à-dire une liste d'adresses et de tailles qui décrivent des données non contigües en mémoire. Le *channel* NEMESIS copie cet en-tête de message MPI et les données dans un élément de file de réception, appelé *cellule* (*cell*) puis y insère également un en-tête de message NEMESIS. Cette cellule est alors insérée dans la file de réception du processus destinataire (dans le cas d'une communication intra-nœud) ou bien dans la file d'émission d'un module réseau (dans le cas d'une communication inter-nœuds). Si la taille du message CH3 excède celle d'une cellule, plusieurs doivent être utilisées et elles sont émises en FIFO. Dans le cas où il n'y a plus assez de cellules libres, l'*iovec* décrivant les données non encore émises est sauvegardé dans la requête MPI qui est mise dans une file des envois en cours (file située dans la couche CH3). Dès que des cellules sont libérées, les messages en attente dans cette file sont envoyés autant que possible jusqu'à ce que l'*iovec* complet soit transmis. NEMESIS interroge alors la couche CH3 pour savoir si d'autres données doivent être émises. Si c'est le cas, l'*iovec* est «rechargé» avec de nouvelles informations et sinon la requête MPI est marquée comme complétée. Les cellules étant situées en mémoire partagée et en nombre limité, il est donc primordial de traiter une cellule et d'en extraire les données aussi vite que possible pour la libérer. Ainsi, un message inattendu est copié dans un tampon intermédiaire plutôt que de rester dans une cellule le temps que la requête en réception soit postée par le processus.

2.5.2.2 Cas des messages courts

NEMESIS suit la troisième approche décrite en section 2.5.1.2 car il est possible d'implémenter un système de file avec multiples insertions et unique extraction à l'aide de files sans verrous. Ainsi nous évitons le problème de processus multiples qui scrutent une file vide et les processus émetteurs dans les machines NUMA n'ont pas à accéder à de la mémoire distante pour récupérer une cellule et la remplir. Cela assure une plus grande équité parmi les processus et nous pouvons utiliser un algorithme simple, similaire à l'algorithme de verrou MCS [140] qui utilise des opérations atomiques de type *swap* et *compare-and-swap*. Le pseudo-code est donné par la figure 2.9.

Insertion d'un élément La file possède une tête et une queue. Pour insérer un élément, le processus échange atomiquement le pointeur sur l'élément à insérer avec celui sur la queue de la file. Si la valeur de queue est `NULL`, la file est vide et le pointeur de tête est également positionné sur l'élément. Si la file n'est pas vide, le processus positionne le pointeur `next`

```
Enqueue (queue, element) {
  prev = SWAP (queue->tail, element);
  if (prev == NULL)
    queue->head = element;
  else
    prev->next = element;
}

Dequeue (queue, &element) {
  element = queue->head;
  if (element->next != NULL)
    queue->head = element->next;
  else {
    queue->head = NULL;
    old = CAS (queue->tail, element, NULL);
    if (old != element) {
      while (element->next == NULL)
        SKIP;
      queue->head = element->next;
    }
  }
}
```

FIGURE 2.9 – Algorithme *Lock-free*, utilisant les opérations *swap* atomique (`SWAP`) et *compare-and-swap* (`CAS`).

de l'élément précédent sur le nouvel élément à insérer. Pour vérifier que la file est vide, un processus peut tout simplement vérifier la valeur du pointeur de tête.

Extraction d'un élément Pour extraire un élément de la file, le processus récupère un pointeur sur l'élément de tête. Si cet élément possède un pointeur `next` qui n'est pas `NULL`, le processus positionne la tête de la file sur cet élément suivant et l'opération d'extraction est terminée. Si le pointeur `next` est `NULL`, nous devons gérer une potentielle condition de course (*race condition*) : en effet, il est possible qu'un autre processus ait commencé à extraire un élément de la file et a effectué l'échange de la queue de la file mais n'a pas encore positionné le pointeur `next` de l'élément à extraire. À ce moment, le processus extracteur positionne le pointeur de tête de file à `NULL` et effectue une opération *compare-and-swap* avec le pointeur de queue de file et la valeur `NULL` seulement si la queue de la file pointe toujours sur l'élément qui est en cours d'extraction. Si cette opération de *compare-and-swap* arrive à échanger le pointeur de queue, alors aucun autre processus n'est en train d'insérer d'élément et l'extraction est un succès. Dans le cas contraire, un autre processus essaye d'insérer un élément et le processus essayant d'extraire attend sur le pointeur *next* jusqu'à l'insertion d'un élément par un autre processus qui va positionner ce pointeur *next* de l'élément extrait. Le processus extracteur termine son opération en positionnant la tête de file sur l'élément nouvellement inséré. Dans la version actuelle de NEMESIS, l'implémentation est relativement efficace, avec seulement six instructions pour une insertion et onze pour une extraction.

2.5.2.3 Cas des messages longs et protocole de type *rendez-vous*

Si les files de cellules en mémoire partagée sont très efficaces pour le transfert de messages de petite/moyenne taille, elles sont en revanche moins adaptées pour le transfert des gros messages. Les réseaux rapides possèdent des capacités d'accès direct en mémoire distante (*Remote*

Direct Memory Access) qui permettent de transférer des données directement depuis la mémoire du programme utilisateur (la source) vers le tampon de destination (lui aussi en mémoire utilisateur) situé sur un autre nœud de calcul, ce qui évite les copies mémoires intrinsèques à l'utilisation des files de cellules. Certaines machines SMP, comme les Altix de SGI offrent des mécanismes similaires pour des processus situés sur un même nœud. En l'absence de tels mécanismes, l'emploi des files partagées ne constitue pas la meilleure méthode pour transférer de grandes quantités de données entre processus s'exécutant sur un même nœud, ainsi que nous l'avons détaillé dans [CI12].

Interface pour les messages longs Afin de fournir un éventail de mécanismes pour le transfert de messages de grande taille, nous avons défini une interface dédiée appelée *Large Message Transfer* que nous avons rajoutée à la couche CH3. Éviter d'utiliser les files de NEMESIS améliore non seulement le débit du transfert mais permet également de réduire l'impact sur le cache par les données de l'application (cf. [CI12]). La couche CH3 utilise un protocole de transfert de type *rendez-vous* pour l'envoi de messages de grande taille. Ce protocole s'assure que la requête de réception a été postée du côté récepteur avant l'envoi effectif des données du message afin de ne pas avoir à les stocker dans un tampon intermédiaire et éviter ainsi une copie inutile dans le cas où la requête n'aurait pas été postée. La figure 2.10 montre les échanges de messages intervenant lors d'un protocole de type rendez-vous.

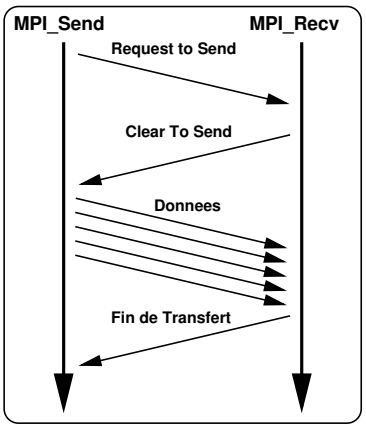


FIGURE 2.10 – Schéma d'un protocole de type rendez-vous.

Notre interface LMT est utilisée conjointement avec ce protocole *rendez-vous* et permet au *channel* NEMESIS de greffer des informations sur les messages CH3 du protocole *rendez-vous*. Un *channel* implémente trois fonctions pour l'envoi d'un message : `lmt_pre_send()`, `lmt_start_send()` et `lmt_post_send()` ainsi que trois `_recv` fonctions correspondantes pour la réception d'un message.

- Initialisation du transfert : Avant l'envoi du message *Ready-To-Send* (RTS), l'émetteur appelle `lmt_pre_send()`, indique la destination et décrit les données à envoyer. Cela permet au *channel* de faire la mise en place nécessaire pour le transfert (comme enregistrer le tampon d'émission par exemple). Le *channel* renvoie un *cookie* d'émission qui est une donnée de longueur variable qui sera envoyée avec le message RTS au destinataire. Selon l'implémentation, ce *cookie* peut contenir des clefs d'enregistrement, une description du buffer d'émission, une adresse, etc.

- Traitement du message RTS : À la réception du message RTS et après l'avoir fait correspondre à une requête de réception déjà postée, la couche CH3 appelle la fonction `lmt_pre_recv()` avec en argument le *cookie* du message RTS et la description du tampon de réception.

Cette fonction permet au récepteur de se préparer au transfert des données. Le *channel* renvoie un *cookie de réception* et indique si la couche CH3 doit envoyer ou non un message *Clear-To-Send* (CTS). Certains mécanismes de transfert, comme les opérations de type *get* dans le cas des RDMA ne requièrent aucune action de la part de l'émetteur qui reste passif (un message CTS est donc inutile). Si cela est nécessaire, la couche CH3 envoie le message CTS avec le *cookie de réception* généré par le récepteur. Ensuite, CH3 appelle `lmt_start_recv()`. À partir de ce moment, le *channel* – du côté récepteur – possède toute l'information nécessaire pour commencer le transfert des données.

- Traitement du message CTS : Lorsqu'un message CTS émis par le processus récepteur est reçu par le processus émetteur, CH3 appelle la fonction `lmt_start_send()` avec le *cookie de réception* extrait du message CTS. L'émetteur possède à présent toute l'information nécessaire pour commencer le transfert des données. Selon le mécanisme utilisé pour ce transfert, seule une partie ou les deux parties sont actives. Dans ce dernier cas (e.g. en copiant dans un tampon en mémoire partagée) alors les deux processus appellent respectivement `lmt_post_send()` et `lmt_post_recv()` une fois le transfert accompli, de façon à procéder au nettoyage post-transfert (e.g. désenregistrement de la mémoire ou détachement de la zone de mémoire partagée). Cependant, si un seul côté est actif, comme dans le cas des opérations unilatérales RDMA de type *put* ou *get*, alors le processus actif envoie un message `DONE` pour signifier au processus passif que l'opération en mémoire est terminée. À la réception de ce message, CH3 fait appel à `lmt_post_send()` ou `lmt_post_recv()` de façon appropriée selon le type de transfert.

- Gestion des informations complémentaires : Certains mécanismes peuvent éventuellement avoir besoin d'échanger des informations complémentaires durant le transfert. Par exemple, la taille des données à transférer peut être supérieure à la quantité de mémoire qu'un réseau est capable d'enregistrer et le message doit être transféré par morceaux. Pour La gestion de ce type de cas, le *channel* envoie un message `COOKIE` au processus distant. À la réception de ce message, CH3 appelle la fonction `lmt_handle_cookie()` implémentée par le *channel*. Ce message peut être utilisé pour indiquer au processus distant que de la mémoire supplémentaire a été enregistrée ou bien qu'une partie du message a bien été transmise et que la mémoire associée est libérable.

Mise en œuvre des LMT dans NEMESIS NEMESIS implémente une interface LMT pour les communications intra-nœuds par mémoire partagée et pour les communications inter-nœuds avec certains modules réseaux (cf. section 2.5.3). Pour la mémoire partagée, un tampon intermédiaire partagé est utilisé. Le récepteur alloue une zone de mémoire partagée et donne l'information à l'émetteur via un *cookie* de réception. Quand l'émetteur reçoit le message CTS avec ce *cookie*, il attache la zone de mémoire dans son espace d'adressage et commence à copier les données dans le tampon. Les données sont copiées selon la technique du *double buffering* pour permettre à deux processeurs de procéder simultanément à la copie des données. Comme les deux parties sont actives pour le transfert, il n'est pas nécessaire d'envoyer un message `DONE`.

2.5.2.4 Support des accès mémoire distants

Les accès mémoire distants (*Remote Memory Access*) sont également supportées dans NEMESIS. Pour rappel, ces opérations permettent à un processus – initiateur de l'opération – d'accéder à des variables situées dans l'espace d'adressage d'un autre processus (potentiellement sur un autre nœud de calcul) qui est la cible de l'opération. L'initiateur de l'opération fournit tous les arguments nécessaires à l'opération (d'où le qualificatif souvent donné d'opérations «unilatérale»). NEMESIS permet la réalisation d'opérations de type *Put* (écriture dans l'espace

d'adressage) et *Get* (récupération dans l'espace d'adressage) avec des données contigües ou non. Dans ce cas, les données sont décrites à l'aide d'un tableau d'adresses et de longueurs.

Principes de conception des accès mémoire distants Il existe deux interfaces pour effectuer des accès mémoire distants, selon que la mémoire partagée ou bien le réseau est utilisé. Dans le premier cas, l'opération repose sur le concept de fenêtre, qui est une zone de l'espace d'adressage du processus à laquelle les autres processus vont accéder. Avant de pouvoir réaliser une opération dans une fenêtre, un processus initiateur doit avoir les informations concernant la zone et s'y attacher. Dans le cas des réseaux, la mémoire utilisée doit être enregistrée par le réseau qui fournit une clef d'identification la concernant. Cette clef doit être envoyée à l'ensemble des processus initiateurs qui vont l'utiliser pour effectuer les opérations.

Les accès mémoire distants utilisant les fenêtres sont des opérations bloquantes : la fonction retourne une fois l'opération terminée. Celles utilisant le réseau sont non-bloquantes : un pointeur sur une variable (non nulle) indiquant la fin de l'opération est donné à l'opération quand elle est initiée. NEMESIS met cette variable à 0 une fois l'opération terminée. L'application doit appeler une fonction de communication ou de scrutation de NEMESIS pour avoir une garantie que l'accès mémoire distant est terminé. Selon le type de réseau (et le module implémentant son support dans NEMESIS), le processus cible peut être amené lui aussi à appeler des fonctions de ce type pour avoir une garantie de complétion de l'opération.

Mise en œuvre dans NEMESIS Lorsqu'un processus alloue une fenêtre, un fichier servant de support pour une zone de mémoire partagée est créé et projeté dans l'espace d'adressage de ce processus. La description de la fenêtre contient le nom de ce fichier ainsi que l'adresse de projection. Ainsi, quand un autre processus s'attache à cette fenêtre, il projette à son tour le fichier dans son espace d'adressage. Le processus initiateur ne pouvant pas projeter la zone à la même adresse que le processus cible, l'adresse d'un objet dans la fenêtre est différente entre l'initiateur et la cible. Pour cette raison, l'interface RMA spécifie que les adresses passées an argument aux fonctions RMA doivent être des adresses du processus cible. Le processus initiateur n'a même pas besoin de savoir où la fenêtre est projetée dans son propre espace d'adressage. L'opération RMA de l'initiateur traduit l'adresse avant d'être réalisée.

2.5.2.5 Optimisations des communications intra-nœud

Nous allons maintenant détailler dans cette section une série d'optimisations que nous avons mises en place afin de rendre NEMESIS le plus performant possible. Nous ne décrivons que les principales et d'autres sont abordées dans [CI13] et [RI4], notamment l'amélioration des copies mémoires et l'utilisation de fonctions *inline* pour aboutir sur un code plus rapide.

Doubler les files de cellules avec des *Fast Boxes* La latence peut être réduite davantage en évitant les files de cellules et en les remplaçant par un mécanisme que nous avons appelé des *Fast Boxes*. Une *Fast Box* est un simple tampon auquel est adjoint un drapeau *vide/rempli*. Si une *Fast Box* est vide, un processus va y copier son message et mettre le drapeau à *rempli* plutôt que d'utiliser la file de réception du processus destinataire. Ce dernier vérifie d'abord si la *Fast Box* est remplie avant de regarder sa file de réception. Après que le processus extracteur a traité le message, il remet le drapeau à *vide*. Il n'y a qu'une paire de *Fast Box* par paire de processus sur un nœud de calcul. Une telle approche pose des problèmes de passage à l'échelle pour les machines SMP de grande taille ou les processeurs avec un nombre élevé de cœurs de calcul, mais elle est acceptable pour des machines de petite ou moyenne taille. Dans le cas de machines

de grande taille, il est possible de n'utiliser des *Fast Box* qu'entre voisins les plus proches ou bien de les créer dynamiquement selon le schéma de communication applicatif [18].

– Scrutation de multiples sources : un problème introduit par les *Fast Boxes* est que même s'il ne subsiste qu'une seule file à scruter pour les processus (la file de réception), l'ajout de *Fast Boxes* entraîne pour le récepteur de devoir vérifier plusieurs zones en mémoire pour la réception. Or, dans le paradigme du passage de messages, l'application doit spécifier la source du message quand elle poste une requête de réception. NEMESIS fournit une fonction indiquant de quels processus des messages sont attendus. CH3 peut utiliser cette fonction au moment où l'application poste une requête de réception et NEMESIS saura quelles *Fast Boxes* regarder. Pour gérer le cas `MPI_ANY_SRC`, NEMESIS va regarder toutes les *Fast Boxes*, mais moins fréquemment afin de réduire l'impact sur la réception des autres messages.

– Conservation de l'ordre des messages : un autre problème concerne l'ordre des messages puisqu'il existe désormais deux chemins de réception entre un émetteur et un récepteur : la file de réception et la *Fast Box*. Un émetteur peut tout à fait envoyer un message en l'insérant dans la file de réception du processus destinataire et envoyer le message suivant dans la *Fast Box*. Avant d'utiliser la *Fast Box*, l'émetteur ne peut pas savoir si le récepteur a déjà reçu le message dans sa file de réception. De la même façon, un récepteur qui vérifie une *Fast Box* ne peut savoir s'il existe d'autres messages émanant du même émetteur dans sa file de réception. S'il vérifie le message dans la *Fast Box*, l'ordre des messages ne sera pas conservé. NEMESIS utilise des numéros de séquence pour gérer ce cas de figure : si le numéro de séquence du message dans la *Fast Box* n'est pas celui attendu, c'est qu'il y a un message dans la file de réception qui doit être retiré prioritairement. Et inversement si le numéro de séquence dans la file de réception n'est pas le bon.

Réduire le nombre de défauts de cache Premièrement, nous avons regroupé dans une même ligne de cache les variables souvent utilisées conjointement afin à reduire le nombre de défauts de cache L1 lors de l'exécution du chemin critique. De plus, des défauts de cache (*cache misses*) de niveau 2 sont inévitables à partir du moment où des processus situés sur des processeurs distincts accèdent aux mêmes emplacements mémoire. Étant donné que l'architecture de NEMESIS est basée sur des files de messages en mémoire partagée, les défauts de L2 sont relativement coûteux [19] et il est donc impératif d'éliminer autant que faire se peut ces défauts au moment d'accéder aux pointeurs de tête ou de queue servant à la gestion des files de messages.

– Problèmes de (faux) partage : un processus doit accéder à la fois à la tête et à la queue de la file pour y insérer un élément quand elle est vide ou pour extraire la dernière cellule de la file (cf. la figure 2.9). Dans de tels cas, si le pointeur de tête et le pointeur de queue sont stockés dans la même ligne de cache, un seul défaut de L2 va se produire. Si la file contient plus de cellules, alors le processus émettant une cellule n'a besoin que d'accéder à la queue et le processus récepteur qu'à la tête. Si les pointeurs de tête et de queue étaient dans la même ligne de cache, des défauts de L2 se produiraient à cause du faux partage à chaque fois qu'un processus rajoute une cellule après qu'une autre a été retirée et inversement. Dans un tel cas de figure, il est préférable que tête et queue ne partagent pas la même ligne de cache.

– Le pointeur fantôme : la solution implémentée dans NEMESIS est de placer la tête et la queue dans une même ligne de cache et d'avoir en supplément un pointeur *fantôme* de tête dans une ligne du cache distincte. Ce pointeur est initialisé à `NULL`. Le processus extracteur utilise la tête fantôme en lieu et place de la tête réelle excepté quand la tête fantôme est à `NULL`, ce qui signifie que la file est vide. Si la tête fantôme est à `NULL` quand le processus extracteur essaye de retirer une cellule, alors il vérifie la valeur de la tête «réelle». Si cette tête n'est pas à

18. Un peu sur le modèle des connexions à la demande dans certains modules réseaux par exemple.
19. Environ 400 cycles soit environ 200 ns sur un processeur de type Dual Xeon cadencé à 2GHz.

`NULL`, cela signifie qu'une cellule a été inserée dans la file depuis que cette dernière a été vidée. Le processus extracteur initialise sa tête fantôme avec la valeur de la tête «réelle» et positionne cette dernière à `NULL`. Ainsi, un seul défaut de cache L2 se produit quand une cellule est insérée dans une file vide ou quand l'unique cellule d'une file est extraite. De plus, les pointeurs de tête et de queue étant situés dans des lignes de cache distinctes, les défauts de L2 dûs au faux partage sont évités.

Courtourner les files de messages de la couche CH3 La latence des messages de petite taille peut être davantage réduite en appliquant une optimisation consistant à contourner dans certains cas les deux files de réception globales de la couche CH3 : la file des messages attendus (pour lesquels une requête de réception a été postée) et la file des messages inattendus (cas contraire). Dans l'implémentation de CH3, quand une requête de réception est postée par l'application (e.g. avec un appel à `MPI_Recv`), la file des messages inattendus est vérifiée pour trouver un message correspondant qui aurait déjà été reçu. Si tel n'est pas le cas, la requête est postée dans la file de réception des messages attendus. Ensuite, CH3 fait appel au moteur de progression des communications pour vérifier l'arrivée de nouveaux messages. Quand un nouveau message est reçu, la couche CH3 cherche une requête correspondante dans la file des messages attendus et si elle n'en trouve pas, elle insère le message dans l'autre file, celle des messages inattendus.

Il est à noter que si une requête de réception est postée, pour laquelle il n'y a pas de message en attente dans la file des messages inattendus, et que le message correspondant est en attente de réception dans une file de NEMESIS (file de réception ou file réseau), alors la requête de réception est insérée dans la file des messages attendus pour être extraite juste ensuite quand le moteur de progression est appelé et que le message correspondant est effectivement reçu.

L'optimisation que nous avons mise en place consiste en une nouvelle fonction qui fait appel au moteur de progression des communications avec une requête de réception spécifique. Au moment où des messages sont reçus depuis une file de réception de NEMESIS, ils sont comparés avec cette requête. C'est uniquement dans le cas où aucun message ne correspond à cette requête que cette dernière est insérée dans la file CH3 des messages attendus. Si une requête est déjà présente dans la file CH3 des messages attendus et qu'elle peut potentiellement correspondre à cette requête de réception, cette optimisation n'est pas utilisable et à la place nous adoptons le comportement original.

Le LMT *Kernel Nemesis* (Knem) Afin d'optimiser les échanges de messages longs en mémoire partagée, une interface LMT est proposée au-dessus d'un module noyau permettant d'éviter des recopies mémoires entre processus. Ce module est appelé KNEM (pour *Kernel* NEMESIS) et a été intégré dans MPICH2 dans le cadre de l'Équipe-Associée MPI-Runtime [CI9]. KNEM est principalement le fruit des travaux de Brice GOGLIN et Stéphanie MOREAUD ([145], [73]) et a par la suite été intégré dans Open MPI. La figure 2.11 montre la pile logicielle de MPICH2 avec les parties concernant KNEM.

Le module KNEM pour linux supporte les communications non-bloquantes, asynchrones et vectorielles (permettant le transfert de données vers ou depuis des zones mémoire non-contigües) et tire parti des fonctionnalités de déport de copies mémoire dans le matériel (*Input/Output Acceleration Technology* [20]). Ce module s'appuie sur un périphérique caractère (`/dev/knem`)

20. La technologie I/O AT (*Input/Output Acceleration Technology*) implémentée dans les contrôleurs mémoire Intel® comprend un dispositif matériel, le moteur DMA (*DMA Engine*), capable d'effectuer efficacement des copies mémoire en arrière plan. Il libère ainsi le processeur de ce transfert au profit de travail «utile». Ce modèle empêche par la même occasion la pollution des caches, exonérés des données relatives à la copie.

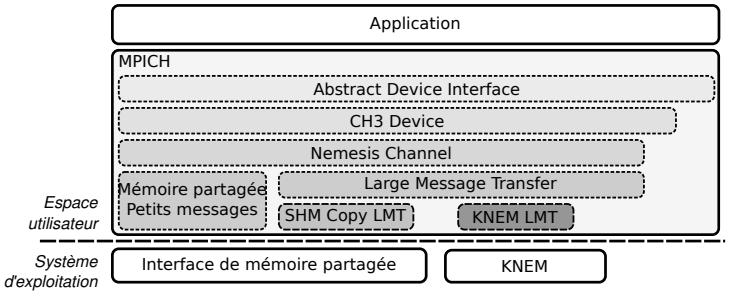


FIGURE 2.11 – Architecture de MPICH2-Nemesis avec le LMT Knem.

qui implémente deux principales commandes de communication. Le processus émetteur déclare une zone mémoire d'émission à KNEM et obtient un *cookie* en retour (commande *Send*). Ce *cookie* est ensuite envoyé au destinataire via un message RTS. Le récepteur qui souhaite récupérer les données dans une zone mémoire la déclare à KNEM en lui associant le *cookie* d'émission (commande *Recv*). Après avoir récupéré la zone mémoire d'émission via le *cookie* , le module KNEM prend en charge le transfert de données d'une zone mémoire à l'autre directement depuis le noyau linux.

2.5.3 Support des communications inter-nœuds : les module réseaux

2.5.3.1 Implémentation générale des modules réseaux (*netmods*)

Le système de communication intra-nœud décrit en section 2.5.1.2 est secondé par un autre système pour les communications inter-nœuds, qui s'appuie sur des modules réseaux (*netmods*). Le fonctionnement entre un processus et son module réseau [21] est similaire à celui de deux processus situés sur une même machine. Chaque module réseau possède une file d'émission (qui joue le même rôle que la file de réception en mémoire partagée du processus) et une file libre. Les messages à envoyer par le réseau sont insérés dans la file d'émission du module réseau en utilisant des cellules extraites de la file libre du processus. Le module réseau extrait les messages de sa file d'émission et les envoie sur le réseau. Les messages reçus depuis le réseau sont insérés dans la file de réception du processus en prenant des cellules dans la file libre du module réseau du processus récepteur.

Similitudes avec la mémoire partagée De la même façon qu'un processus maintient un tableau de pointeurs sur les files de réception et les files libres des autres processus locaux, des pointeurs sont également présents pour les processus distants, sauf que le pointeur sur la file de réception pointe maintenant sur la file d'émission du module réseau et le pointeur de file libre pointe sur la file libre du module réseau. Le module réseau détermine à quel processus distant envoyer un message en examinant l'en-tête de ce dernier. Ainsi, le code pour envoyer des messages intra-nœud et inter-nœuds est le même ce qui rend le chemin critique très efficace : dans tous les cas, un processus extrait puis remplit une cellule prise dans une file libre, il déréférence le pointeur sur la file de réception du processus récepteur et l'insère dans cette même file. Dans le cas inter-nœuds, toutefois, le pointeur sur la file de réception est en fait un pointeur sur la file d'émission du module réseau. Également, comme le module réseau insère les messages reçus dans la file de réception du processus, la réception de messages émis par

21. Par simplicté, nous supposons ici qu'il n'y a qu'un réseau de disponible mais le support multi-réseaux est possible dans NEMESIS.

des processus distants est identique à la réception de messages émis par des processus situés sur un même nœud.

Considérations pratiques d'implémentation La file libre du module réseau doit être allouée dans une zone de mémoire partagée quand bien même les cellules ne sont utilisées que par un seul processus. En effet, quand le module réseau insère une entrée dans la file de réception d'un processus et qu'un autre processus essaye d'y insérer un autre élément ensuite, ce processus doit être capable de modifier la valeur du pointeur *suivant* de l'élément du module réseau. Afin que le module réseau puisse faire progresser les messages en émission, (depuis sa file d'émission) et la réception de messages dans la file de réception du processus, ce dernier appelle régulièrement la fonction de scrutation du module réseau.

Cas des tampons réseaux pré-enregistrés Lors de l'utilisation de réseaux de niveau utilisateur contournant le système d'exploitation et qui demandent d'enregistrer des tampons d'émission et de réception, des copies mémoires peuvent être évitées dans le module associé en enregistrant toutes les cellules de la file libre du processus. Ainsi, quand le processus insère un message dans la file d'émission du module réseau, ce dernier peut directement envoyer le message sans avoir à le copier dans un tampon pré-enregistré. Pareillement, des copies mémoires peuvent être évitées lors de la réception d'un message en enregistrant toutes les cellules de la file libre du module réseau. Ces cellules peuvent ainsi être extraites et deviennent utilisables directement en tant que tampons de réception pour le réseau. Les messages arrivants sont reçus directement dans les cellules de la file qui sont insérées dans la file de réception du processus. Comme les cellules de la file sont remises dans la file libre du module réseau, elles sont de nouveau marquées comme tampons de réception pour le réseau.

Pour les réseaux de niveau utilisateur requiérant des tampons d'émission et de réception mais ne pouvant employer la méthode décrite précédemment, des copies peuvent toujours être évitées à la réception d'un message. Un message entrant est reçu dans un tampon de réception du module réseau. Ensuite, une cellule insérée dans la file de réception du processus inclut un pointeur sur ce tampon de réception plutôt que sur le message lui-même. Les tampons sont réutilisés quand le processus insère dans la file libre du module réseau une cellule référençant le tampon. Cette méthode exige du processus une action particulière quand il envoie un message à un processus distant via un module réseau ce qui rallonge le chemin critique.

2.5.3.2 Support multi-réseaux : vers plus de hiérarchie

Nous avons détaillé pour le moment le support d'un seul module réseau mais rien dans l'architecture de NEMESIS n'empêche la présence de multiples modules pour supporter plusieurs technologies d'interconnexion simultanément. Ainsi, il serait possible d'avoir un module pour un réseau de type Infiniband destiné aux communications intra-grappe tandis qu'un réseau de type Ethernet sera plutôt utilisé pour des communications inter-grappes. Le support de plusieurs technologies d'interconnexion rajoute des possibilités quant aux hiérarchies matérielles pouvant être gérées par notre implémentation de MPI. Du point de vue de la conception, une multiplicté de modules réseaux implique une multiplicité des fonctions de scrutations associées. Il faut donc que le processus appelle la fonction pour chaque module et que les pointeurs soient correctement positionnés dans le tableau des pointeurs de files libres et de réception du processus.

Les réseaux qui ont besoin d'enregistrer des tampons d'émission et de réception n'autorisent pas d'autres réseaux à enregister les mêmes pages mémoire et donc un unique réseau est autorisé à enregistrer tous les éléments de la file. Nous appelons ce réseau le réseau *principal*.

Les autres réseaux vont devoir utiliser le second mécanisme décrit précédemment, c'est-à-dire utilisant des pointeurs sur les tampons de réception et effectuant une copie additionnelle. Ce sont les réseaux dits *secondaires* [22].

Ainsi, quand les réseaux sont organisés hiérarchiquement, et qu'un seul réseau est utilisé pour les communications intra-grappe tandis qu'un autre est utilisé pour les communications inter-grappes, alors le premier sera le réseau principal tandis que le second sera le réseau secondaire. Ce réseau inter-grappe aura des latences plus élevées (car moins performant) et n'aura sans doute pas autant besoin de réduire le nombre de copies mémoire.

2.5.3.3 Instances de modules réseaux dans NEMESIS

NEMESIS supporte donc un large éventail de technologies réseaux grâce à divers modules réseaux dédiés. Chaque module implémente un jeu restreint de fonctions, surtout en comparaison des *channels* ou *devices*. Schématiquement, les quatre fonctions suivantes doivent être implémentées par un module réseau :

- `net_module_init` sert à initialiser le module, par exemple en réservant des ressources, enregistrer des tampons d'émission et de réception, etc. ;
- `net_module_send` sert à envoyer un message, c'est-à-dire le mettre dans la file d'émission du module ;
- `net_module_poll` sert à faire progresser les communications pour le module, c'est-à-dire vérifier l'arrivée de nouveau messages (et les insérer dans la file de réception du processus) et aussi envoyer sur le réseau proprement dit les messages présents dans la file d'émission du module ;
- `net_module_finalize` sert à libérer les ressources utilisées par le module.

Un module n'a pas besoin d'implémenter de fonction `net_module_recv` car la fonction de scrutation `net_module_poll` appelée par le moteur de progression général de NEMESIS se charge déjà des réceptions de messages provenant du réseau.

Les réseaux supportés dans NEMESIS par le biais d'un module réseau sont :
- Ethernet/TCP : historiquement, c'est le tout premier module réseau qui a été intégré dans NEMESIS. J'ai effectué une première implémentation une fois que le travail sur les communications intra-nœuds a été suffisamment avancé. Le support d'Ethernet/TCP était capital car il n'était envisageable d'avoir un *channel* ne supportant que les communications intra-nœud. Ce module a par la suite été optimisé pour en améliorer les performances [CI13]. Un module SCTP a également été développé pour NEMESIS ;

- Myrinet/GM/MX : les interfaces de programmation de bas-niveau GM et MX sont supportées dans NEMESIS. Initialement, c'est Darius BUNTINAS qui a implémenté le module utilisant GM. Par la suite, GM est tombée en désuétude et a été remplacée par MX. Dans le cadre de l'Équipe-Associée Inria, j'ai donc développé le module MX afin de pouvoir continuer le support des réseaux Myrinet dans MPICH2.

- Quadrics QSNet : un module destiné au réseau hautes-performances QSNet de Quadrics a été également implémenté par mes soins. Ce travail n'a fait l'objet ni de documentation ou de publication ;

- Cray Gemini/Aries/uGNI : le réseau Gemini (et son successeur Aries) de Cray est supporté dans NEMESIS également. via son interface *User-level Gemini Network Interface* (uGNI). Ce module a pris pour base le module Myrinet/MX existant et son implémentation est décrite plus en détails dans [165] ;

22. Note : tous les réseaux secondaires n'ont pas vocation à effectuer des copies additionnelles. TCP/Ethernet, par exemple, n'en n'a pas besoin.

- Infiniband : un module spécifique décrit dans [129] a été implémenté par l'équipe responsable du développement de MVAPICH2 (implémentation dérivée de MPICH2). Ce module réseau qui ne fait pas partie de la distribution de base de MPICH2 [23] ;

- NEWMADELEINE : dans le cadre de l'Équipe-Associée Inria MPI-Runtime, j'ai effectué le portage de la bibliothèque de communication générique NEWMADELEINE dans MPICH2-NEMESIS L'avantage de la présence d'un module réseau NEWMADELEINE est la possiblité d'utiliser des technologies réseaux qui ne sont pas forcément supportés nativement dans MPICH2. Par exemple, comme Infiniband n'est disponible que dans MVAPICH2, le support de NEWMADELEINE nous laisse donc la possibilité d'utiliser MPICH2 dans une grappe équipée du réseau Infiniband sans être contraint dans l'implémentation choisie (même si Open MPI reste un autre choix valable). Par ailleurs, NEWMADELEINE effectue de nombreuses optimisations et MPICH2-NEMESIS peut donc en profiter «gratuitement». C'est par exemple le cas de l'agrégation des messages qui permet de réduire les nombres de transferts effectifs sur le réseau ou encore du support du multirail autorisant l'agrégation le cas échéant des bandes passantes de multiples réseaux d'interconnexion installés au sein de la grappe [AI3]. Enfin, il est possible de bénéficier de modes de scrutation avancés grâce au moteur PIOMAN basé sur des threads [199]. L'implémentation de ce module NEWMADELEINE a nécessité de modifier la couche CH3 afin de mettre en place des optimisations particulières (cf. section 2.5.3.4) car l'interface de NEWMADELEINE est proche de celle de l'ADI3. Il est donc possible d'appeler directement NEWMADELEINE à ce niveau sans descendre dans les couches du *channel* NEMESIS dont l'interface est différente, ce qui est une source potentielle d'inefficacité. Cependant, le problème est que nous n'utilisons plus le système centralisé de réception de NEMESIS puique les communications inter-nœuds sont gérées directement par NEWMADELEINE. Il faut donc créer une correspondance entre des requêtes de l'ADI et des requêtes NEWMADELEINE et conserver une cohérence entre toutes ces requêtes, ce qui pose problème notamment dans le cas de l'utilisation de `MPI_ANY_TAG` ou `MPI_ANY_SRC`. Plus de détails sont disponibles dans [CI11] qui montre quelles solutions sont envisageables.

2.5.3.4 Optimisations des communications inter-nœuds

Si les communications intra-nœud ont fait l'objet d'une attention toute particulière, il n'en demeure pas moins que nous avons cherché également à fournir des communications inter-nœuds les plus efficaces possible. Nous décrivons quelques optimisations qui permettent une amélioration substantielle des performances de ces communications.

Contournement des files de messages pour les communications inter-nœuds Le fonctionnement classique des modules réseaux dans NEMESIS tel que décrit en section 2.5.1.2 peut être amélioré du côté émetteur. En effet, au lieu d'insérer un message dans la file d'émission du module réseau puis d'appeler le moteur de progression qui va extraire ce message pour l'envoyer, nous effectuons directement l'envoi quand la file d'émission est vide (l'ordre des messages doit être conservé). Notre intention première était de simplifier le chemin critique et donc d'adopter une interface similaire entre communications intra-nœud et inter-nœuds, mais le surcoût de la vérification est négligeable pour les performances intra-nœud tandis que les performances inter-nœuds sont significativement améliorées.

23. Vraisemblablement pour des questions politiques.

2.5.3.5 Exploitation des capacités de *Tag-Matching* des réseaux

Une optimisation importante réside dans l'exploitation de la capacité de *Tag-Matching* de certains réseaux rapides ou bibliothèques de communication. Pour rappel, les messages MPI sont identifiés par un triplet (*source*, *étiquette*, *contexte*) :

— la *source* correspond au rang du processus émetteur du message ;
— l'*étiquette* (*tag*) permet de réaliser un multiplexage/démultiplexage des communications entre deux processus ;
— le *contexte* correspond au communicateur MPI dans lequel la communcation du message est effectuée. En effet, la *source* étant un numéro de rang, cela n'a de sens qu'au sein d'un communicateur spécifique (cf. section 3.2.1.1). Il est donc précisé avec ce champ.

À noter que la taille du message ne fait *pas* partie de ses informations d'identification, ce qui permet de gérer les cas de troncature en réception. Lorsqu'un message est reçu, l'implémentation MPI compare ses informations d'identification qui se trouvent dans son en-tête avec d'autres informations postées par l'application qui identifient les messages attendus par cette dernière. Cette opération d'appariement des requêtes d'émission et de réception se produit au niveau de la couche CH3 car c'est elle qui gère les files de messages (postés/attendus et inattendus) correspondantes. La plupart des réseaux ignorent cela car ils ne sont pas destinés à supporter spécifiquement un standard tel que MPI.

Bénéfice pour les petits messages Cependant, certains réseaux rapides proposent une interface qui utilise directement *toutes* les informations d'en-tête (*source*, *étiquette*, *contexte*) et pas uniquement la *source* comme c'est souvent le cas. La latence pour les petits messages sera réduite car seules les données MPI (i.e en-tête MPI et données de l'application) seront transmises sans avoir à rajouter des données de gestion utilisées par les protocoles internes à l'implémentation (typiquement, NEMESIS va rajouter ses propres en-têtes aux messages). La figure 2.12 illustre cette différence d'approche.

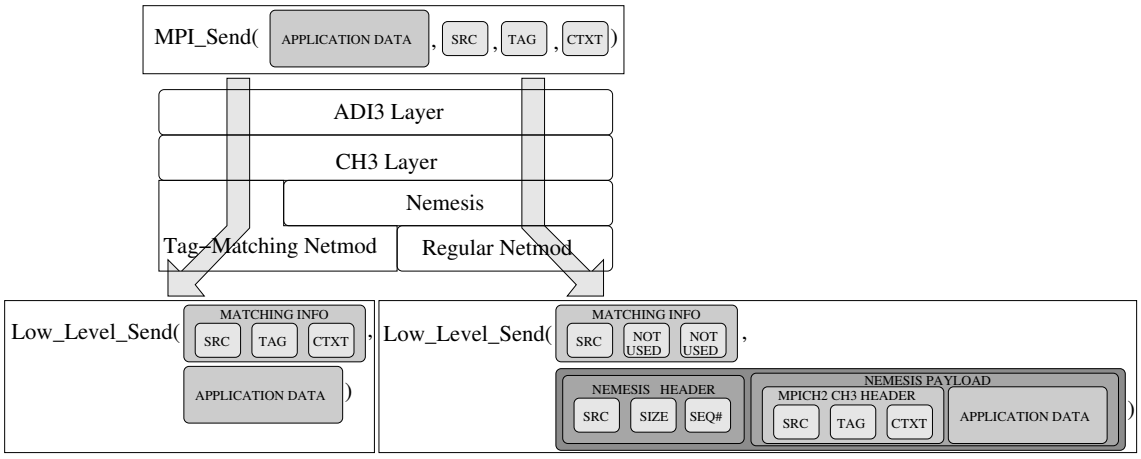


FIGURE 2.12 – Pile logicielle de MPICH avec et sans *Tag Matching*

Bénéfice pour les grands messages Les messages de grande taille peuvent également bénéficier de ce mécanisme car il permet d'éviter des échanges de messages superflus. La figure 2.13 expose le problème : supposons que l'on veuille échanger un message de grande taille : le protocole de type *rendez-vous* est alors utilisé. Dans un tel cas, il y a un échange de messages de contrôle (RTS et CTS) entre l'émetteur et le récepteur avant l'envoi des données de l'ap-

plication proprement dites. Comme ces données sont volumineuses, il est fort probable que le réseau utilise lui-même en interne un protocole de type rendez-vous juste pour ce dernier envoi, ce qui se traduit par un nouvel échange totalement superflu car les deux processus se sont déjà mis d'accord et sont prêts à effectuer le transfert.

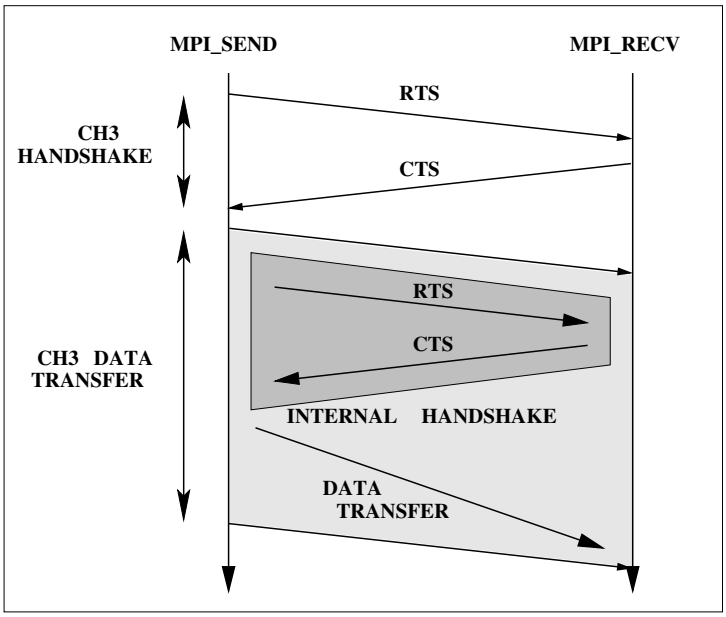


FIGURE 2.13 – Imbrication indésirable de protocoles rendez-vous.

Cependant, la mise en place d'un tel support demande de modifier le chemin critique assez haut dans la pile logicielle de MPICH2. En pratique, cela revient à court-circuiter le système des modules réseaux de NEMESIS pour fournir une implémentation de niveau *Device* pour ce qui concerne les communications réseaux (cf. discussion en section 2.5.1.1) et laisser la couche CH3 appeler directement la bibliothèque de communication bas-niveau du réseau cible. Les modules MX et NEWMADELEINE que j'ai implémentés utilisent de telles capacités, ce qui leur permet d'obtenir des performances intéressantes (cf. section 2.5.4.2). L'implémentation de MPI fournie par Intel® propose un mécanisme appelé *Tag Matching Interface* qui permet l'exploitation des capacités de *Tag Matching* de certains réseaux rapides (Myrinet, Infiniband par exemple) mais aucune information n'est véritablement disponible à ce sujet. De son côté, Open MPI implémente une PML *Point-to-Point Management Layer* qui fonctionne de façon similaire pour certains réseaux (dont MX/Myrinet).

2.5.4 Performances de MPICH2-NEMESIS

Tout ce que nous avons détaillé jusqu'à présent avait pour principal objectif d'arriver à diminuer au maximum le surcoût engendré par le passage de message, c'est-à-dire par la pile logicielle qu'il faut nécessairement traverser lorsqu'une application utilise une implémentation de MPI pour effectuer ses communications. Les performances sont donc un point important pour évaluer si NEMESIS a atteint ou non ses objectifs. Nous allons donc présenter dans cette section des résultats concernant surtout les communications intra-nœuds car elles sont la cible principale de NEMESIS. Les performances des communications inter-nœuds sont détaillées dans les articles suivants : [CI13], [CI14], [RI4], [CI11] et [CI9]). Nous allons cependant présenter des résultats inédits n'ayant jamais fait l'objet de publication jusqu'à présent en particulier concernant le support du *Tag Matching* dans les modules réseaux de NEMESIS.

Du point de vue de la configuration, MPICH2-NEMESIS est compilé avec l'option `-enable-fast` qui désactive la vérification des erreurs potentielles dans les paramètres passés aux fonctions MPI et active des optimisations de compilation. L'*inlining* de fonctions a également été activé. Pour LAM/MPI et Open MPI, la vérification d'erreurs ainsi que le support des processeurs hétérogènes ont été désactivés. Toutes les implémentations utilisées ont par ailleurs été compilées avec l'option `-O3`. Les diverses plates-formes utilisées seront décrites dans les sections appropriées.

2.5.4.1 Performance des communications intra-nœuds

Nous allons d'abord examiner les performances des communications intra-nœuds dans MPICH2-NEMESIS : latence et débit, mais également nombre d'instructions car cela constituait un élément important dans la justification de la conception de ce système de communication, ainsi qu'expliqué en debut de section 2.5. Les mesures présentées ont été obtenues sur une machine bi-processeur Opteron 280 cadencé à 2GHz avec deux cœurs par processeur et équipée de 2 Go de mémoire vive [24].

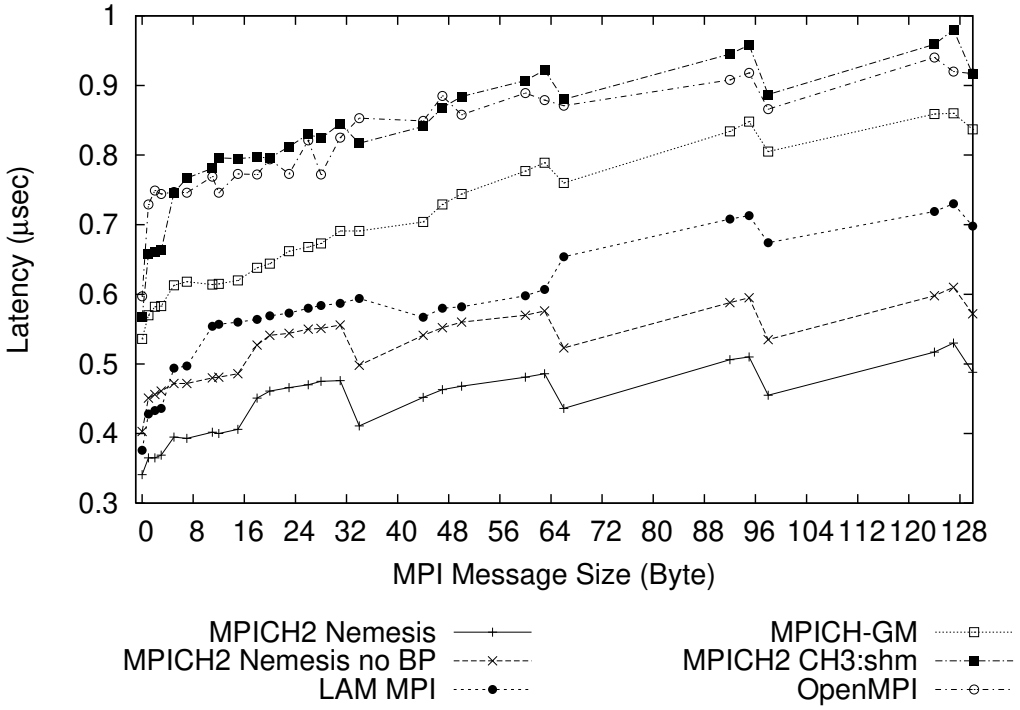
Évaluations point-à-point : latence et débit Nous présentons tout d'abord sur la figure 2.14 des mesures «classiques» de latence et de débit obtenues avec le programme de test NetPIPE [200]. Les différentes versions de MPI considérées sont les suivantes :

- MPICH2-NEMESIS avec les optimisations de contournement de la file de réception de CH3 (cf. section 2.5.2.5), et l'interface LMT (cf. section 2.5.2.3) ;
- MPICH2-NEMESIS sans ces optimisations («MPICH2-NEMESIS no BP» et «MPICH2-NEMESIS no LMT» dans les légendes ci-dessous) ;
- MPICH2 (version 1.0.3) avec le *channel shared-memory* (`shm`) de CH3 : ils'agit en effet de la méthode de communication la plus rapide dans MPICH2 avant l'arrivée de NEMESIS ;
- MPICH-GM (version 1.2.6..14b) avec le support des communications par mémoire partagée activé car à l'époque ce support offrait de très bonnes performances ;
- LAM/MPI (version 7.1.2) avec le support des communications par mémoire partagée activé ;
- Open MPI (version 1.1) avec le support des communications par mémoire partagée activé.

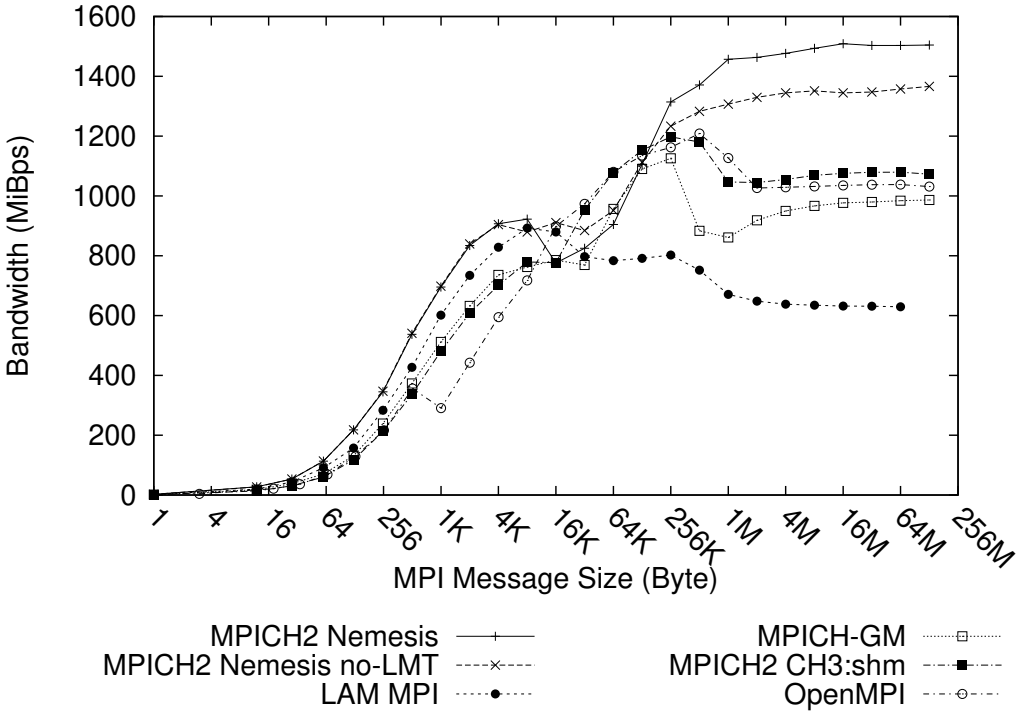
L'implémentation YAMPII a été exclue de l'évaluation car elle ne supportait pas les communications par mémoire partagée de façon efficace : c'est l'interface *loopback* des cartes réseaux Ethernet qui était mise à contribution, un peu à la manière des implémentations de première génération [25].

Les courbes de latence (figure 2.14(a)) montrent que l'optimisation qui consiste à court-circuiter les files de réception des messages postés/messages inattendus de CH3 améliore significativement la latence des communications. MPICH2-NEMESIS arrive à maintenir une latence de l'ordre de 500 ns jusqu'à une taille de messages de 128 octets et montre des performances supérieures à tous ses concurrents. Les performances de LAM/MPI restent encore très intéressantes (pour une implémentation de première génération) tandis celles de MPICH2 avec le *channel* `shm` sont assez faibles. Il est donc clair qu'il était nécessaire de revoir de façon drastique la gestion des communications bas-niveau dans MPICH2.

24. Je n'ai pas été en mesure de retrouver les informations concernant les mémoires-caches de cette machine.
25. Cette approche a déjà été mentionnée en section 2.4.1



(a) Latence



(b) Débit

FIGURE 2.14 – Performance des communications intra-nœuds pour différentes implémentations de MPI.

Les comparaisons de débit sont montrées sur lea figure 2.14(b). NEMESIS utilise une fonction de copie mémoire optimisée utilisant des opérations *store* non-temporelles qui évitent de passer par le cache, ce qui est utile dans le cas de messages dont la taille est supérieure à celle dudit cache. Nous pouvons constater que si l'interface LMT permet d'améliorer le débit des gros messages (256 Ko et plus), une pénalité apparaît pour les messages entre 16 et 256 Ko, pour partie imputable à l'utilisation d'un protocole de type rendez-vous. La version de LMT utilisée ici est celle de base et non celle reposant sur KNEM. De façon générale les performances de MPICH2-NEMESIS sont très satisfaisantes et souvent supérieures à celles de la concurrence. On remarquera que les bonnes performances de LAM/MPI pour la latence ne se maintiennent pas pour le débit.

Évaluation du nombres d'instructions Le nombre d'instructions est une métrique permettant d'évaluer la complexité du chemin critique d'émission/ réception des messages dans une implémentation de MPI. Le programme de test utilisé est décrit ci-dessous :

```
Process 0 (Maître):  Process 1 (Esclave):
MPI_Init();          MPI_Init();
for(NLOOPS)          for(NLOOPS)
 {                    {
  PAPI_Reset();        usleep(2);
  MPI_Send(...);       MPI_Recv(...);
  PAPI_Accum();
  usleep(4);
  PAPI_Reset();
  MPI_Recv(...);       MPI_Send(...);
  PAPI_Accum();
 }                    }
MPI_Finalize();      MPI_Finalize();
```

Nous mesurons le nombre d'instructions nécessaires pour l'envoi puis la réception bloquante d'un message d'une taille de 8 octets. Ces échanges se produisent au sein du communicateur `MPI_COMM_WORLD`. Afin d'éliminer le temps de scrutation (car variable), un appel à `usleep` est effectué afin de garantir la disponibilité du message au moment de l'appel à `MPI_Recv`. Les mesures sont effectuées en accèdant aux compteurs matériels des processeurs qui permettent de récupérer ce genre d'informations. La bibliothèque utilisée est PAPI [28]. La valeur de `NLOOPS` a été fixée à 10000 dans nos tests.

Les résultats sont indiqués dans la table 2.1 et montrent une fois encore que l'optimisation permettant de contourner les files de CH3 diminue drastiquement le nombre d'instructions nécessaires à la réception d'un message (sans surprise, cela n'a aucun effet sur l'émission). De plus, contrairement aux affirmations de Cray, il existait déjà à l'époque une implémentation capable de faire cette émission/réception de message en moins de 1000 instructions : LAM/MPI. Une fois encore, nous pouvons constater que le *channel* `shm` de MPICH2 n'exhibe pas des performances satisfaisantes. Ces mesures nous confortent d'une part sur le fait que la création de NEMESIS était nécessaire et d'autre part que les choix de conception ont été pertinents et l'implémentation de ces mêmes choix, efficace. Des résultats complémentaires sont disponibles dans [CI13], [CI14] et [RI4].

2.5.4.2 Performance des communications inter-nœuds

Nous montrons maintenant certains résultats obtenus dans le cas des communications inter-nœuds. Bien que NEMESIS a été conçu principalement pour obtenir des communications

TABLE 2.1 – Instructions nécessaires pour envoyer/recevoir un message de 8 octets.

Implémentation MPI	`MPI_Send`	`MPI_Recv`	Total
Open MPI	550	1,745	2,295
MPICH-GM	455	617	1,072
LAM/MPI	436	472	908
MPICH2 CH3 : shm	311	748	1,059
MPICH2-NEMESIS (sans *bypass*)	241	712	952
MPICH2-NEMESIS (avec *bypass*)	**241**	**259**	**500**

intra-nœuds très efficaces, il est cependant crucial que les échanges entre nœuds de calcul soient également performants. Les résultats concernant les technologies Ethernet/TCP et Myrinet/GM sont disponibles dans [CI13] avec notamment une comparaison avec le *channel* de MPICH2 dédié à GM : GASNet. Le cas du réseau Infiniband est traité dans l'article consacré au module NEWMADELEINE ([RI4]) qui montre des comparaisons entre ce dernier, Open MPI et MVAPICH2. Étant donné que NEWMADELEINE est une bibliothèque générique de communication, elle introduit un surcoût, ce qui a un impact sur la latence des petits messages. Le débit pour les gros messages est en revanche plus satisfaisant. [RI4] montre également les bénéfices du support du multirail car NEWMADELEINE est capable d'exploiter plusieurs réseaux rapides simultanément pour agréger les débits.

Nous allons maintenant nous concentrer sur le support du *Tag Matching* qui permet *quasiment* d'obtenir une version *Device* de MPICH2 au niveau des communications réseaux, ce qui rapproche MPICH2-NEMESIS des implémentations spécialisées pour un réseau particulier qui réimplémentent leur propre ADI pour obtenir de meilleures performances. Le support du *Tag Matching* est effectif pour deux modules réseaux : dans le module Myrinet/MX (MX ayant remplacé progressivement GM, cette bibliothèque n'était plus utilisable sur les nouvelles générations de cartes Myrinet) et dans le module NEWMADELEINE, qui permet de pouvoir exploiter les réseaux Infiniband dans MPICH2 sans avoir à utiliser MVAPICH2 [26].

Environnement expérimental Les plates-formes utilisées sont différentes selon les réseaux. Pour Infiniband, nous avons mené les expériences sur la grappe Plafrim qui possède 32 nœuds interconnectés par des HCA Mellanox modèle MT26428 (ConnectX IB QDR). Les nœuds sont composés [27] de deux processeurs Intel® Xeon Nehalem X5550 avec 4 cœurs cadencés à 2,66 GHz chacun. Au niveau de la mémoire, 8 Mo de cache L3 sont disponibles par processeur et un total de 24 Go de RAM (DDR3 à 1,33 GHz) équipe chaque nœud. Enfin le système d'exploitation utilisé est Linux (noyau 2.6.27.39).

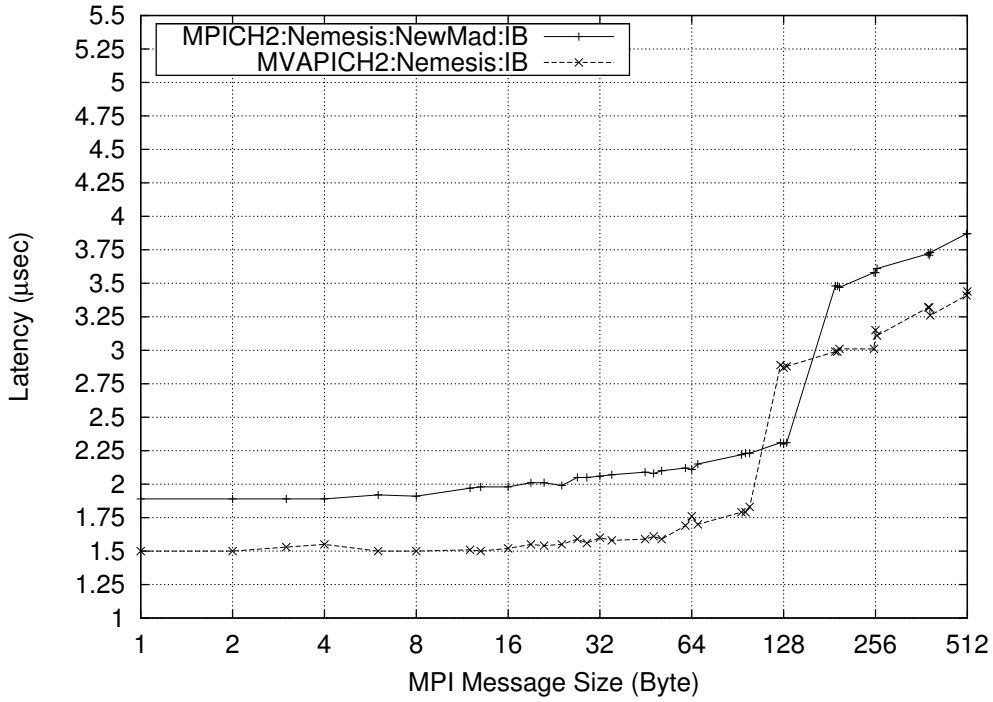
Dans le cas de Myrinet, la machine utilisée est Borderline, composée de 10 nœuds interconnectés par un réseau Myrinet 10G-PCIE-8A-C. Chaque nœud possède 4 processeurs AMD Opteron 2218 quadi-cœurs cadencés à 2,6 GHz. 2 Mo de cache L2 par cœur sont présents (il n'y a pas de cache L3), ainsi que 32 Go de mémoire (800MHz).

Évalution des communications point-à-point

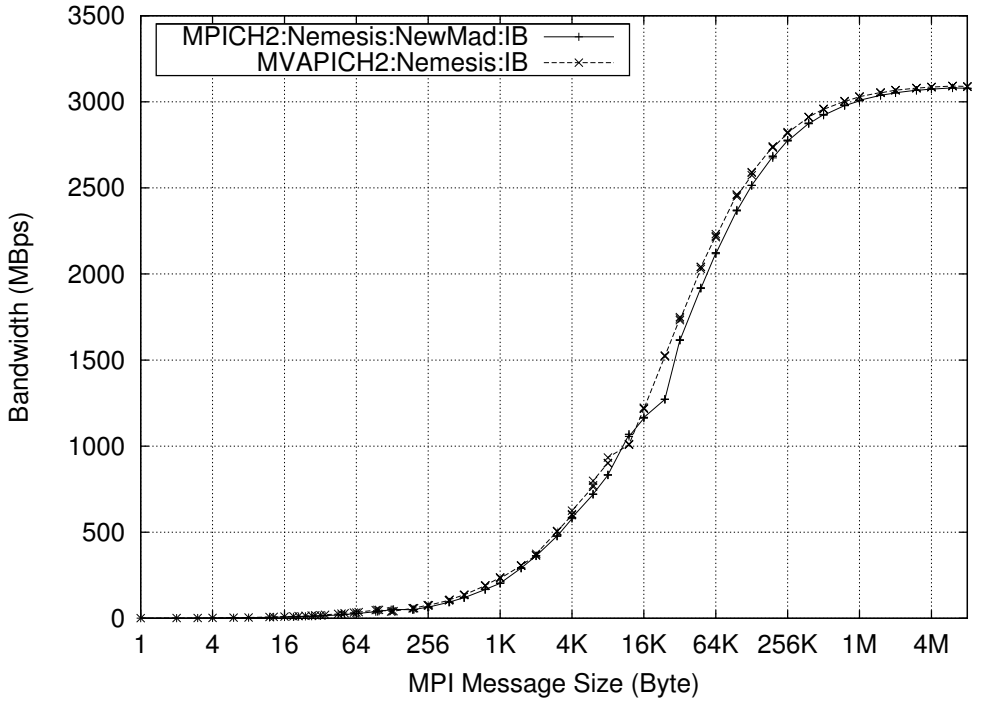
- Cas du réseau Infiniband : La figure 2.15 montre la comparaison entre notre module réseau NEWMADELEINE exploitant le réseau Infiniband et l'implémentation MVAPICH2. La différence de latence entre les deux implémentations est de l'ordre de 20% en faveur de MVAPICH2 et pour le débit, les deux implémentations atteignent des niveaux similaires de performances.

26. À partir du moment où l'on souhaite rester dans l'«univers» mpich, cela va sans dire.
27. Au moment de la réalisation des mesures.

Il faut cependant rappeler que NEWMADELEINE introduit un surcoût par rapport à une utilisation directe de la bibliothèque bas-niveau exploitant Infiniband mais que cela permet la fourniture de mécanismes d'optimisation, comme l'agrégation des messages ou le support du multirail.



(a) Latence



(b) Débit

FIGURE 2.15 – Performances point-à-point avec le réseau Infiniband

- Cas du réseau Myrinet/MX : La figure 2.16 montre les performances respectives de trois implémentations de MPI : MPICH2-NEMESIS avec son module réseau MX, Open MPI avec son support pour Myrinet/MX configuré pour utiliser les capacités de *Tag matching* dans MX et enfin l'implémentation fournie par le fabricant des cartes réseau Myrinet : Myricom. Les trois implémentations affichent des latences très proches avec un léger avantage pour l'implémentation de Myricom, suivie par MPICH2-NEMESIS avec le module MX et enfin Open MPI. Les débits sont identiques pour les messages jusqu'à 32Ko qui est le seuil de rendez-vous dans MX. Entre 32Ko et 1Mo, MPICH2-MX offre de meilleures performances que MPICH2-NEMESIS et Open MPI qui sont identiques. À partir de d'une taille de 1Mo, les trois implémentations sont équivalentes. Ces résultats montrent qu'il est possible d'optimiser les transferts inter-nœuds dans MPICH2-NEMESIS (qui est une implémentation générique) pour atteindre des niveaux de performance proches, voire équivalents, à ceux d'implémentations spécialisées.

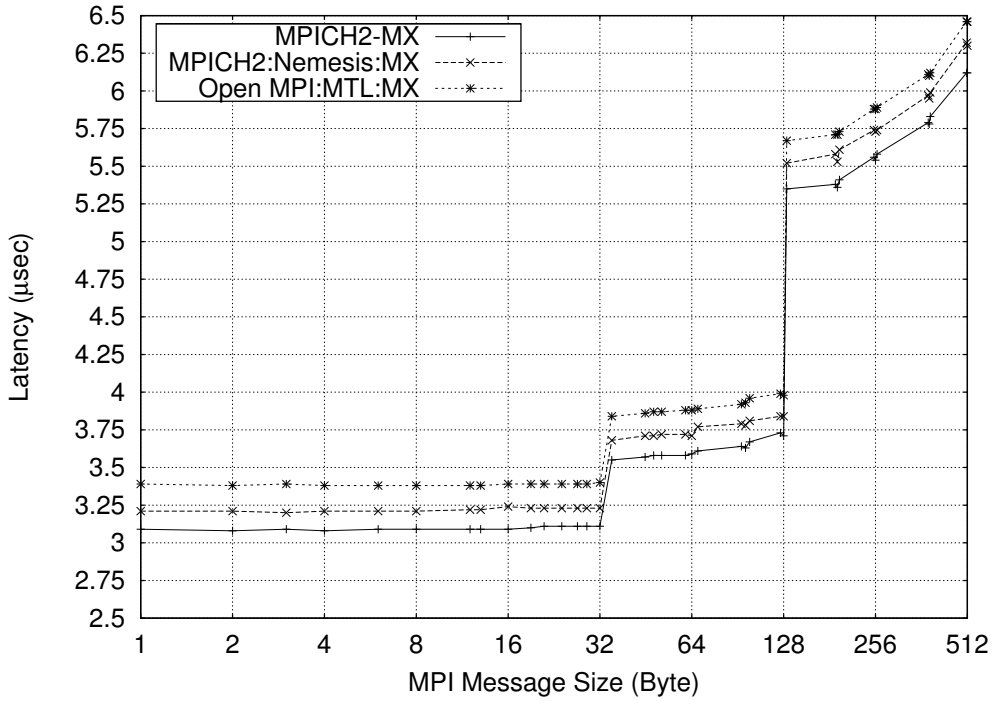
Évaluation avec Les *NAS Parallel Benchmarks* Outre les comparaisons de performance pour les opérations point-à-point, nous avons également évalué le comportement de MPICH2-NEMESIS avec des tests plus significatifs. Pour cela, nous avons décidé d'utiliser huit noyaux *NAS Parallel Benchmarks* (NAS) [9], version 3.3. Les versions MPI «pure» des noyaux BT, BG, EP, FT, IS, LU, MG et SP ont été employées pour mener les expériences. Aussi bien dans le cas de Myrinet que d'Infiniband, le cache d'enregistrement des tampons mémoire a été activé de qui permet d'obtenir de meilleures performances. Différentes classes de NAS sont évaluées : A, B, C et D. Pour chaque classe, le nombre de processus doit être choisi pour correspondre à la taille du problème. Comme nous n'avions pas assez de nœuds disponibles pour ne lancer qu'un unique processus MPI par nœud et nous placer dans une situation où seules des communications inter-nœuds sont possibles, nous avons dû évaluer les performances avec des cas où plusieurs processus MPI se partagent un même nœud. Cependant, nous avons choisi une politique de placement des processus de type *Round-Robin* [28] ce qui permet de diminuer significativement les communications intra-nœuds au profit des communications inter-nœuds, comme le montre la table 2.2.

	Classe A		**Classe B**				**Classe C**			**Classe D**
	8/9	16	8/9	16	32/36	64	16	32/36	64	64
BT	100%	100%	100%	100%	100%	66%	100%	100%	66%	66%
CG	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
EP	100%	40%	100%	40%	17%	7%	40%	17%	7%	7%
FT	100%	94%	100%	94%	91%	89%	94%	91%	89%	89%
IS	100%	93%	100%	93%	90%	89%	93%	90%	89%	89%
LU	100%	100%	100%	99%	70%	50%	100%	70%	50%	50%
MG	100%	100%	100%	100%	40%	66%	100%	30%	59%	50%
SP	100%	100%	100%	100%	100%	66%	100%	100%	66%	66%

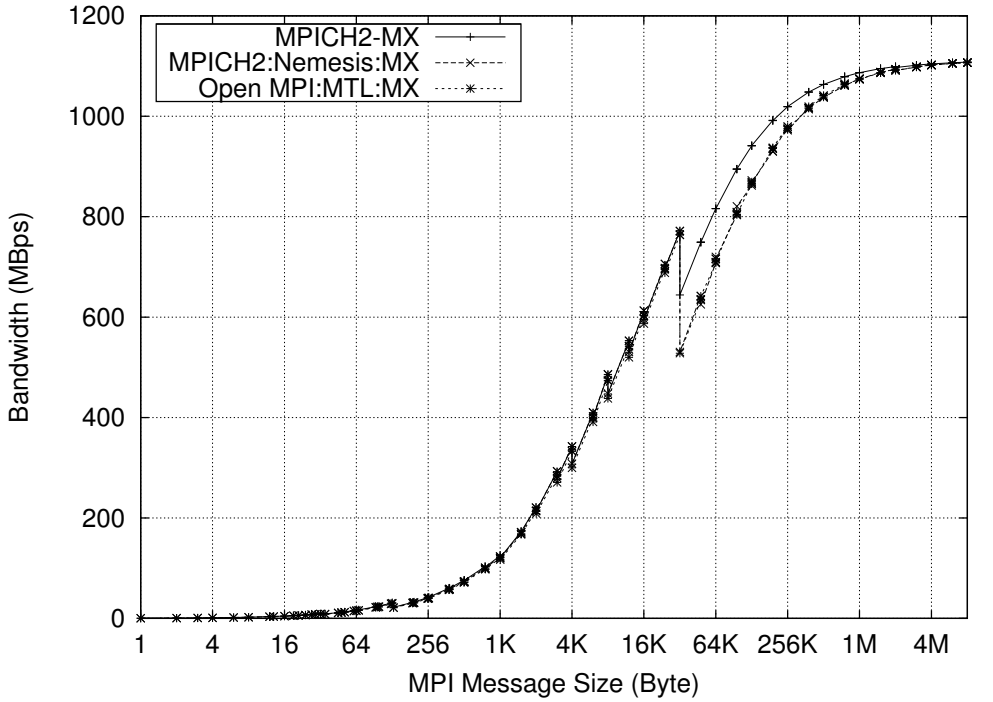
TABLE 2.2 – Ratio de communications réseaux des noyaux NAS en fonction de la classe et du nombre de processus (politique de placement *Round-Robin*).

Les figures 2.17 et 2.18 montrent les résultats pour les noyaux NAS pour les quatre classes considérées (A, B, C et D), avec un nombre de processus compris entre 8 et 256 pour Infiniband, 8 et 64 pour Myrinet (la taille de la grappe ne permet pas de lancer plus de 160 processus et je n'ai pas pu la réserver entièrement). Ces résultats confirment ceux obtenus par des

28. Dont la définition est donnée en section 3.1.2.3.



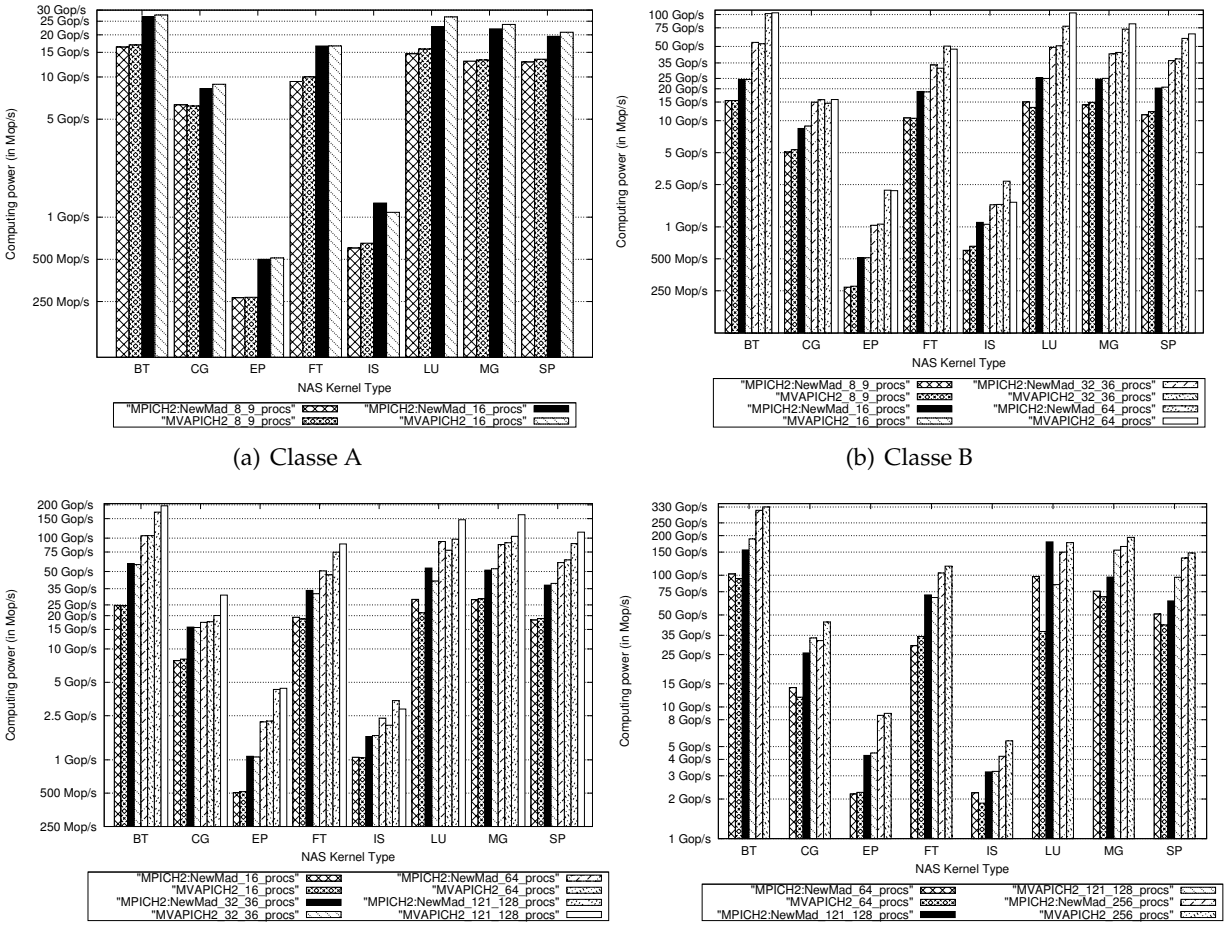
(a) Latence



(b) Débit

FIGURE 2.16 – Performances point-à-point avec le réseau Myrinet/MX

tests plus basiques de type point-à-point : MPICH2-NEMESIS offre des performances équivalentes à celles d'implémentations spécialisées. Quelques tests montrent même des cas où notre implémentation est plus rapide, mais comme le taux de communications inter-nœuds n'est pas de 100%, la différence provient vraisemblablement du support des communications intra-nœuds (le cas de EP, *Embarrassingly Parallel*, est spécifique car peu de communications sont impliquées).



(a) Classe A

(b) Classe B

(c) Classe C

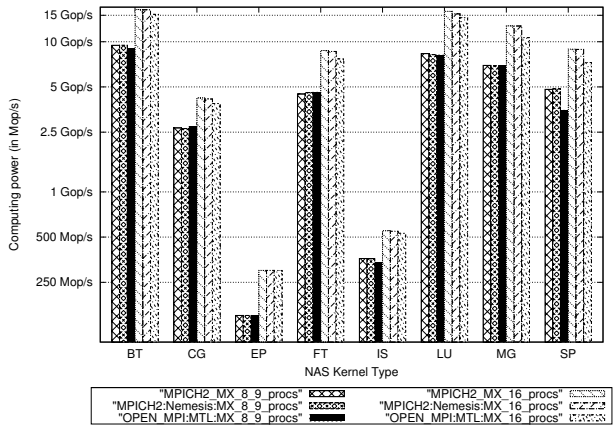
(d) Classe D

FIGURE 2.17 – NAS Parallel Benchmarks sur Infiniband

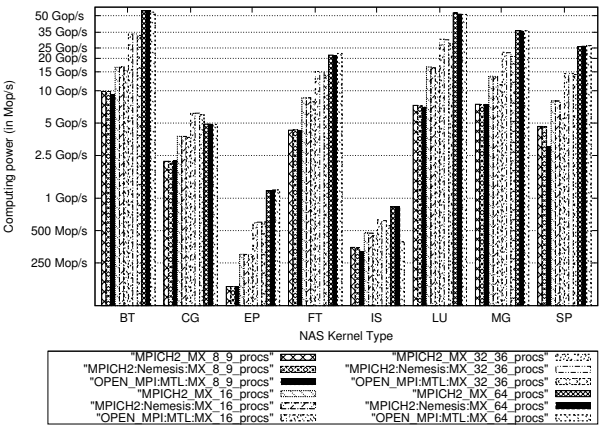
2.5.5 Bilan critique de MPICH2-NEMESIS

2.5.5.1 Ce que NEMESIS n'est pas devenu ...

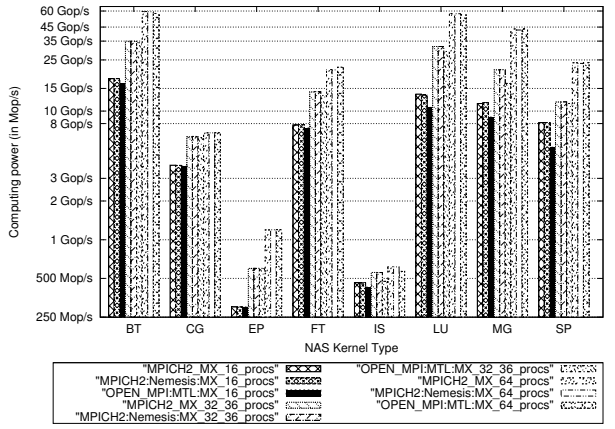
Après avoir présenté NEMESIS et son intégration dans MPICH2 puis avoir évoqué certains résultats, il est temps de tirer un bilan du travail effectué et des enseignements retirés. Il est indéniable que les objectifs initiaux de NEMESIS n'ont pas été complètement tenus. En effet, NEMESIS était destiné à la base à devenir un système de communication *générique*, utilisable pour l'implémentation et le support de divers modèles de programmation : passage de messages, langages basés sur un espace d'adressage global (*GAS languages*), etc. Son intégration dans MPICH2 devait initialement démontrer son potentiel en termes de performances. Au final, le destin de NEMESIS restera étroitement lié à celui de MPI, et de MPICH en particulier.



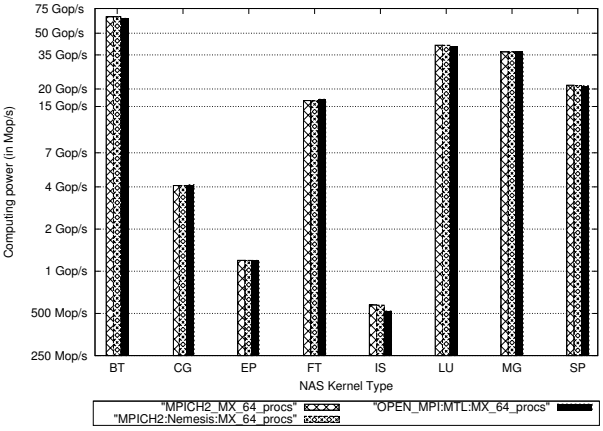
(a) Classe A



(b) Classe B



(c) Classe C



(d) Classe D

FIGURE 2.18 – NAS Parallel Benchmarks sur MX/Myrinet

Pour qu'il en soit autrement, il aurait fallu sans doute plus de ressources et de personnes ce qui n'était pas forcément le plus évident à obtenir à cette époque.

Si l'intégration de NEMESIS dans MPICH a été un franc succès, je regrette que nous ne soyions pas allés assez loin. Idéalement, NEMESIS n'aurait pas dû être un *channel* mais un *device*, au même titre que CH3 et aurait dû remplacer cette dernière. En dépit de modifications importantes dans cette couche (cf. les diverses optimisations intra-nœud et inter-nœuds), une refonte totale de l'ADI3 n'a pas été effectuée, ce qui est un travail d'une ampleur assez conséquente. J'ai quitté l'équipe de développement de MPICH après 18 mois de travail post-doctoral et peut-être sans ce départ, les choses auraient été différentes. Cette refonte de l'ADI aura lieu, mais dix ans plus tard, avec l'arrivée de CH4.

Le dernier regret que je formulerai aura été de ne pas avoir transféré plus de multithreading dans MPICH2-NEMESIS. En effet, venant d'une équipe où cela occupait une place prépondérante, et avec à mon actif une des seules implémentations de première génération supportant *véritablement* le niveau `MPI_THREAD_MULTIPLE`, il aurait été sans doute très intéressant de favoriser la prise en compte de cet aspect dans MPICH2, surtout au regard du développement des modèles hybrides de programmation quelques années plus tard. Même si nous avons fait des travaux en ce sens dans le cadre de l'Équipe-Associée Inria MPI-Runtime ([CI11]), cela ne s'est pas traduit par quelque chose de véritablement systématique dans l'implémentation. Il est d'ailleurs intéressant de constater que cette thématique s'est fortement développée à Argonne depuis quelques années, comme en témoigne le développement d'Argobots [179].

2.5.5.2 . . . et ce qu'il a été.

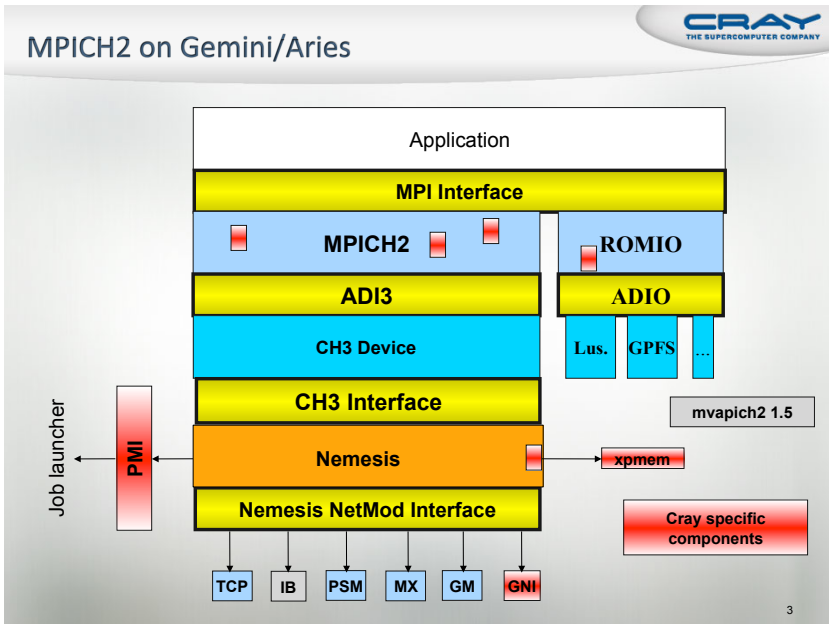
Tout d'abord, il me semble important de rappeler que le système de communication NEMESIS, ainsi que ses éléments satellites (KNEM) ont occupé une place importante dans le paysage du HPC des quinze dernières années. En effet, NEMESIS est devenu depuis 2007 et jusqu'à 2016 le *channel* de communication par défaut de MPICH2 puis MPICH3 [29]. Ses performances de tout premier plan (faibles latences et hauts débits), couplées à la relative facilité d'intégration de nouveaux modules réseaux ont permis cette longévité dans un domaine où l'obsolèscence marque de son sceau le matériel et les logiciels. La base d'utilisateurs de mes travaux a donc été considérable, car outre MPICH2, de nombreuses versions dérivées ont adopté NEMESIS comme système de communication de bas niveau. Cela a notamment été le cas de la version dédiée aux réseaux Infiniband, MVAPICH2 mais également le cas de versions constructeurs de MPI, comme celles fournies par Intel®, par Microsoft pendant un temps (MS-MPI) et par Cray (cf. figures 2.5.5.2 et 2.5.5.2). Ainsi, NEMESIS a été disponible sur les séries XE-6/XK-7 équipées du réseau d'interconnexion propriétaire de Cray, Gemini grâce à un module réseau dédié [165] puis sur les séries XC avec le réseau Aries [30]. Au final, NEMESIS restera comme un travail utile pour la communauté et qui aura permis de pousser relativement loin le degré de sophistication qu'une implémentation de MPI est capable d'atteindre.

2.6 Conclusion

Quel bilan est-il possible de tirer de ce chapitre ? Le premier constat que nous pouvons effectuer, c'est que le problème des la gestion des topologies matérielles a été identifié très tôt dans la vie de MPI et que les implémentations ont proposé des mécanismes pour les prendre

29. Ce statut a perduré jusqu'à la version 3.2.1 de MPICH, mais NEMESIS reste toutefois disponible dans l'actuelle 3.3.

30. NEMESIS est encore disponible avec la série XC-40 toujours en vente à l'heure actuelle.



Features of the Cray MPI library

- **Cray MPI uses MPICH3 distribution from Argonne**
  - Provides a good, robust and feature rich MPI
    - Well tested code for high level features like MPI derived types.
  - Cray provides enhancements on top of this:
    - low level communication libraries
    - Point to point tuning
    - Collective tuning
    - Shared memory device is built on top of Cray XPMEM
- **Cray MPI uses the NEMESIS module**
  - SMP aware
  - communication between ranks on the same node uses shared memory
    - rather than the Aries network
    - gives higher performance
- **Supported under all Cray PrgEnv**
  - The compiler/linker wrappers takes care of including the correct header and libs

en compte. D'un certain point de vue, le problème de la gestion des multiples niveaux hiérarchiques dans les grappes de nœuds NUMA est très similaire au problème des niveaux dans les grilles de calcul ou dans les grappes de grappes.

Cependant, la pratique est différente car les solutions exploitant plus de deux niveaux hiérarchiques (i.e. intra-nœud et inter-nœuds) seront relativement rares et l'essentiel des travaux va se concentrer sur l'optimisation des opérations de communication collectives. Quelques solutions plus sophistiquées ont été proposées, mais elle n'étaient destinées qu'à des architectures particulières et pas des plus répandues, ce qui a pu nuire à leur exposition et leur diffusion.

Également, on peut constater une certaine multiplicté dans les solutions proposées sans véritablement d'approche globale pour répondre au problème. L'arrivée de grappes de nœuds multicœurs va forcer les implémentations à faire leur *aggiornamento* et ce qui était vu comme quelque chose d'optionnel (la gestion de plus de deux niveaux) va s'imposer comme un défi d'importance à relever : la hiérarchie s'est étendue vers l'intra-nœud, mais les conséquences de cette extension vont mettre quelques années avant de pleinement se révéler. Par exemple, la problématique du placement de processus applicatifs dans ces architectures va mettre quelques années avant d'émerger (cf. chapitre 3).

Durant cet intervalle de temps, les implémentations vont encaisser le choc et permettre aux applications d'exploiter les performances offertes par cette nouvelle génération de machines : MPICH2-NEMESIS est un exemple fort éloquent. Cependant, cette approche opaque va rapidement connaître ses limites et s'il est toujours primordial de mettre au point les mises en œuvre de MPI les plus performantes possible, la recherche de solutions allant au-delà de ces implémentations va être indispensable : la prise en compte d'éléments connexes dans l'écosystème HPC va permettre de répondre plus avant au problème posé.

Chapitre 3

Vers une meilleure prise en compte de la localité dans l'écosystème HPC

Le chapitre précédent nous a donné l'occasion de restituer le contexte de mes travaux ainsi que d'en exposer une première série aboutissant à des communications performantes dans les grappes de machines hiérarchiques multicœurs. À partir de 2008, et toujours dans l'optique d'optimiser les communications, je me suis intéressé à des aspects connexes aux implémentations de MPI et surtout plus indépendants de ces dernières. En effet, il me paraissait de plus en plus évident que si les implémentations de MPI étaient arrivées à un point de sophistication avancé, des marges de manœuvre étaint encore exploitables en étudiant les interactions de ces implémentations avec l'environnement d'exécution des applications. Ces interactions sont multiples : elles couvrent des aspects importants et qui avaient été peu exploités jusqu'alors et notamment la répartition, le déploiement et le lancement des processus applicatifs dans l'écosystème HPC.

Comme je le développerai dans les sections suivantes, la répartition des processus applicatifs est une problématique ancienne mais qui a regagné de l'intérêt avec l'avènement des nœuds hiérarchiques multicœurs complexes présentant des effets NUMA. Ces derniers sont susceptibles d'affecter les performances des communications au sein des différents nœuds et *in fine*, de l'application tout entière à partir du moment où le volume des communications intra-nœud prend une part significative du volume globale de communication. Il s'agit d'un levier relativement efficace et qui a pour avantage de ne pas avoir à modifier les applications existantes, où alors marginalement.

Ces travaux, qui débutent par un article concernant les politiques de placement de processus MPI dans les nœuds multicœurs ([CI10]), va enclencher une dynamique importante puisque le thème de la *localité* va devenir un axe majeur de recherche au sein de l'EPC Runtime puis devenir le thème principal des recherches de l'EPC TADaaM. De plus, ces travaux se sont inscrits dans un contexte très favorable car ils ont entraîné de nombreux travaux similaires menés au sein d'autres équipes. En particulier, les logiciels résultants de ces recherches sont devenus des références dans leur domaine.

La section suivante (3.1) va nous permettre de définir les liens entre *localité* et *placement*. Nous formaliserons le problème et indiquerons un ensemble de solutions algorithmiques existantes. La section 3.2 détaillera plus avant cette problématique dans le contexte particulier du passage de message et montrera un panel de solutions existantes, aussi bien pour les implémentations de première que de seconde génération. La section 3.3 exposera l'approche retenue avec les multiples contributions au domaine résultant de nos travaux et constituant un environnement d'allocation, de déploiement et d'exécution. Enfin, la section 3.5 dressera un bilan et montrera notamment en quoi notre approche est généralisable à d'autre modèles de programmation.

3.1 Localité de données et placement de processus applicatifs

Nous allons maintenant préciser la notion de localité avant de montrer sur un exemple synthétique l'impact que peut avoir une meilleure prise en compte de cette dernière dans une application parallèle. Nous formaliserons rapidement le problème avant de faire un état de l'art non exhaustif.

3.1.1 Localité et placement : même combat ?

3.1.1.1 Importance de la localité

L'exploitation efficace des machines multicœurs hiérarchiques complexes constitue un véritable défi pour les développeurs d'applications et les modèles de programmation en particulier. La passage à l'échelle est un facteur déterminant et des solutions permettant de faire face à la multiplication des unités de calcul et à la croissance substantielle de la consommation énergétique doivent être trouvées. Une réponse (partielle toutefois) réside dans la l'amélioration de la *localité des données* dans les applications parallèles, c'est-à-dire la façon dont leurs données sont placées en mémoire, accédées et déplacées par les unités physiques de calcul de l'architecture cible sous-jacente. Cela rentre en forte cohérence avec le comportement des applications parallèles justement, puisque ces dernières distribuent leurs données parmi les différentes unités logiques de calcul (par exemple des processus ou encore des threads) les composant. Ces unités logiques de calcul accèdent ou s'échangent des données pendant l'exécution des applications mais pas forcément régulièrement (aussi bien d'un point de vue temporel que quantitatif).

Ces accès et échanges de données peuvent par conséquent être optimisés en vue d'exploiter au mieux le matériel disponible, par exemple en plaçant des **unités logiques d'exécution** communicant beaucoup sur des **unités physiques d'éxécution** proches physiquement parlant. Ce faisant, les temps de calcul sont réduits et cela diminue le temps total d'exécution d'une application d'autant plus qu'elle communique. La consommation énergétique pourrait également s'en trouver réduite, comme le note le *roadmap* IESP [56] :

> *Since much of the power in an Exascale system will be expended moving data, both locally between processors and memory as well as globally, the X-stack must provide mechanisms and APIs for expressing and managing data locality. These will also help minimize the latency of data accesses.*

3.1.1.2 Lien entre localité et placement

Ainsi la **localité des données** peut être améliorée via un **placement** adéquat des unités logiques d'exécution (les processus) sur les unités physiques d'éxécution (les cœurs ou threads matériels des processeurs). Fondamentalement, cela revient à mettre en correspondance deux informations :

- le comportement de l'application en termes d'accès ou d'échanges de données. Ce comportement applicatif décrit une **topologie virtuelle** ;
- les caractéristiques de l'architecture cible qui exécute ladite application, ce que l'on peut appeler la **topologie physique**.

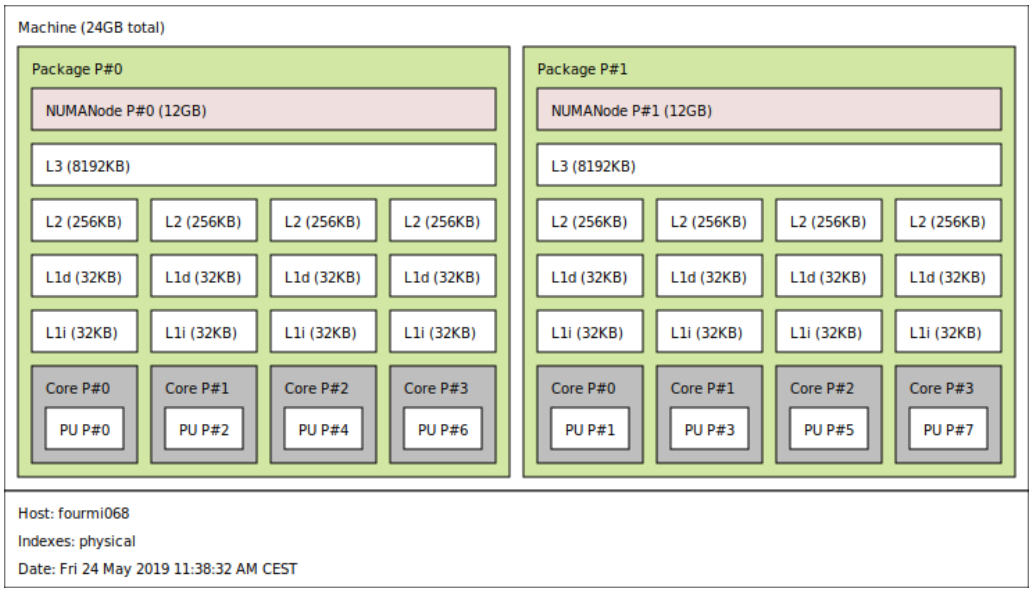
L'hypothèse importante sur laquelle nous allons nous appuyer a déjà été formulée en quelque sorte au chapitre 2 (section 2.2.2) : des unités physiques d'éxécution proches vont utiliser des ressources mémoire de plus en plus performantes et donc les communications s'en trouveront accélérées. Au final, le problème que nous allons devoir résoudre est celui d'une mise en correspondance (ou *mapping*) de la topologie virtuelle avec la topologie matérielle, c'est-à-dire

trouver **une répartition adéquate (selon une métrique donnée) de l'ensemble de flots d'exécution de l'application parallèle (processus et/ou threads) sur l'ensemble des ressources de calcul de la plate-forme matérielle cible (cœurs, threads matériels, etc.)**.

3.1.2 Illustration de la problématique par l'exemple

3.1.2.1 Un exemple synthétique : échange de jeton dans des anneaux

Nous allons illustrer la problématique du placement de processus à l'aide d'un exemple synthétique. Dans ce programme MPI, la situation est la suivante : nous lançons un ensemble de 64 processus sur une machine composée de 8 nœuds de calcul qui sont tous reliés au même *switch*. De ce fait, nous nous restreignons à l'observation du placement intra-nœud sur les performance puisque la topologie réseau est plate. Chaque nœud de calcul possède 2 *packages* avec un cache de niveau L3 et 4 cœurs. Un cache L2 et un cache L1 sont disponibles pour chaque cœur de calcul. La figure suivante montre la configuration matérielle d'un nœud utilisé dans notre exemple.



En ce qui concerne l'application, elle est assez simple et n'implique pas de calculs et uniquement des communications, ce qui est bien évidemment fort peu réaliste. Là encore, nous mettons l'accent sur l'amélioration possible des communications. Une application qui ne communique pas, ou très peu, n'aura qu'un intérêt limité à mettre en place une politique de placement sophistiquée. Le schéma de communication retenu est celui d'un jeton qui circule parmi les processus : les processus sont regroupés par 8 et le jeton circule de voisin en voisin (i.e le processus P_i envoie le jeton au processus p_{i+1}) dans chaque groupe en parallèle, les communications étant initiées par un *leader* choisi au sein de chaque anneau (le processus de rang minimal par exemple). Ces communications utilisent des opérations point-à-point de de type `MPI_Send` et `MPI_Recv`. Une fois le jeton revenu à son point de départ, tous les *leaders* s'échangent leurs jetons respectifs avec une opération collective de type `MPI_Allgather`. Cela permet d'avoir des communications entre les anneaux de processus au lieu d'avoir une application avec des parties complèment indépendantes. La figure 3.1 montre le schéma de communication retenu.

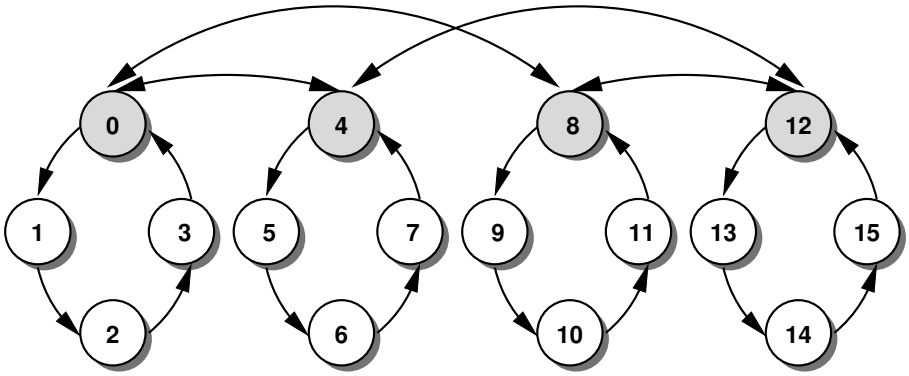


FIGURE 3.1 – Schéma de communication pour quatre anneaux avec quatre processus chacun (les cercles grisés sont les *leaders*).

3.1.2.2 Représentativité de l'exemple

Cet exemple est très simple, mais n'est cependant pas totalement artificiel. En effet, si l'on compare son schéma de communication à ceux d'applications de test connues comme les NAS parallel benchmarks [9] et plus traditionnellement utilisées pour des évaluations de performances, on remarquera des similitudes. Il y a moins de communications globales dans notre exemple mais les communications locales, c'est-à-dire impliquant un processus et ses voisins proches (au sens du numéro de rang qui leur est attribué) suivent un schéma proche. C'est ce que montre la figure 3.2.

3.1.2.3 Définition et analyse de politiques de placement par défaut

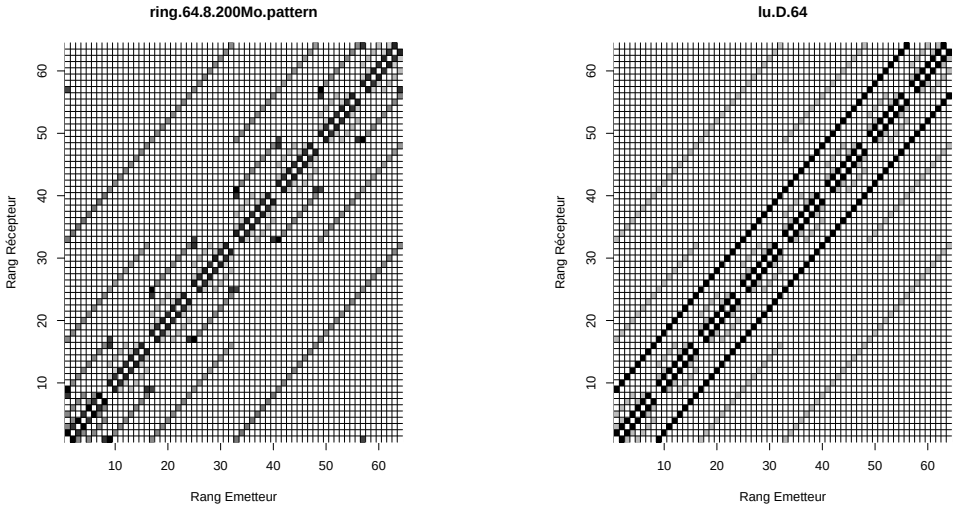
Nous allons maintenant décrire différentes politiques de placement qui vont nous servir de points de comparaison. Il existe en effet des politiques de placement assez rudimentaires qui sont fréquemment appliquées par défaut par les gestionnaires de processus et les lanceurs de programmes dans les implémentations de MPI. Nos hypothèses sont les suivantes :

— on doit placer P processus, P_i avec $i \in [0, \ldots, P-1]$. Il s'agit d'une numérotation linéaire, ce qui correspond à la plupart des cas dans les applications MPI avec un nombre statique de processus. Ce numéro, dit *numéro de rang* (cf. section 3.2.1.1) est contextuel, c'est-à-dire dépendant du communicateur MPI auquel appartient le processus en question et au sein duquel les communications sont effectuées.

— on dispose de N nœuds de calcul N_j avec $j \in [0, \ldots, N-1]$. Là encore, on utilise une numérotation linéaire, ce qui est toujours possible via une renumérotation si ce n'est pas le cas. De plus, nous allons également supposer que les nœuds sont structurés de façon **homogène**, c'est-à-dire que chaque nœud de calcul possède K cœurs C_k où $k \in [0, \ldots, K-1]$. Ces cœurs sont donc numérotés linéairement et identiquement d'une machine à l'autre. Au total, nous disposons de $N \times K$ cœurs de calcul. Il devient alors possible de numéroter ces cœurs de façon absolue dans la configuration utilisée avec la formule suivante :

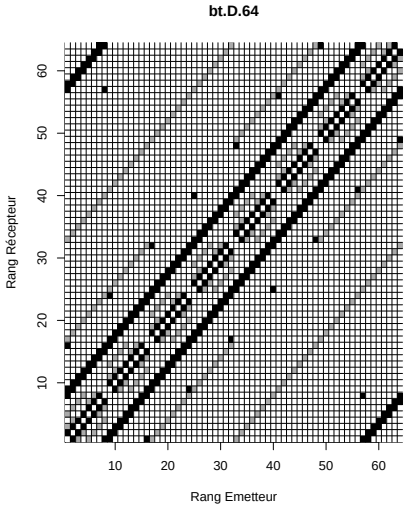
$$indice\_absolu = K \times n + indice\_local$$

C'est-à-dire que le cœur C_{indice_absolu} est le cœur C_{indice_local} du nœud N_n. À noter que cette numérotation des cœurs de calcul est une numétotation **logique** qui peut différer de la numéroation **physique** des cœurs. Il n'est pas conseillé d'utiliser cette numérotation car elle est susceptible de changer au cours du temps, avec une mise à jour du BIOS de la machine par exemple. La numérotation logique permet de mettre en place de

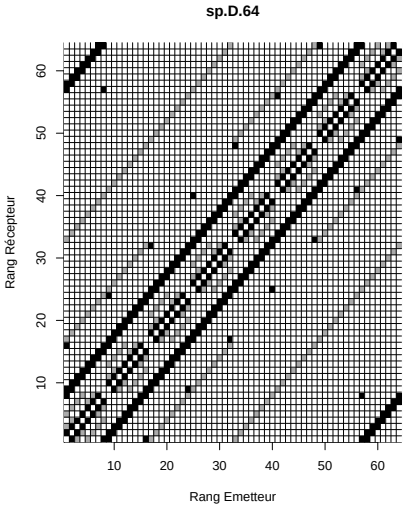


(a) Exemple de l'anneau

(b) NAS-LU



(c) NAS-BT



(d) NAS-SP

FIGURE 3.2 – Comparaison de schémas de communication : kernels NAS et échange de jeton

façon pérenne et portable des politiques de placement de processus. Une telle numérotation peut être fournie et gérée par un logiciel permettant d'accéder aux informations de topologie matérielle, comme hwloc [CI7] que nous évoquerons ultérieurement (cf. section 3.3.2) et qui va devenir un élément majeur dans la gestion des machine multi-cœurs hiérarchiques à partir des années 2010 et sur lequel nous allons travailler.

Dans notre exemple, nous avons donc $N = K = 8$ et $P = 64 = N \times K$.

Politique de placement séquentielle La première politique simple placement est appelée **séquentielle** (également connue sous le nom de *Sequential/Packed/Serial Ranking*). Avec cette politique, on cherche à affecter des processus pour remplir un nœud avant de passer au sui-

vant [1]. Ainsi, Des processus dont les rangs sont consécutifs ont une chance accrue d'être placés sur une même machine et donc de pouvoir utiliser des communications intra-nœuds. Dans le cas général, la politique de placement séquentielle est décrite par la fonction suivante :

$$\sigma_{seq} \ : \left| \begin{array}{ccc} \{0,\ldots,P-1\} & \mapsto & \{0,\ldots,N\times K-1\} \\ i & \to & i \pmod{(N\times K)} \end{array} \right.$$

S'il n'y a pas d'*oversubscribing* de ressources et qu'elles sont toutes utilisées, alors il y a une bijection entre les numéros de processus et les numéros de cœurs. c'est-à-dire :

$$P = N \times K$$

et par conséquent :

$$i \pmod{(N\times K)} = i \quad | \quad \forall i \in \{0,\ldots,N\times K-1\}$$

alors la fonction σ_{seq} est en fait la permutation identité :

$$\sigma_{seq} \ : \left| \begin{array}{ccc} \{0,\ldots,N\times K-1\} & \mapsto & \{0,\ldots,N\times K-1\} \\ i & \to & i \end{array} \right.$$

Dans notre exemple, les processus P_0, P_1, ..., P_7 seront placés sur le nœud 0, les processus P_8, P_9, ..., P_{15} sur le nœud 1 et ainsi de suite.

Politique de placement *Round-Robin* La seconde politique simple de placement est appelée ***Round-Robin***. Avec ce placement, les processus sont répartis successivement selon leur rang sur les différents nœuds de calcul. Des processus avec des rangs consécutifs seront donc potentiellement placés sur des machines différentes, ce qui implique des communications inter-nœuds. Dans le cas général, la politique de placement *Round-Robin* est décrite par la fonction suivante :

$$\sigma_{rr} \ : \left| \begin{array}{ccc} \{0,\ldots,P-1\} & \mapsto & \{0,\ldots,N\times K-1\} \\ i & \to & K \times ((i \pmod{N\times K}) \pmod N) + \frac{(i \pmod{N\times K})}{N} \end{array} \right.$$

À nouveau, en l'abscence d'*oversubscribing* de ressources et qu'elles sont toutes utilisées, alors il y a une bijection entre les numéros de processus et les numéros de cœurs. Nous avons alors :

$$P = N \times K$$

et par conséquent :

$$i \pmod{(N\times K)} = i \quad | \quad \forall i \in \{0,\ldots,N\times K-1\}$$

Alors la fonction σ_{rr} devient la permutation suivante :

$$\sigma_{rr} \ : \left| \begin{array}{ccc} \{0,\ldots,N\times K-1\} & \mapsto & \{0,\ldots,N\times K-1\} \\ i & \to & K \times (i \pmod N) + \frac{i}{N} \end{array} \right.$$

Ce qui signifie que le processus P_i est placé sur le cœur $C_{\frac{i}{N}}$ du nœud $N_{i \pmod N}$ (numérotation locale au nœud, selon la formule énoncée en début de section 3.1.2.3). Dans notre exemple, le processus P_0 sera placé sur le cœur C_0 (i.e le cœur 0 du nœud 0), P_1 sera placé sur le cœur C_8

1. Mais jamais en affectant plus de processus que de cœurs disponibles toutefois.

(i.e le cœur 0 du nœud 1), P_2 sera placé sur le cœur C_{16} (i.e le cœur 0 du nœud 2) et ainsi de suite.
Enfin, dans le cas particulier où $K = 1$, c'est-à-dire le cas de machines uniprocesseurs, nous avons alors :

$$\sigma_{rr}(i) = ((i \pmod N) \pmod N) + \frac{(i \pmod N)}{N} = (i \pmod N) = \sigma_{seq}(i) \mid \forall i \in \{0, \ldots, P-1\}$$

c'est-à-dire :

$$\sigma_{rr} = \sigma_{seq}$$

Les deux politiques de placement sont donc bien identiques dans ce cas particulier.

3.1.2.4 Analyse des résultats

Nous avons utilisé pour effectuer nos comparaisons de politiques de placement une grappe de nœuds interconnectés par un réseau de type Infiniband (HCA : Mellanox Technologies MT26428 (ConnectX IB QDR)). Chaque nœud possède la structure décrite par la figure 3.1.2.1 avec deux processeurs INTEL® XEON NEHALEM X5550 quadri-cœurs cadencés à 2,66 GHz. Les premiers résultats concernent les temps d'exécution de notre exemple selon les diverses politiques de placement choisies et pour différentes tailles du jeton en circulation.

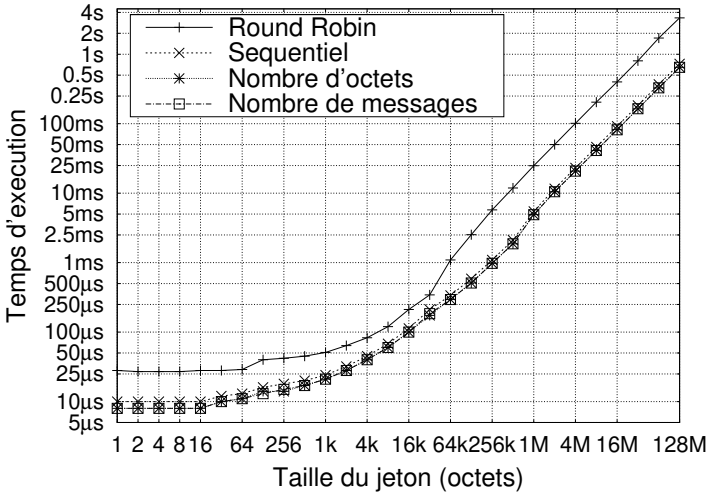


FIGURE 3.3 – Temps de circulation du jeton en fonction de sa taille et de la politique de placement.

La figure 3.3 montre les résultats pour quatre politiques de placement différentes. Outre les placements basiques séquentiel et *Round-Robin*, nous avons également appliqué deux politiques de placement essayant de mettre en correspondance le schéma de communication avec la topologie matérielle sous-jacente. Deux critères sont utilisés : le nombre de messages échangés ou le volume total de données échangées, c'est-à-dire le nombre d'octets. Nous verrons dans la section 3.3 comment faire pour mettre en place pratiquement de telles politiques de placement.

Nous constatons que le placement *Round-Robin* est celui qui donne les pires résultats et que les trois autres placements affichent des performances relativement similaires. Les deux politiques de placement utilisant effectivement la topologie virtuelle de l'application présentent toutefois un léger avantage. Le placement *Round-Robin* ne respecte pas la localité des communications : tous les échanges dans les anneaux utilisent des communications inter-nœuds

et seule l'opération collective `MPI_Allgather` en fin de programme utilise des communications intra-nœuds car dans ce cas, tous les *leaders* d'anneaux sont placés sur le même nœud. Le ratio des communications intra-nœud par rapport aux communications inter-nœuds est donc assez défavorable. Dans le cas de la politique séquentielle, la situation est inverse : les échanges intra-anneaux utilisent des communications intra-nœuds tandis que l'opération collective `MPI_Allgather` n'utilise que des communications inter-nœuds. Les deux dernières politiques, basées sur la topologie virtuelle de l'application vont vraisemblablement raffiner cette politique en s'efforçant d'utiliser le cache L3 partagé entre les quatre cœurs d'un même processeur NEHALEM. Enfin, ces deux politiques ont les mêmes performances car tous les messages sont de même taille. La figure 3.3 montre des temps d'exécution qui sont parfois pratiquement divisés par dix (en comparaison avec le placement *Round-Robin*), ce qui est considérable mais il ne faut pas perdre de vue que ce exemple n'est constitué *que* de communications, sans phases de calcul qui devraient normalement prendre une part significative du temps d'exécution dans une application parallèle réelle.

3.1.3 Une formalisation simple du problème

De façon abstraite, ce problème de mise en correspondance de topologies peut être reformulé comme un problème de minimisation de diverses métriques. La topologie matérielle est représentée par un graphe pondéré $H = (V_H, \omega_H$ où l'ensemble des sommets $V_H \in \mathbb{N}$, représente les unités physiques d'exécution tandis que les arêtes pondérées $\omega_H(u, v) \in \mathbb{R}$ avec $u, v \in V_H$ représentent le coût (ou la vitesse) des communications entre les deux sommets u and v.

La topologie virtuelle (statique) de l'application est quant à elle représentée par un autre graphe pondéré $A = (V_A, \omega_A)$, où V_A représente l'ensemble des unités virtuelles d'exécution communicant et $\omega_A(u, v)$ une métrique des communications entre deux unités d'exécution virtuelles $u, v \in V_A$. En ce qui concerne cette métrique, certains travaux utilisent le volume total des communications (en nombre d'octets ou de messages) effectuées durant une certaine phase uniquement ou bien durant la totalité de l'application. D'autres travaux proposent le nombre d'accès aux pages mémoires où sont stockées les données.

La mise en correspondance de topologies consiste à déterminer une fonction dite de **placement** $\sigma : V_A \to V_H$, qui attribue à chaque sommet $s \in V_A$ du graphe de topologie virtuelle un sommet du graphe de topologie matérielle $t \in V_H$. Certains travaux imposent que σ soit injective ou surjective, mais dans le cas général cela n'est pas nécessaire : σ peut attribuer plusieurs sommets de V_A au même sommet de V_H (cas d'*oversubscribing* de ressources) et certains sommets de V_H peuvent ne pas être attribués (cas de ressources non utilisées). Enfin, chaque fonction de placement σ possède une métrique de coût qui est typiquement la cible du problème d'optimisation (souvent une minimisation). Les métriques classiques utilisées dans la littérature sont souvent basées sur le coût ou la vitesse des communications, la congestion du réseau ou le *Hop-Byte* [213] qui est défini comme le nombre de sauts dans la topologie matérielle, pondéré par le coût des communications :

$$\text{Hop-Byte}(\sigma) = \sum_{1 \leq i < j \leq n} \omega(i, j) \times d(\sigma(i), \sigma(j))$$

avec n le nombre de processus à placer (le processus numéro i sera placé sur le cœur de numéro $\sigma(i)$), $\omega(i, j)$ la quantité de données échangées entre les processus i et j, et $d(C_1, C_2)$ la distance (mesurée en nombre de sauts) entre les unités physiques d'exécution (les cœurs) C_1 et C_2. D'autres métriques un peu plus empiriques sont parfois utilisées également pour évaluer la qualité du placement comme par exemple le ratio entre communications inter-nœuds et com-

munications intra-nœuds dans le cas de grappes de nœuds SMP ou multicœurs (cf. l'analyse des résultats de l'exemple ci-dessus).

3.1.4 Stratégies et solutions algorithmiques pour le placement

3.1.4.1 Stratégies algorithmiques

La plupart des problèmes consistant à faire correspondre une topologie quelconque A à une autre topologie quelconque H est NP-dur eu égard à l'optimisation de la métrique considérée. De fait, un nombre important de problèmes d'optimisation peuvent être formulés en tant que problèmes d'assignation quadratique. Un certain nombre d'heuristiques ont été développées par le passé et de nouvelles apparaissent régulièrement pour des topologies réseaux particulières (le plus souvent). La plupart de ces stratégies rentrent dans l'une des catégories suivantes qui sont efficaces en général pour une classe particulière de graphes :

— les solutions les plus simples sont dérivées de **stratégies gloutonnes**. Par exemple, avec une approche gloutonne locale, deux sommets de départ $u \in V_H$ et $v \in V_A$ sont choisis puis d'autres sommets sont rajoutés en parcourant les voisinages de ces deux sommets initiaux. Une approche gloutonne globale sélectionnera le prochain sommet sur la base d'une propriété globale, par exemple le poids des arêtes sortantes. Une approche mélangeant critères locaux et globaux est également possible.

— la deuxième approche fréquemment utilisée pour le placement est le **partitionnement de graphes**, par exemple le k-partitionnement et en particulier le bipartitionnement (avec $k = 2$). Le partitionnement peut être utilisé récursivement pour couper les deux graphes de topologies (A and H) en graphes plus petits qui servent à effectuer un nouveau placement au fur et à mesure que la récursion se développe. Il s'agit donc d'une approche *top-down*. Une heuristique bien connue de bipartitionnement est celle de Kernighan-Lin [7]. Träff [197] montre qu'un k-partitionnement est possible à partir de l'heuristique de bipartitionnement de Fiduccia et Mattheyses [60] (basée sur celle de Kernighan-Lin) avec une complexité temporelle équivalente.

— Une autre approche, exploitée dans LibTopoMap [86], repose sur le principe de **similarité de graphes**. Les listes d'adjacence des deux graphes de topologies sont permutés en une forme canonique (par exemple en minimisant la bande passante de la matrice d'adjacence en utilisant des heuristiques habituelles) de façon à ce que les deux graphes puissent être mis en correspondance sur la base de cette similarité.

— La dernière approche repose sur le principe d'**isomorphisme de graphe** : on suppose ici que H possède plus de sommets que A et on essaye de trouver un ensemble de sommets dans H que l'on pourrait attribuer aux sommets de A.

3.1.4.2 Quelques solutions algorithmiques

Le problème de *mapping* est souvent modélisé comme un problème de **plongement de graphe** (*Graph Embedding Problem*). Une formulation de ce problème de plongement est introduite par Hatazaki [81] tandis que Rosenberg [174] discute de sa complexité et d'un algorithme pour le résoudre.

Bokhari [21] modélise le problème comme un problème d'isomorphisme de graphe et utilise comme métrique de coût le nombre d'arêtes sortantes de la topologie virtuelle qui permettent d'atteindre des voisins dans la topologie matérielle. L'algorithme débute avec un placement initial et procède par échanges entre paires pour améliorer la métrique. Des résultats sont donnés pour des graphes jusqu'à 49 sommets.

Lee et Aggarwal [119] améliore la stratégie de Bokhari en prenant en compte l'ensemble des arêtes du graphe de topologie virtuelle. Ils proposent un algorithme glouton suivi d'une phase d'amélioration. La première étape consiste à placer la tâche qui communique le plus sur un cœur possédant un degré similaire. Les placements suivants sont guidés par une fonction d'objectif.

Berman et Snyder [12] présente une approche où les différences de cardinalité (pour le nombre de processus et de cœurs) et les variations de topologie (différences dans le graphe de topologie virtuelle) sont prises en compte. Bollinger et Midkiff [22] utilise un modèle similaire et un recuit simulé pour optimiser le *mapping* des topologies.

Baba *et al.* [8] présente un ensemble d'heuristiques de placement gloutonnes. Un processus est choisi à chaque itération sur la base d'une heuristique puis un cœur est choisi sur la base d'une autre heuristique. En particulier, une heuristique de sélection de processus pertinente est de prendre celui qui communique le plus avec les autres processus déjà sélectionnés et de le placer sur le cœur dont les coûts de communication sont les plus faibles. Ce coût de communication est modélisé avec une métrique de type *Hop-Bytes*.

Sudheer et Srinivasan [190] modélise le problème d'optimisation consistant à minimiser le *Hop-Bytes* comme un problème d'assignation quadratique. Cependant, seules des instances de taille modeste peuvent être résolues de cette façon. Une heuristique pour minimiser le nombre moyen de sauts (*hops*) est proposée. Beaucoup de solutions utilisent en pratique le partitionnement récursif et plus particulièrement le bipartitionnement récursif (BPR). Cependant, Simon et Teng [181] montre que ce bipartitionnement récursif ne permet pas toujours d'obtenir les meilleurs résultats.

Taura et Chien [192] propose de faire un placement dans un système hétérogène avec des cœurs et des liens de capacités variables. Avec cette approche les processus sont d'abord triés de façon à ce que les processus qui communiquent le plus soient placés le plus près les uns des autres. Ensuite, ces processus sont placés sur les cœurs en suivant cet ordre de tri.

Nous avons proposé avec Emmanuel JEANNOT l'algorithme TREEMATCH ([CI8], [RI3]), qui sera détaillé en section 3.3.3 et qui travaille dans un contexte de structures arborescentes hiérarchiques.

Voglestein *et al.* [204] propose une solution d'approximation du problème d'assignation quadratique, avec une complexité cubique.

Galvez *et. al* [66] propose une solution gloutonne explorant en parallèle plusieurs espaces de solutions sur la base de différents critères afin de trouver le meilleur placement.

Schultz *et. al* [178] revisite le problème d'assignation quadratique et l'optimize dans le cas de systèmes métériels hiérarchiques.

Enfin, Bordage et Jeannot [25] étudie l'impact des métriques (c'est-à-dire des fonctions de coût que l'on cherche à optimiser) sur la qualité du placement, et donc sur les gains de performances obtenus dans les applications cherchant à améliorer leur localité.

3.1.5 Mise en œuvre pratique du placement : un état (partiel) de l'art

Les mises en œuvre pratiques du placement de processus pour les architectures parallèles sont très nombreuses. En effet, depuis le milieu des années 80, certains s'intéressent à ce problème dans le contexte des machines multiprocesseurs parallèles comme Steele [188] et Chittor *et al.* [37]. Par la suite vont apparaître trois catégories de travaux : d'abord avec l'essor de logiciels dédiés au partitionnement de graphes, utilisables de façon générique, et donc plus particulièrement pour résoudre notre problème de placement. Ensuite, des solutions *ad-hoc* destinées à des topologies de réseaux particulières comme les tores, les *Mesh*, les *Dragonfly* ou encore les *Fat Trees* vont émerger. Enfin, l'arrivée des nœuds multicœurs va créer un climat

particulièrement propice pour la proposition de solutions destinées à ces machines exhibant encore plus de niveaux de hiérarchie.

3.1.5.1 Les partitionneurs de graphes

L'offre de logiciels permettant de partitionner des graphes est riche. Les cas typiques d'utilisation de ces logiciels est le partitionnement de graphes de grande taille en vue de la parallélisation de problèmes scientifiques. Les heuristiques mises en œuvre ne sont par conséquent pas toujours pertinentes pour des graphes plus réduits ou possédant une structure particulière. C'est notamment ce constat qui va nous conduire à la création et au développement de TREEMATCH (cf. section 3.3.3). Metis [106] ainsi que sa version parallèle ParMetis [177] sont des partitionneurs de graphes très utilisés. Chaco [84] et Zoltan ([54] et plus récemment [23]), développés et maintenus par le laboratoire de Sandia aux États-Unis, emploient plusieurs approches de partitionnement. Scotch [158] est un logiciel de partitionnement destiné à des graphes de grande taille qui met en œuvre un bipartitionnement récursif. Scotch utilise en entrée une structure de données arborescente appelée *tleaf* afin de procéder au *mapping* de la topologie virtuelle sur la topologie matérielle. D'autres partitionneurs sont disponibles également, mais moins connus et employés, comme Jostle [207], PATOH [32] et UMPA [53].

3.1.5.2 Placement pour topologies réseaux spécifiques

Le réseau d'interconnexion des nœuds de calcul est une partie très importante d'une machine parallèle. Ce réseau est organisé selon une topologie matérielle qui permet d'optimiser des critères variés comme le diamètre, la contention, etc. selon la taille de la machine à construire. Les hypercubes sont des topologies qui ont été très utilisées par le passé et pour lesquelles de nombreux travaux relatifs au placement de processus existent. Cependant, les hypercubes n'étant plus trop utilisés dans les architectures parallèles depuis quelques années, nous ne les aborderons pas. En revanche, d'autres topologies matérielles sont d'actualité et en particulier :

- les *Mesh* et tores multidimensionnels qui sont utilisés par exemple dans les séries BlueGene (BG/L, BG/P et BG/Q) d'IBM, dans le K Computer de Fujitsu (tore 6D) ou dans les réseaux Gemini de Cray (séries XE6 et XK7) ;
- les *dragonfly* qui sont utilisés par exemple dans les réseaux Aries de Cray (séries XC) et dans les systèmes PERCS d'IBM ;
- les *Fat Trees* qui sont souvent mis en place dans les grappes de nœuds multicœurs.

Logiquement, ces topologies sont les cibles de travaux sur le placement, dont le but est l'optimisation des communications dans les applications parallèles.

Chittor *et al.* [37] et Bhatele *et al.* [17]. travaillent sur les topologies de type *Mesh*. Bhatele *et al.* va ensuite regarder plus particulièrement l'optimisation des communications collectives pour les tores [18] puis étudier le cas des tores à cinq dimensions [19] (5D Torus). Les machines de la série BlueGene d'IBM ont été l'objet de beaucoup de travaux : Bhanot *et al.* [14], Smith et Bode [182], Yu *et al.* [213], Agarwal *et al.* [3] (qui examine en plus les problèmes de contention), Bhatele *et al.* [15] et enfin Balaji *et al.* [10]. Jingjin Wu *et al.* [208] propose un placement hiérarchique basé sur des algorithmes récursifs pour les *fat-trees* et les tores dont les performances sont intéressantes et le surcoût faible. Quant à Von Alfthan *et al.* [6], non seulement les machines BG/P d'IBM sont visées mais également les séries XT de Cray et les grappes de machines multicœurs. Pour finir, les travaux récents de Qiao *et al.* [166] étudient les effets conjoints du placement et du routage dans les topologies de type *Fat Tree*.

3.1.5.3 Placement dans les grappes de nœuds SMP

L'arrivée des nœuds SMP d'abord puis multicœurs ensuite va également permettre d'étendre et de compléter les travaux existants. Désormais, la topologie du réseau d'interconnexion n'est plus la seule qu'il faut prendre en considération : les effets NUMA et la hiérarchie mémoire interne des nœuds de calcul deviennent tout aussi importants pour les applications, autant que la congestion où les performances du réseau. Certains travaux vont même d'ailleurs se concentrer prioritairement sur la structure interne des nœuds de calcul avant de s'attaquer au reste de la hiérarchie (i.e. le réseau d'interconnexion).

Le cas des grappes de machines SMP a d'abord été traité, ainsi que nous l'avons vu, par les implémentations de MPI en offrant aux applications utilisant le passage de messages des communications performantes intra-nœuds, que ce soit au niveau des point-à-points que des collectives (cf. section 2.4.4). Le placement a surtout fait l'objet au début de quelques travaux concernant les topologies virtuelles de MPI (cf. section 2.4.5). Puis, avec la conjonction de l'augmentation du nombre de cœurs et de l'apparition des effets NUMA, le placement s'est généralisé comme une technique pertinente pour améliorer les performances des applications parallèles. En effet, le placement permet d'influencer le déploiement et l'exécution de l'application sans avoir *nécessairement* à modifier cette dernière.

Dès le milieu des années 2000, des travaux seront ainsi consacrés au grappes de machines SMP, comme ceux de Wallace [206] ou d'Orduña *et al.* [154], qui s'attache surtout à gérer les effets des routeurs dans le réseau d'interconnexion et laisse de côté la structure interne des nœuds de calcul. La topologie est modélisée statiquement, sous la forme d'une matrice de distance, sans faire appel à un outil particulier pour récupérer ces informations. *MPI Process Placement toolset* (MPIPP) [36] utilise un $k-$partitionnement pour calculer un placement destiné à améliorer les communications dans les grappes de grappes. La hiérarchie visée ici est donc clairement celle du niveau inter-nœuds mais le niveau intra-nœud pourrait être exploité si les informations pertinentes étaient fournies à MPIPP qui utilise une approche quantitative dans sa modélisation du matériel sous-jacent. Chavarría-Miranda *et al.* [35] quant à lui, essaye de mettre en correspondance une topologie virtuelle cartésienne (à deux ou trois dimensions) avec une topologie matérielle simplifiée à deux niveaux [2].

3.1.5.4 Placement dans les grappes de nœuds multicœurs

Des travaux préliminaires décisifs En 2009, deux articles publiés à quelques semaines d'intervalle décrivent deux approches s'intéressant (enfin) à la structure interne des nœuds NUMA multicœurs : celui de Rodrigues *et al.* [173] et le mien [CI10]. Non seulement les cibles de ces travaux sont identiques (des applications purement MPI s'exécutant sur des grappes de nœuds multicœurs) , mais les approches sont également très similaires : la topologie virtuelle est obtenue via un profilage de l'application MPI (avec des métriques de type nombre de messages ou bien volume total des communications en octets) et le placement des processus MPI est calculé par un bipartitionnement récursif grâce à Scotch. La façon de modéliser l'architecture matérielle diffère quelque peu : là où Rodrigues *et al.* utilise une approche quantitative en mesurant des latences et des débits (au niveau intra-nœud et inter-nœuds), je fournis à Scotch une représentation un peu plus abstraite avec des valeurs numériques choisies pour exprimer plus une différence de performance entre niveaux que des valeurs réelles (approche un peu plus qualitative donc). Enfin la différence fondamentale est que je me suis concentré sur la topologie matérielle intra-nœud tandis que Rodrigues *et al.* considère également le réseau (le niveau inter-nœuds, donc). Dans les années qui suivent, je vais poursuivre ces travaux en

2. Le même type de simplification rencontré dans l'optimisation des opérations collectives des implémentations de MPI.

cherchant à améliorer les différentes étapes (récolte des diverses informations, calcul et application concrète du placement obtenu) en restant dans le contexte des applications MPI pures comme nous le verrons en section 3.3. Rodrigues *et al.* explorera plutôt les applications hybrides mélangeant passage de messages et multithreading (MPI + OpenMP par exemple, cf. ci-dessous).

Influence sur la thématique du placement intra-nœud Dans le sillage de ces travaux, de nombreuses autres équipes vont s'intéresser au placement de processus dans les grappes de nœuds hiérarchiques complexes : Zhang *et al.* [218] propose une décomposition des opérations collectives de MPI en opérations point-à-point puis effectue un placement des processus afin de réduire les communications.

Ito *et al.* [99] applique des politiques de placements dans le contexte des applications basées sur la décomposition de domaines, ce qui permet d'exprimer une forme de hiérarchie. Aktulga *et al.* [4] fait de même mais pour des applications de calcul de valeurs propres.

Subramoni *et al.* [189] travaille plus particulièrement sur les grappes dont le réseau d'interconnexion est basé sur la technologie Infiniband. Ces travaux décrivent un système de découverte de la topologie matérielle du réseau Infiniband dans la machine : c'est donc la partie de récupération des informations de topologie matérielle qui est ici ciblée. Ce travail utilise pour se faire la technique du *Neighbor-Joining*.

Li *et al.* [121] propose de considérer la topologie matérielle du réseau en plus de la topologie intra-nœud, mais adopte une approche un peu plus qualitative que Rodrigues *et al.* [173]. Ces travaux utilisent hwloc pour la topologie matérielle intra-nœud et une solution *ad-hoc* pour ce qui concerne la topologie réseau.

Abdel-Gawad *et al.* présente RAHTM [2], qui utilise trois phases pour déterminer le placement avec d'abord une phase pour le placement dans les nœuds multicœurs puis deux autres phases pour le réseau. La congestion ainsi que des informations de routage sont utilisées à cette fin. RAHTM se destine plutôt pour des topologies de type tore/mesh.

PTRAM [142] vise les réseaux de type *fat-tree* dans les grappes de nœuds interconnectés par Infiniband. Il s'agit d'une approche gloutonne qui utilise Scotch pour le partitionnement avec une étape de raffinage. Les métriques utilisée pour l'optimisation du coût ne sont pas le *Hop-Bytes* où la congestion du réseau mais des métriques dites hybrides mélangeant ces métriques plus traditionnelles.

Cas particuliers de travaux sur le placement Certains travaux prennent en considération des critères parfois différents de ceux traditionnellement utilisés. Par exemple, Venkatesan *et al.* [203] ou Latham *et al.* [117] cherchent à établir un placement permettant d'améliorer les performances des opérations d'entrées/sortie sur le système de fichier (MPI/IO).

Si la topologie matérielle du réseau d'interconnexion est parfois prise en compte (pas toujours), certaines autres caractéristiques, comme la contention, peuvent être aussi considérées dans le placement afin de la réduire au minimum. C'est notamment le cas de Soryani *et al.* ([185] ou de [186]) Deveci *et al.* [52] qui proposent une approche en deux phases : un partitionnement de la topologie virtuelle d'abord, suivi de la mise en correspondance avec la topologie matérielle. Trois algorithmes de placement sont détaillés : une première version gloutonne utilisant Metis pour le partitionnement et deux autres versions améliorant l'initiale : amélioration du nombre de sauts pondérés (*weighted hops*) et de la congestion, respectivement.

Deveci *et al.* [51] propose d'utiliser un partitionnement géométrique (le *Multi-dimensional Jagged*) pour le placement dans le cas d'une allocation des ressources de calcul non contigües. L'algorithme proposé cible les machines Cray XE6 avec un réseau Gemini et utilise les coordonnées des routeurs pour représenter la topologie matérielle du réseau d'interconnexion de

la machine. L'outil de partitionnement utilisé est Zoltan2.

Enfin, des travaux sont consacrés au effets NUIOA (*Non Uniform I/O Access*) c'est-à-dire aux conséquences d'un placement de processus effectuant un nombre important de communication inter-nœuds loin de la carte réseau. C'est le cas en particulier de Brice GOGLIN [72] (membre de TADaaM) et de Zarrinchian *et al.* [214] qui essaye de minimiser les accès concurrents sur la carte réseau.

3.1.5.5 Placement d'applications hybrides MPI + X

Le passage de messages n'est évidemment pas le seul paradigme de programmation parallèle. Il est en effet possible de paralléliser une application sans échanger des données de façon explicite. C'est notamment le cas des applications basées sur les processus légers (*threads*) qui se partagent l'espace d'adressage du processus classique duquel ils dépendent. Dans le cas de machines à mémoire partagée, cette approche est intéressante car elle élimine le surcoût lié à l'utilisation d'une bibliothèque de passage de messages. Cependant, dans le cas de machines à mémoire physiquement distribuée – comme dans les grappes – l'utilisation des threads peut se révéler problématique. Une approche dite hybride, mélangeant le passage de messages et les threads peut apporter des réponses intéressantes.

La question de l'utilisation d'un tel paradigme s'est posée dès l'arrivée des grappes de nœuds SMP [183], mais avec des résultats parfois mitigés [31]. Le problème est redevenu d'actualité avec l'apparition des grappes de nœuds multicœurs. En effet, le nombre de cœurs par nœud augmentant de façon sensible, la question de l'adéquation de MPI dans un tel environnement d'exécution se pose. Dans une telle configuration, il faut non seulement effectuer un placement des processus MPI mais également un placement des threads qui sont créés à l'intérieur (création avec des directives OpenMP par exemple). Le graphe de la topologie virtuelle (ie le comportement applicatif) ne peut désormais plus être déterminé uniquement à l'aide des volumes de données échangés ou du nombre de messages car les communications sont implicites dans une section parallèle avec des threads. Les accès mémoire peuvent néanmoins être instrumentés de façon à obtenir une information pertinente et utilisable concrètement.

Dümmler *et al.* [57] est l'un des premiers à regarder cette thématique dans le contexte des nœuds multicœurs. L'architecture matérielle est modélisée avec la notation de Dewey et des indices logiques de cœurs/threads matériels sont utilisés. Le placement se base en fait sur la structure des sections parallèles et met en place des politiques simples de type *scatter* (un peu comme le *Round Robin* vu en section 3.1.2.3) où les threads d'une section sont distribués sur les nœuds de calcul pour éviter de la contention sur un bus mémoire en particulier, ou bien de type séquentielle.

Rabenseifner *et al.* [169] discute plus généralement de la problèmatique de la programmation hybride. Il propose une taxonomie des différents types de programmation hybride possibles et montre les problèmes qui peuvent se poser lorsque le passage de messages est utilisé conjointement avec des threads. L'influence du placement est évoquée et l'approche retenue pour la modélisation de l'architecture est qualitative (i.e. des différences de performances entre niveaux hiérarchiques sont utilisées plutôt que des performances mesurées réellement).

Rodrigues *et al.* [173] évoque le placement hybride et le liste dans les travaux à poursuivre. En particulier, Diener *et al.* [55] utilise les accès des theads aux pages mémoires afin d'établir la topologie virtuelle, ce qui suppose dans certains cas des modifications au niveau de la gestion de la mémoire virtuelle dans le système d'exploitation, voire au niveau du matériel lui-même, ce qui est délicat. Cruz *et al.* [80] établit un état de l'art consacré au modèle MPI + threads, auquel le lecteur pourra se reporter.

Enfin, il existe des approches hybrides un peu plus originales, telle que celle décrite dans

Hoefler *et al.* [89], qui utilise MPI + MPI. c'est-à-dire le passage de messages et l'interface de communication par mémoire partagée, intégrée dans le standard depuis MPI 3.0 (cf. section 4.1.2).

3.1.6 Impact de mes premiers travaux consacrés au placement

Un partie importante des travaux listés dans les paragraphe précédents et postérieurs à 2009 citent mon article [CI10]. C'est aussi le cas de [88] qui servira de base pour la standardisation des topologies virtuelles distribuées dans MPI 2.2. Ce travail a donc eu un impact important dans le domaine et de plus il a permis de mettre en évidence plusieurs éléments, totalement évidents aujourd'hui mais qui étaient loin d'aller de soi à l'époque :

1. le placement dans les nœuds hiérarchiques multicœurs est **important pour les performances** et peut avoir un impact négatif si la politique choisie n'est pas pertinente ;
2. il est **indispensable** de lier un cœur de calcul à un processus MPI sinon la politique mise en place n'a aucun sens. En effet, sans cette opération de lien (*process pinning* ou *process binding*), un processus est susceptible de changer de cœur en cours d'exécution suite à un changement de contexte par le système d'exploitation. Outre la perte éventuelle du contenu des caches mémoire et de l'invalidation du TLB et de la table des pages de la MMU du cœur utilisé, un tel changement peut induire des effets néfastes sur l'accès aux données et sur les échanges intra-nœuds. Là aussi, il s'agit d'un problème qui a été remis au gôut du jour avec les machines hiérarchiques multicœurs [3] car dans le cas uniprocesseur, il n'y avait évidemment pas de *binding* à faire. La politique de placement consistait «juste» à faire une répartition sur les différents nœuds de la machine, via une option du gestionnaire de processus de l'implémentation de MPI, le *machinefile* ;
3. il serait souhaitable, voire **nécessaire** de mettre au point un outil portable permettant de récupérer des informations sur la topologie matérielle sous-jacente (au moins au niveau intra-nœud). C'est ce qui motivera la création de hwloc [CI7] (cf. section 3.3.2). En particulier, l'utilisation de la numérotation **physique** des cœurs de calcul devrait être abandonnée, au profit d'une numérotation **logique** pour assurer une portabilité et une pérénité des politiques de placement.

Ainsi que nous le verrons en section 3.2.2.3, je vais poursuivre ces travaux initiaux dans de multiples directions ainsi que d'autres membres des EPC Runtime/TADaaM. En particulier, cela conduira au développement de hwloc et de TREEMATCH (cf. section 3.3.3), un algorithme de placement dédié aux architectures hiérarchiques. TREEMATCH possède un certain nombre de compétiteurs dont nous ferons une liste non exhaustive dans la section qui lui sera consacrée.

3.2 Le placement de processus dans MPI (et ses applications)

Dans cette section, nous allons examiner comment la problématique du placement de processus est traitée dans le standard MPI, ses implémentations et dans les applications l'utilisant. Dans un premier temps, nous allons revenir sur les notions de processus MPI (*MPI Process*) et de numéro de rang (*MPI Rank*) afin de dissiper les amalgames. Nous verrons ensuite comment la politique de placement permet de changer les propriétés d'un processus MPI puis ce que dit (ou non) le standard MPI et ce que font effectivement les implémentations à ce sujet avant de discuter des problèmes que cela entraîne.

3. Problème qui existait déjà dans le cas de machines multiprocesseurs en fait.

3.2.1 Placement ou renumérotation de processus ?

En fait, le problème du placement de processus peut être abordé sous deux angles différents. Les résultats sont les mêmes mais la mise en place pratique diffère selon les cas et apporte son lot d'avantages et d'inconvénients. Avant de faire cette comparaison d'approches et de méthodes, il nous faut préciser des concepts et objets très souvent utilisés dans MPI mais dont les utilisateurs ne saisissent pas toujours le sens précis et qui sont sources de confusion.

3.2.1.1 MPI Process vs. MPI Rank

Il est courant pour les utilisateurs de MPI d'employer indistinctement les termes *MPI Process* et *MPI Rank* pour désigner un processus applicatif. Or il s'agit de deux concepts bien distincts dont la rigueur interdit la confusion. Déjà, et de façon étonnante, **la notion de *MPI Process* n'est toujours pas définie à ce jour dans MPI** ! Les *MPI Process* ont été assimilés à des processus lourds UNIX au moment de la création de MPI mais le terme n'a jamais été clairement défini. Il s'agit d'un point particulièrement délicat car un certain nombre d'implémentations utilise des threads en tant que support pour les *MPI Process* en lieu et place de processus lourds traditionnels (cf. section 2.1.3). De plus, imposer une telle définition reviendrait à imposer un certain type d'implémentation, ce qui n'est pas le rôle du standard. Au final, la situation est paradoxale car les utilisateurs se retrouvent à manipuler un concept pour lesquels il n'existe pas d'objet correspondant dans MPI.

Jusqu'à présent, cela n'a pas constitué un problème majeur car les utilisateurs pouvaient manipuler ces *MPI Process* via une de leur propriété qui est leur **numéro de rang**. Chaque processus dans MPI appartient à un (ou des) **communicateur(s)** au sein duquel il possède un unique numéro de rang qui lui sert d'identificateur. Dans un communicateur, les numéros de rang sont linéaires et contigus, de 0 à $n-1$ si n processus distincts sont membres de ce communicateur. Un communicateur est en quelque sorte un «groupe» de *MPI Process* auquel on ajoute une information de contexte, qui permet de discriminer plus précisément les communications qui s'y déroulent [4]. Ainsi, toutes les opérations de communication dans MPI sont **contextuelles** et se déroulent dans le cadre d'un communicateur. Un *MPI Process* peut appartenir à autant de communicateurs qu'il le souhaite, mais ne peut pas posséder plusieurs rangs dans un communicateur.

Au lancement d'une application MPI, tout processus appartient en fait à deux communicateurs :

— `MPI_COMM_SELF` : un communicateur qui ne contient qu'un seul processus, de numéro de rang 0 (forcément) ;
— `MPI_COMM_WORLD` : un communicateur contenant tous les processus de l'application (à son lancement). Donc si l'application est composée de P processus, ils seront numérotés dans `MPI_COMM_WORLD` de 0 à $P-1$.

Il est courant de lire ou d'entendre parler du «*processus de rang i*», mais sans l'information de contexte (i.e le communicateur associé) cela n'a aucun sens dans l'absolu. Dans un tel cas, la référence au communicateur `MPI_COMM_WORLD` est implicite car il s'agit du communicateur comprenant tous les processus de l'application, créé à son lancement par l'implémentation MPI et avec lequel les utilisateurs sont habitués à travailler. Il s'agit toutefois d'un abus de langage [5], assez fréquent.

4. Il existe également la notion de groupe dans MPI, mais qui est moins utilisée en pratique que les communicateurs.

5. Abus que j'ai moi-même commis dans ce document pour plus de simplicité, cf. la figure 3.2 qui présente des rangs d'émetteur et récepteur dans `MPI_COMM_WORLD`, bien entendu ...

3.2.1.2 Une question d'interprétation

Dans la section 3.1.1.2, nous avons expliqué les liens entre localité et placement. En fait, il est possible d'améliorer la localité d'une application par deux moyens distincts, en jouant sur l'une des deux propriétés suivantes d'un *MPI Process* :

— en imposant son emplacement physique, c'est-à-dire le numéro de cœur sur lequel il va s'exécuter. Dans ce cas, il s'agit de faire du **placement de processus** proprement dit (ce que nous avons vu jusqu'à présent) ;

— en créant un nouveau contexte de communication (i.e. un nouveau communicateur) dans lequel le rang des différents *MPI Process* sera modifié, de façon à refléter la topologie virtuelle de l'application. Dans ce cas, il s'agit de faire une **renumérotation de rangs (*rank reordering*)**. Si dans le premier cas, il est évident que les processus doivent être liés (ou *bindés* selon le barbarisme consacré) à des cœurs, c'est aussi le cas pour la renumérotation de rangs. En effet, cette renumérotation ne peut être efficace que si des garanties existent quant à la localisation des processus.

La modification de l'une ou l'autre de ces deux propriétés du *MPI Process* repose sur une différence dans l'interprétation du résultat donné par la fonction de placement σ, comme montré sur la figure 3.4. Si le résultat $\sigma(i)$ pour $i \in [0, \ldots, P-1]$ est interprété comme un numéro de

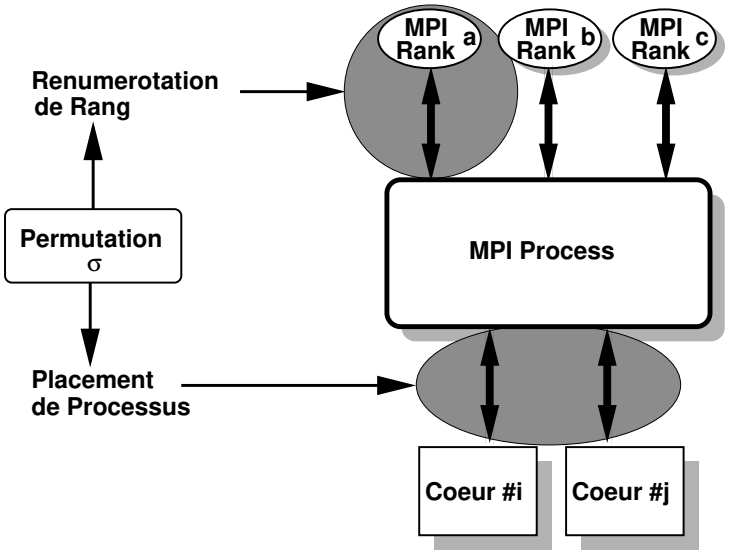


FIGURE 3.4 – Placement et renumérotation de processus

cœur, alors il s'agit de faire du placement de processus. C'est la façon la plus courante d'interpréter ce résultat. En revanche, $\sigma(i)$ peut également être interprété comme un nouveau numéro de rang et dans ce cas, il s'agit de faire une renumérotation du rang du processus, auquel cas le placement d'origine de l'application n'est pas modifié [6].

3.2.1.3 Comparaison des deux approches

Avantages et inconvénients du placement de processus Le placement de processus est une chose connexe à MPI mais n'en faisant pas partie *stricto sensu*. En effet, il s'agit d'une problématique relevant du gestionnaire de processus de l'implémentation de MPI et pas de cette

6. En réalité, cette seconde interprétation n'est vraie que si les ensembles servant à numéroter les processus et les cœurs contiennent les mêmes éléments. Dans le cas contraire, il y aura une étape supplémentaire consistant à déduire le rang du processus i à partir de $\sigma(i)$, mais cette étape est triviale, d'où notre simplification.

dernière directement car le placement n'est, à l'heure actuelle, pas pris en charge ou même considéré dans le standard MPI (cf. ci-dessous section 3.2.2). Les conséquences sont que la mise en application d'une politique de placement doit se faire exclusivement au moment du lancement de l'application et qu'elle ne peut pas *a priori* être modifiée par la suite (de façon standard avec MPI tout du moins). De plus, la commande de lancement d'applications dans MPI, `mpirun` ou `mpiexec` ne possède pas d'arguments standardisés pour le placement, ce qui signifie que ces arguments ou/et leur sémantique peut changer d'une implémentation de MPI à l'autre, voire même d'une version à l'autre d'une même implémentation. Cela pose des problèmes sérieux de portabilité (cf. section 3.2.3). En revanche, – et c'est là un avantage considérable – il est possible d'appliquer une politique de placement **sans avoir à modifier ou recompiler une application MPI existante**.

Avantages et inconvénients de la renumérotation de rangs C'est à l'application elle-même de procéder au calcul de la renumérotation de rangs, par le biais d'une fonction dédiée, telle que `MPI_Graph_map` ou grâce à la création d'un nouveau communicateur au sein duquel les communications seront optimisées selon les topologies virtuelles et matérielles. Il existe dans MPI plusieurs fonctions permettant cette opération, notamment `MPI_Graph_create` et `MPI_Dist_graph_create`, via un paramètre `reorder` qui autorise une renumérotation des processus du groupe. Cela permet donc à l'application de **pouvoir changer sa politique d'affinité dynamiquement**, ce qui peut être intéressant pour des applications avec des phases différentes de calcul comme les maillages adaptatifs par exemple. Les conséquences sont que 1) l'application doit être modifiée pour pouvoir utiliser la renumérotation et 2) le calcul de cette renumérotation incombe à l'application, ce qui n'est pas le cas pour un placement déterminé avant le déploiement et l'exécution des processus applicatifs. Cependant, le problème majeur de cette approche réside dans son aspect non-standard. En effet, les implémentations ne sont pas forcées de mettre en œuvre cette renumérotation et quand bien même, elle sera basée sur leurs propres critères. Ainsi, une application faisant un appel à la version d'une bibliothèque MPI A de la fonction `MPI_Dist_graph_create` pourra voir cette dernière prendre en compte effectivement le schéma de communication et la topologie matérielle sous-jacente, tandis que la version d'une bibliothèque MPI B pourra utiliser tout autre chose. Il n'y a donc pas de garanties, du point de vue fonctionnel, que la topologie matérielle est bien prise en compte.

Présences respectives dans la littérature Il y a donc plusieurs possibilités d'améliorer la localité dans les applications utilisant le paradigme du passage de messages. La technique de renumérotation est toutefois bien moins utilisée que celle du placement, comme en témoigne l'état de l'art. Outre mes travaux dédiés spécifiquement à la fonction `MPI_Dist_graph_create` [CI6], Brandfass *et al.* [26] propose une renumérotation des processus en modifiant le contenu du *machinefile* de l'application. Cependant, la technique proposée qui vise uniquement à augmenter le ratio de communications intra-nœuds dans l'application, n'est pas une renumérotation *stricto sensu* mais plutôt une forme de placement de *MPI Process* en jouant sur la façon dont l'implémentation interprète le contenu de ce fichier de lancement de l'application. Certaines implémentations commerciales, comme celle de Cray, utilisent ce mécanisme de renumérotation [83]. Il existe quelques autres travaux plus récents, portant sur l'amélioration des communications collectives, comme Mirsadeghi *et al.* [143] ou Llorente *et al.* [126]. Enfin, et comme nous l'avons un peu évoqué dans les paragraphes précédents, ces deux possibilités pâtissent de la frilosité du standard MPI (mais moins de ses implémentations) quant à cette problématique. Nous allons donc examiner ce que MPI dit à ce sujet dans la section suivante.

3.2.2 MPI et la prise en compte de la localité

3.2.2.1 Une situation ambigüe

Du point de vue du standard MPI, le placement est considéré comme une problématique externe, ainsi que l'indique cet extrait de la première page du chapitre consacré aux topologies virtuelles dans la version actuelle du standard (MPI 3.1, chap. 7) :

> *"A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent. The functions that are described in this chapter deal with machine-independent mapping and communication on virtual process topologies."*

On notera au passage l' ambiguïté du terme «*done*» qui peut à la fois être interprété comme «*calculé*» ou bien comme «*appliqué*», ce qui n'implique pas vraiment la même chose. La renumérotation n'est pas mieux lotie, avec cet autre extrait du même chapitre relatif à la fonction `MPI_Dist_graph_create` :

> *"If* `reorder` *= true" then the MPI library is free to remap to other processes (of* `comm_old`*) in order to improve communication on the edges of the communication graph. The weight associated with each edge is a hint to the MPI library about the amount or intensity of communication on that edge, and may be used to compute a "best" reordering."*

Il n'y a donc pas d'obligation d'implémenter la renumérotation des processus et les critères demeurent à la discrétion de l'implémentation.

3.2.2.2 Évolutions dans le standard MPI

La tendance vers des machines *exascale* et l'apparition des machines hiérarchiques complexes a néanmoins fait bouger les lignes avec une remise en question du modèle de programmation «plat» de MPI afin d'y introduire de la hiérarchie et d'y prendre en considération la localité. Une illustration concrète de ce mouvement est par exemple l'introduction dans MPI 3.1 de la notion de voisinage dans les opérations collectives (*neighborhood collectives*) et des fenêtres de mémoire partagée. Avec ces nouvelles fonctionnalités, une application a désormais l'opportunité d'optimiser ses échanges intra-nœuds en s'affranchissant du surcoût de la bibliothèque MPI mais aussi est en capacité de mettre en place un schéma de communications collectives plus local.

Cependant, la mise à l'écart du placement des processus perdure. Ce problème ne se posait pas au début de MPI, c'est-à-dire du temps des implémentations de première génération et des nœuds de calcul uniprocesseurs. Mais comme nous l'avons montré, ce point est devenu crucial pour la nouvelle génération d'implémentations qui visent une exploitation optimale des machines multicœurs hiéararchiques complexes. Placer des processus pour mitiger les effets NUMA est essentiel, mais cette problématique du placement déborde du cadre strict du passage de messages et peut s'appliquer à d'autres paradigme de programmation, y compris hybrides (cf. section 3.1.5.5).

La généralisation de la problématique devrait inciter les standards à se pencher sérieusement sur la question, pour ne pas laisser les utilisateurs avec des solutions *ad-hoc* élaborées par les administrateurs de systèmes particuliers et qui ne sont souvent qu'une duplication d'efforts existants, à l'instar de PlaceMe [85], de [40], de [114] ou encore de [150]. Il existe toutefois des

solutions plus génériques : Libquo [79], disponible sous la forme d'une bibliothèque externe basée sur hwloc et MPI, permet de faire du placement d'applications hybrides couplées. Cette solution possède des objectifs similaires à celle que nous avons développée avec le CERFACS (Hippo) et qui est décrite en section 4.2.6. La mise au point d'une solution générique et portable pour le placement d'applications (MPI pures ou hybrides) est d'autant plus pertinente que d'après l'étude de l'*Exacale Computing Project* [13], 90% des développeurs indiquent avoir besoin de pouvoir placer finement les processus MPI et threads de leurs applications (61% via `mpiexec` et 29% via le gestionnaire de ressources et de tâches).

3.2.2.3 Support pratique du placement dans l'écosystème MPI

Phases dans la mise en application du placement Outre les solutions particulières principalement destinées à l'application de politiques simples de placement dans un cadre hybride de type MPI + OpenMP, il existe également un éventail d'environnements qui implémentent les différentes étapes nécessaires pour déterminer puis appliquer une politique de placement plus sophistiquée, c'est-à-dire basée sur le schéma de communication applicatif. Ces étapes sont les suivantes :

Étape 1 : la récupération du schéma de communication applicatif, c'est-à-dire la **topologie virtuelle**

Étape 2 : la récupération des informations de **topologie matérielle**

Étape 3 : la mise en correspondance de ces deux informations, c'est-à-dire le calcul du **placement** ou de la **renumérotation** des processus

Étape 4 : l'**application effective** de ce placement ou de cette renumérotation

Nous détaillerons dans la section 3.3.1 les différentes approches possibles concernant la première étape. En ce qui concerne la deuxième étape, il y a d'une part l'approche basée sur une connaissance *a priori* du matériel et qui est destinée à des architectures spécifiques. Cette approche est souvent suivie par les constructeurs. Une approche générique est également possible, par le truchement d'un élément logiciel permettant de récupérer ces informations (quantitatives et/ou qualitatives), avec plus ou moins de précision. La troisième étape utilise des solutions pour le placement comme celles déjà détaillées en section 3.1.5, en particulier les partitionneurs de graphes (cf. section 3.1.5.1). Quant à la quatrième et dernière étape, elle repose en général sur des commandes spécifiques du système d'exploitation permettant de lier un processus à une unité de calcul, comme `numactl` ou `taskset`. Il convient d'ores et déjà de noter qu'il n'existe pas de commande standard permettant une telle assignation.

Au niveau de l'existant, nous avons déjà evoqué MPIPP [36] destiné au placement dans les grappes de grappes de machines SMP. Il existe aussi TopoMapping [66] qui utilise son propre système de traces (pour la première étape) et TopoMgr [16], destiné aux machines Cray et BlueGene pour récupérer les informations de topologie matérielle (pour la deuxième étape). Les constructeurs/vendeurs proposent souvent des environnements intégrés permettant l'exploitation de la localité dans les applications qui utilisent leur version propriétaire de MPI. C'est notamment le cas de Cray, avec son *Cray Performance Measurement and Analysis Tool* (CrayPAT) [45], de HP [48] et aussi probablement d'IBM [59].

Extension des gestionnaires de processus et/ou de la commande `mpiexec` La mise en place d'une politique de placement (i.e. la dernière étape) peut se faire au moment du lancement de l'application[7]. Ce lancement se fait en règle générale avec une commande `mpirun` ou `mpiexec` dont seuls quelques arguments sont standardisés et dont la plupart dépendent de

7. La mise en place de la rénumérotation se fait dans l'application MPI elle-même, comme expliqué en section 3.2.1.3.

l'implémentation de MPI qui est libre de changer leur nom ou leur sémantique à sa guise, ce qui pose de sérieux problèmes de portabilité pour le déploiement des applications. Il faut de plus noter que l'utilisation de cette commande `mpiexec` n'est pas obligatoire et que d'autres mécanismes sont licites pour le lancement et le déploiement d'applications MPI. En effet, les constructeurs de machines HPC proposent un environnement intégré verticalement, couvrant le matériel et un ensemble de couches logicielles, incluant en particulier la réservation et la gestion des ressources de calcul (*Resource and Job Management System*). Cet élément logiciel peut être amené à déployer une application MPI par ses propres moyens au lieu d'utiliser les commandes habituelles. Par exemple, une application qui réserve des ressources avec le RJMS Slurm sera déployée avec la commande `srun` et non avec `mpiexec`. C'est là l'une des raisons qui fait que la standardisation de `mpirun/mpiexec` n'est actuellement pas à l'ordre du jour.

Néanmoins pour un nombre substantiel de solutions, la mise en application de la politique de placement passe par une extension de cette commande. Par exemple, le gestionnaire de processus de MPICH, Hydra, (sur lequel repose la version «MPICH» de `mpiexec`) accepte une option `-use-app-topology` afin de prendre en compte la topologie virtuelle applicative [96] [8].

Du côté d'Open MPI, on trouve LAMA (*Locality-Aware Mapping Algorithm*), issu des travaux de Hursey *et al.*([93] et [92]) qui est un système s'inspirant des travaux effectués sur les machines de type BlueGene pour un meilleur contrôle de l'affinité des processus MPI au moment du lancement des applications. LAMA offre un large éventail de politiques de placement aux utilisateurs avec une façon relativement synthétique pour ces derniers d'exprimer leurs souhaits. Ces politiques sont dérivées de politiques simples (i.e. *Round-Robin* et séquentielle) mais ne prennent pas en compte le schéma de communication d'une application hybride MPI + OpenMP. Pour la récupération des informations de topologie matérielle, LAMA utilise le logiciel hwloc.

mpipin [69] est une solution basée sur une sémantique générique d'arguments pour le placement. mpipin fonctionne comme un programme intermédiaire s'intercalant entre la commande de déploiement et l'application proprement dite afin de faire le placement qui sera décrit avec la sémantique de mpipin et non pas celle, spécifique, du lanceur utilisé.Une autre solution, proposée par Vallée *et al.* [201] utilise l'infrastructure de PMIx (*Process Manager Interface*) pour centraliser et appliquer les décisions relatives au placement dans le cas d'applications hybrides MPI + OpenMP. Ainsi, c'est PMIx qui est mis à contribution en lieu et place des supports exécutifs des implémentations de MPI et d'OpenMP utilisées. Cette solution, ainsi que mpipin apportent un service comparable à celui fourni par les solutions *ad-hoc* destinées à un système particulier, mais ont le mérite appréciable d'un plus grande généricité.

Enfin, Edgar León propose MPI_Bind [120], une politique de placement sophistiquée prenant en compte la hiérarchie de la mémoire des nœuds de calcul et la présence d'accélérateurs (comme les GPU par exemple). MPI_Bind utilise hwloc pour la récupération des informations de topologie matérielle intra-nœud.

3.2.3 Un problème non trivial

La section précédente nous permet d'entrevoir pourquoi la mise en place d'une politique spécifique de placement est un problème moins simple qu'il n'y paraît. En effet, cela ne concerne pas uniquement MPI et met en jeu d'autres éléments de l'écosystème HPC, comme par exemple le gestionnaire des ressources de calcul ou encore le support exécutif d'un autre modèle de programmation utilisé conjointement avec le passage de messages. L'absence d'un meilleur support dans MPI peut s'expliquer pour ces raisons.

8. Je ne sais pas si cette option existe toujours au moment de la rédaction de ce document.

3.2.3.1 La question *meta* : où placer le placement ?

Ainsi que déjà expliqué précédemment, une meilleure gestion de la localité peut être obtenue en affectant les processus à des ressources de calcul (opération de *binding*) puis en appliquant une politique de placement ou bien en renumérotant les rangs des processus MPI de l'application. Or, ces deux opérations ne se sont pas effectuées au même niveau puisque le placement est décidé au moment du déploiement tandis que la renumérotation est calculée pendant l'exécution de l'application. De façon plus générale, la mise en œuvre du placement est susceptible d'intervenir à différents niveaux dans l'écosystème HPC :

— par le gestionnaire de ressources (RJMS) après la réservation et l'allocation de ces dernières : il peut prendre la décision de procéder au *binding* des processus ;
— par le gestionnaire de processus de l'implémentation MPI au moment du déploiement de l'application : il peut à la fois placer les processus puis faire le *binding* dans la foulée ;
— par l'implémentation de MPI au moment de son initialisation dans l'application : elle peut là encore procéder au placement et surtout faire le *binding* des processus ;
— par l'application MPI elle-même durant son exécution : elle peut décider de *binder* les processus afin de pouvoir faire une renumérotation si ce n'est pas déjà fait.

Il y a donc un besoin de coordonner cette prise de décision et surtout d'arriver à faire en sorte qu'elle ne soit pas prise plusieurs fois, ce qui induit un risque d'obtenir des résultats contradictoires. De plus, l'absence d'options et de sémantique associée transposables d'une implémentation à l'autre renforce le problème : si les implémentations de MPI ne sont pas capables de se mettre d'accord, on voit mal comment elles pourraient s'entendre avec le gestionnaire de ressources à ce sujet. Une approche serait de mettre en place une entité responsable des décisions d'affectation et de placement échangeant des informations avec les différentes couches logicielles capables d'intervenir à ce niveau pour qu'elles sachent si elles seraient autorisées à prendre des décisions et dans quel périmètre.

3.2.3.2 Au-delà de MPI : le cas des applications composées ou hybrides

Non seulement la localisation de la prise de décision et de la mise en œuvre pose problème mais la structure de l'application peut également rendre les choses difficiles. Une application hybride utilisant conjointement MPI et OpenMP doit prendre des décisions pour le placement des processus MPI, décisions qui se doivent d'être compatibles avec la création dynamique de sections parallèles OpenMP afin de pouvoir exploiter au maximum les différentes ressources de calcul disponibles. Par exemple, s'il est intéressant d'affecter un processus à un cœur particulier, cela va poser un problème en cas de création de threads car ces derniers seront tous affectés à ce même cœur, ce qui nuira aux performances. Les interactions entre les différents environnements de programmation doivent être pensées avec soin pour éviter les conflits et incompatibilités.

Le cas des applications composées est aussi important : une application peut très bien faire appel à une bibliothèque utilisant MPI en interne sans que cette dernière n'en soit informée. Dans ce cas, elle peut être amenée à prendre des décisions rentrant en conflit avec ce que l'implémentation MPI souhaiterait faire ou a déjà fait. Un autre cas est celui des applications composées de plusieurs noyaux de calcul possédant chacun une politique de placement spécifique afin d'obtenir des meilleures performances. La possibilité de modifier dynamiquement la politique de placement globale de l'application afin de l'adapter aux contraintes de chaque noyau apparaît dès lors comme quelque chose de souhaitable. Utiliser une politique globale de type «plus petit dénominateur commun» est insuffisant pour obtenir de bonnes performances tandis que choisir la politique d'un noyau en particulier risque de désavantager les autres.

Au final, il est clair que cette problématique importante déborde du cadre de MPI et demande une prise en charge globale dans l'écosystème HPC. Pour autant, cela ne signifie pas que des initiatives ne devraient pas être prises dans le standard pour commencer à s'attaquer aux problèmes constatés, à la fois en termes de performances mais également en termes pratiques comme la portabilité du déploiement des applications. Le CEA est d'ailleurs à l'inititative de la mise en place d'un groupe de travail international consacré au placement des processus et des threads auquel nous participons activement depuis plusieurs mois maintenant. Des ramifications des solutions proposées dans le cadre de ce groupe de travail pourraient trouver un débouché dans MPI.

3.3 Un environnement de déploiement et d'exécution d'applications *locality-aware*

Dans cette section, nous allons détailler les travaux sur le placement que j'ai initiés en 2009 [CI10] ainsi que leur évolution et plus généralement l'impact dans l'EPC Inria TADaaM pour laquelle la localité est devenue l'axe priviliégié de recherches. Ainsi que détaillé en section 3.2.2.3, la mise en œuvre d'une politique efficace de placement peut se faire en suivant un ensemble d'étapes. Nous allons donc décrire chaque étape dans les sections suivantes, en montrant bien l'évolution dans l'appréhension de chacune et les travaux de recherche que cela a permis d'inciter. Ainsi nous verrons :

— pour l'étape 1 : les méthodes que nous avons employées pour récupérer la topologie virtuelle de l'application et les outils développés pour y arriver ;
— pour l'étape 2 : notre participation à la proposition de création d'une interface modélisant de façon portable la topologie matérielle ;
— pour l'étape 3 : comment nos travaux ont entraîné la mise au point d'un nouvel algorithme de placement ;
— pour l'étape 4 : l'intégration dans les implémentations de MPI des éléments précédents pour appliquer un placement ou une renumérotation de processus débouchant sur de meilleures performances applicatives.

Il est important de préciser que tous les environnements travaillant sur la localité dans MPI suivent cette approche c'est-à-dire cette succession d'étapes. Les différences entre les solutions proposées se situent dans les outils, techniques et algorithmes utilisés à chaque étape. Également, certains travaux se concentrent davantage sur une étape particulière, par exemple la troisième pour les heuristiques et algorithmes de placement vus précédemment, tandis que les extensions de gestionnaires de processus ou de lanceurs d'applications (ex : LAMA) ne concernent que la quatrième étape.

3.3.1 Étape 1 : récupération des informations de topologie virtuelle

3.3.1.1 À propos de la topologie virtuelle

La première étape concerne donc la récupération de la topologie virtuelle. La caractérisation de cette topologie virtuelle, peut se baser sur des critères différents selon le paradigme de programmation utilisé, ce qui fait que la méthode décrite ici est généralisable et donc pas uniquement valable pour le passage de messages, qui ne constitue qu'un contexte d'application. Dans le cas de communications implicites, par exemple dans les applications utilisant des threads, le comportement de ces dernières peut être caractérisé par les accès des threads aux pages mémoire du processus. Dans le cas de communications explicites (cas du passage de

messages), il est possible de caractériser le schéma de communication par le nombre de messages échangés ou bien par le nombre d'octets échangés. L'essentiel est de disposer d'éléments quantitatifs représentatifs permettant la pondération du graphe de topologie virtuelle. Dans le cas d'un placement ou d'une renumérotation statique, effectué au moment du déploiement ou au début de l'exécution de l'application, un seul schéma de communication (i.e un seul jeu de valeurs) sera utilisé, ce qui signifie que la notion de temporalité est perdue. Les schémas de communication applicatifs font l'objet de travaux parcimonieux. Ma *et al.* [131] introduit la notion de distance de schéma de communication, de façon à pouvoir créer des classes d'applications. Ainsi, il serait possible d'utiliser non pas le schéma spécifique d'une application mais plutôt un schéma générique, nonobstant proche de ce qu'il devrait être en réalité. Cette approche permet d'économiser la première étape puisque qu'une bibliothèque de schémas applicatifs permettrait d'éviter la détermination d'une information précise pour chaque application.

3.3.1.2 Approches possibles pour la récupération des informations

Normalement, cette information de topologie virtuelle devrait être fournie par l'utilisateur, qui est le plus à même de connaître le comportement et le fonctionnement de son application. Si l'utilisateur ne peut communiquer cette information, il faut trouver un moyen de la déterminer. Cette récupération peut (pourrait) s'effectuer par plusieurs moyens :

- par l'exécution préliminaire d'une version instrumentée de l'application permettant de récupérer des informations dont le degré de précision dépend de l'outil d'instrumentation employé. Cette technique, bien que souvent critiquée (rien ne dit que le schéma de communication récupéré lors de cette exécution préliminaire sera encore pertinent pour les suivantes), est pourtant très souvent employée (ex : MPIPP, CrayPAT, etc), parce qu'elle est relativement simple à mettre en place. Les problèmes posés par cette approche sont d'une part le temps qui est pris pour cette exécution préliminaire et la taille de la trace collectée. Pour ce dernier point, il existe des techniques de compression de traces qui permettent de limiter le problème (cf. CYPRESS [217] par exemple) ;
- par l'exécution préliminaire d'une version modifiée de l'application d'origine. C'est par exemple l'approche suivie dans FACT [216], qui appartient à l'environnement MPIPP. Cela permet de réduire le temps de l'exécution préliminaire puisque seules les phases de communication sont conservées et que les phases de calcul sont simplifiées ;
- par la simulation de l'application dans un environnement réduit. C'est l'approche suivie par PHANTOM [215], développé par la même équipe responsable de FACT et MPIPP. Cela permet de réduire davantage encore le temps d'exécution du *run* préliminaire, ce qui est intéressant pour les applications de longue durée ou bien dont l'échelle est trop importante pour pouvoir se permettre cette fameuse exécution préliminaire ;
- par la génération à la volée des données en appliquant un profilage sélectif sur une partie de l'application comme proposé par Jeannot *et. al* [100] ;
- par l'extraction d'information de façon statique, à la compilation de l'application. Cette technique n'est que peu employée pour effectuer du placement (cf. McPherson *et al.* [139] qui l'utilise mais pour un placement dans une hiérarchie à deux niveaux seulement). La difficulté réside à la fois dans la quantité et la pertinence des informations qu'il serait possible d'extraire statiquement. Je pense néanmoins qu'il s'agit d'une piste intéressante à davantage explorer.

Il est à noter que la plupart de ces techniques font l'hypothèse que le schéma de communication applicatif reste le même d'une exécution sur l'autre.

3.3.1.3 Approche retenue et implémentée

Nous avons opté pour la technique la plus simple reposant sur une exécution préliminaire d'une version instrumentée de l'application cible. La question du choix de l'outil d'instrumentation est importante pour une collecte d'informations fiables. Il existe un certain nombre d'outils pré-existants mais qui présentent des défauts selon moi car les informations relatives aux opérations collectives ne sont pas complètement disponibles, alors que ce type de communications est très utilisé dans les applications MPI. De plus, ces outils ne permettent pas d'avoir un accès à une métrique basique de localité comme la répartition entre communications inter-nœuds et intra-nœuds, qui permet une première évaluation de la politique de placement choisie.

Des modifications dans MPICH2-NEMESIS Toutes ces considérations m'ont poussé à développer mon propre outil pour d'une part récupérer une trace précise d'exécution (le *run* préliminaire) et d'autre part pour obtenir des informations sur l'exécution réelle de l'application afin d'analyser les résultats des politiques mises en œuvre. Étant donnée ma forte implication dans la conception et la réalisation de NEMESIS et de certains de ses modules réseaux, j'ai pris la décision de modifier MPICH2. J'ai donc procédé à une instrumentation assez légère (pour ne pas trop perturber le comportement de l'implémentation de MPI et de l'application à observer) du *channel* NEMESIS ainsi que des modules réseaux supportant le *Tag Matching* (cf. section 2.5.3.5), c'est-à-dire les modules Myrinet/MX et NEWMADELEINE. Comme NEWMADELEINE est une bibliothèque générique de communication supportant un large panel de technologies d'interconnexion, cela ne restreint pas nos possibilités. De toutes façons, cela n'aurait posé véritablement de problème que pour l'analyse des résultats de l'application suite à la mise en place de la politique de placement car le schéma de communication peut être déterminé sur n'importe quelle plate-forme à partir du moment où les données et le nombre de processus est conforme aux exécutions suivantes.

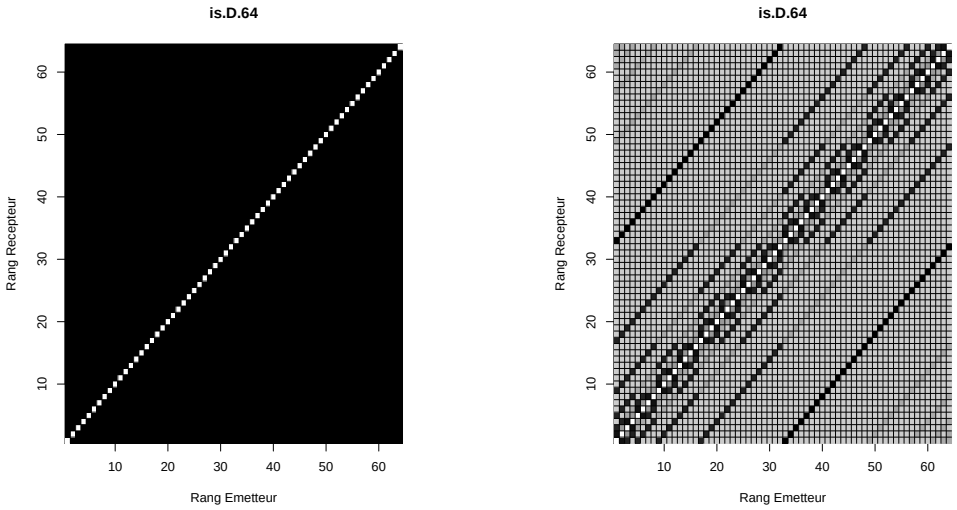
Données de topologie virtuelle et métriques associées Grâce à l'approche choisie décrite précédemment, nous avons gagné l'accès à certaines informations :

- le volume total des données échangées au niveau applicatif car les opérations collectives sont décomposées en opérations point à point (nous ne sommes pas dans le cas où de telles opérations sont supportées directement par le matériel) ;
- la répartition entre les communications inter-nœuds et les communications intra-nœuds ;
- la répartition entre les données applicatives proprement dites et les données internes propres à l'implémentation MPI (émanant des protocoles de communication utilisés en interne).

La figure 3.5 montre la façon dont les informations sont affichées par notre outil de trace. Deux types de métriques sont utilisables pour exprimer le schéma de communication : le volume total des données et le nombre total de messages. Il y a une duplication de l'information exprimée pour chaque paire de processus : les communications entrantes sont marquées `R` tandis que les communications sortantes sont marquées `S`. Les deux colonnes suivantes sont les rangs des processus impliqués dans la communication. Conformément à ce que nous avons expliqué en section 3.2.1.1, ces numéros de rangs sont ceux dans `MPI_COMM_WORLD`. La possibilité d'utiliser plusieurs métriques pour instancier le schéma de communication applicatif est importante. En effet, les informations sont plus ou moins détaillées selon la métrique choisie comme le montre clairement la figure 3.6 où le volume de données échangées est à peu près identique quelle que soit la paire de processus considérée, tandis que le nombre de messages dessine plus clairement la localité de l'application. Nous avons repris le principe de ce travail

```
S       8       15          557090240 bytes          2011 msgs sent
S       8       16         1507275840 bytes          2011 msgs sent
S       8       23         1507275840 bytes          2011 msgs sent
S       8       24                976 bytes             8 msgs sent
S       8       40               1104 bytes             8 msgs sent
R       8       0          1507276752 bytes          2022 msgs received
R       8       1          1507275840 bytes          2011 msgs received
R       8       9           557091104 bytes          2018 msgs received
R       8       10                872 bytes             9 msgs received
R       8       12                888 bytes             9 msgs received
R       8       15         1507275840 bytes          2011 msgs received
R       8       16          557090240 bytes          2009 msgs received
R       8       23          557090240 bytes          2011 msgs received
R       8       24                976 bytes             6 msgs received
```

FIGURE 3.5 – Extrait d'un schéma de communication.



(a) Volume total de données (b) Nombre de messages

FIGURE 3.6 – Schémas de communication pour le noyau NAS IS (Classe D, 64 processus)

effectuée pour MPICH2 afin de le transposer dans Open MPI [CI2] ultérieurement.

Données d'analyse La figure 3.7 montre les informations d'analyse obtenues également par mon outil de génération de traces. Pour chaque processus (avec son rang dans `MPI_COMM_WORLD` là encore), on dispose de deux informations : la quantité de données échangées par ce processus (émission et réception) et le volume de communications réseau pour le processus en question, ce qui permet de calculer le ratio des communications intra-nœuds.

Afin de mieux comprendre le comportement des applications et en particulier de mieux identifier les parties de MPI les plus sollicitées, j'ai également développé un module MPE2 (*MPI Profiling Environment 2*), qui est activable dynamiquement, permettant d'obtenir deux informations supplémentaires : d'une part le temps total passé dans une fonction MPI particulière. Ainsi, il est possible de comprendre l'impact du placement sur l'utilisation de la bibliothèque de communication. La figure 3.8 montre le bilan obtenu suite à l'exécution d'une application. Il est également possible de déterminer le ratio du temps passé dans MPI par rapport au temps total d'exécution. Étant donné que le but du placement est de diminuer le temps

```
[--  6--] NET_comms:   8257469600 bytes   total_comms:  12386207744 bytes
[-- 15--] NET_comms:   8257469560 bytes   total_comms:  12386207704 bytes
[--  4--] NET_comms:   8257469640 bytes   total_comms:  12386207784 bytes
[-- 14--] NET_comms:   8257469600 bytes   total_comms:  12386207744 bytes
[-- 27--] NET_comms:   8257469560 bytes   total_comms:  12386207704 bytes
[-- 16--] NET_comms:   8257469640 bytes   total_comms:  12386207784 bytes
[--  1--] NET_comms:   8257469560 bytes   total_comms:  12386207704 bytes
[-- 11--] NET_comms:   8257469560 bytes   total_comms:  12386207704 bytes
[-- 40--] NET_comms:   8257469640 bytes   total_comms:  12386207784 bytes
[-- 18--] NET_comms:   8257469600 bytes   total_comms:  12386207744 bytes
[-- 20--] NET_comms:   8257469640 bytes   total_comms:  12386207784 bytes
[--  7--] NET_comms:   8257469560 bytes   total_comms:  12386207704 bytes
```

FIGURE 3.7 – Extrait de données d'analyse : communications inter-nœuds et total des communications

passé dans les communications, cela donne une autre métrique d'évaluation de la politique de placement mise en place.

Par ailleurs, j'ai enrichi le jeu d'informations d'analyse en ajoutant une gestion de compteurs matériels dans les *wrappers* MPE2 avec la bibliothèque PAPI [28]. En revanche, les temps calculés dans ce cas ne sont pas représentatifs car le surcoût associé à la bibliothèque PAPI est comptabilisé dans le temps passé dans MPI. La figure 3.8 montre un résultat avec les accès au cache L3 et les cycles perdus en attente de ressources (*Resource Stalls*). Le principal problème rencontré avec cette gestion des compteurs matériels est leur disponibilité fluctuante selon les processeurs utilisés, ne permettant pas toujours d'effectuer des analyses complètes relatives à l'impact du placement sur ces événements.

En conclusion, nous disposons d'outils pratiques collectant les informations de topologie virtuelle et permettant l'analyse des performances applicatives pour mieux comprendre l'influence de la politique de placement appliquée. Cependant, il est clair que le principe du *run* préliminaire est loin d'être idéal (même si massivement utilisé par ailleurs) et devrait être amélioré.

```
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>: *** Total time                               = 140.487157
[1,0]<stdout>: *** ---- Time spent in MPI                   = 22.240209
[1,0]<stdout>: *** ---- Time spent in computation          = 118.246948
[1,0]<stdout>: *** ---- Ratio comm/total                   = 15.830777
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>: *** pt2pt - time in MPI_Send                 = 0.020532
[1,0]<stdout>: *** pt2pt - time in MPI_Bsend                = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Ssend                = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Rsend                = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_ISend                = 0.189534
[1,0]<stdout>: *** pt2pt - time in MPI_Ibsend               = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Issend               = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Irsend               = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Send_init            = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Bsend_init           = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Ssend_init           = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Rsend_init           = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Recv                 = 12.810225
[1,0]<stdout>: *** pt2pt - time in MPI_Irecv                = 0.038964
[1,0]<stdout>: *** pt2pt - time in MPI_Recv_init            = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Sendrecv             = 0.000000
[1,0]<stdout>: *** pt2pt - time in MPI_Sendrecv_replace     = 0.000000
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>: *** test -- time in MPI_Wait                 = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Waitall              = 3.418353
[1,0]<stdout>: *** test -- time in MPI_Waitany              = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Waitsome             = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Test                 = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Testall              = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Testany              = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Testsome             = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Probe                = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Iprobe               = 0.011713
[1,0]<stdout>: *** test -- time in MPI_Start                = 0.000000
[1,0]<stdout>: *** test -- time in MPI_Start_all            = 0.000000
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>: *** coll -- time in MPI_Allgather            = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Allgatherv           = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Allreduce            = 5.201132
[1,0]<stdout>: *** coll -- time in MPI_Alltoall             = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Alltoallv            = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Barrier              = 0.228151
[1,0]<stdout>: *** coll -- time in MPI_Bcast                = 0.000213
[1,0]<stdout>: *** coll -- time in MPI_Gather               = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Gatherv              = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Scatter              = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Scatterv             = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Reduce               = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Reduce_scatter       = 0.000000
[1,0]<stdout>: *** coll -- time in MPI_Scan                 = 0.000000
[1,0]<stdout>:**** ========================================== ****
[1,0]<stdout>: *** topo -- time in MPI_Dist_Graph_create    = 0.321393
[1,0]<stdout>:**** ========================================== ****
```

FIGURE 3.8 – Extrait de traces d'analyse : temps passé dans MPI et ratio calcul/communication

```
**** ========================================== ****
 *** Total_Time                          = 225.328214
 *** ---- Time spent in MPI              = 83.193468
 *** ---- Time spent in computation      = 142.134746
 *** ---- Ratio comm/total               = 36.921017
**** ========================================== ****
 *** pt2pt - time in MPI_Send            = 0.031346    PAPI_RES_STL   13622352    PAPI_L3_TCM     306771    PAPI_L3_TCA    1238549
 *** pt2pt - time in MPI_Bsend           = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Ssend           = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Rsend           = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_ISend           = 0.236890    PAPI_RES_STL  211333650    PAPI_L3_TCM    5139899    PAPI_L3_TCA   14069466
 *** pt2pt - time in MPI_Ibsend          = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Issend          = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Irsend          = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Send_init       = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Bsend_init      = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Ssend_init      = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Rsend_init      = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Recv            = 39.044062   PAPI_RES_STL 31378943812    PAPI_L3_TCM     261233    PAPI_L3_TCA    1694792
 *** pt2pt - time in MPI_Irecv           = 0.062205    PAPI_RES_STL   79301185    PAPI_L3_TCM    1048523    PAPI_L3_TCA    4017017
 *** pt2pt - time in MPI_Recv_init       = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Sendrecv        = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** pt2pt - time in MPI_Sendrecv_replace = 0.000000   PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
**** ========================================== ****
 *** test -- time in MPI_Wait            = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Waitall         = 5.803646    PAPI_RES_STL 5361357523    PAPI_L3_TCM   18444934    PAPI_L3_TCA   45931655
 *** test -- time in MPI_Waitany         = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Waitsome        = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Test            = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Testall         = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Testany         = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Testsome        = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Probe           = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Iprobe          = 0.014801    PAPI_RES_STL   15633459    PAPI_L3_TCM     129135    PAPI_L3_TCA    1748055
 *** test -- time in MPI_Start           = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** test -- time in MPI_Start_all       = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
**** ========================================== ****
 *** coll -- time in MPI_Allgather       = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Allgatherv      = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Allreduce       = 37.556544   PAPI_RES_STL 34605199775    PAPI_L3_TCM    4249631    PAPI_L3_TCA   31472291
 *** coll -- time in MPI_Alltoall        = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Alltoallv       = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Barrier         = 0.144392    PAPI_RES_STL   86048607    PAPI_L3_TCM       2487    PAPI_L3_TCA       4055
 *** coll -- time in MPI_Bcast           = 0.000033    PAPI_RES_STL      13638    PAPI_L3_TCM        110    PAPI_L3_TCA        650
 *** coll -- time in MPI_Gather          = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Gatherv         = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Scatter         = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Scatterv        = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Reduce          = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Reduce_scatter  = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
 *** coll -- time in MPI_Scan            = 0.000000    PAPI_RES_STL          0    PAPI_L3_TCM          0    PAPI_L3_TCA          0
**** ========================================== ****
 *** topo -- time in MPI_Dist_Graph_create = 0.299547  PAPI_RES_STL   69013457    PAPI_L3_TCM      25904    PAPI_L3_TCA     147959
```

FIGURE 3.9 – Extrait de données d'analyse : compteurs matériels

3.3.2 Étape 2 : récolte des informations de topologie matérielle

3.3.2.1 Approches possibles pour la récupération des informations

La deuxième étape consiste à récupérer des informations concernant le matériel sous-jacent afin d'établir le graphe de topologie matérielle. Tout comme la première étape, plusieurs approches sont possibles : d'une part l'approche statique qui consiste à établir une structure de données représentant la topologie matérielle une fois pour toutes. Cette structure de données (ou ce fichier de configuration) n'est ensuite plus modifié ni consulté pour le calcul du placement. C'est par exemple l'approche retenue par MPIPP ou celle que j'ai adoptée dans mes premiers travaux où je modélisais l'architecture avec une simple matrice de topologie matérielle. L'approche dynamique repose sur l'utilisation d'un élément logiciel, avec une interface bien déterminée, permettant l'utilisation et la consultation des informations de topologie matérielle.

Également, il y a deux façons principales de pondérer le graphe de topologie matérielle (cf section 2.2.2). La première, dite *quantitative*, repose sur l'utilisation de données numériques caractérisant le matériel comme la latence, la bande passante, la taille de la ressource, etc. Une telle approche est censée refléter au mieux la réalité du matériel. Cependant, il n'est pas toujours aisé de récupérer ces informations de façon fiable et qui de plus sont susceptibles de changer durant l'exécution d'une application. Par exemple, une partie du réseau peut devenir congestionnée par la présence d'une autre application alors que ce n'était pas le cas au moment de la mesure de performance utilisée pour la pondération. La seconde approche, dite *qualitative*, utilise également des valeurs numériques, mais plutôt censées représenter des différences de performances. Cette approche exprime de façon plus synthétique la hiérarchie des performances des canaux de communication disponibles[9]. Par exemple, l'architecture d'un processeur Opteron (figure 3.10) peut être représentée qualitativement par une matrice comme celle de la figure 3.11 avec des coéfficients choisis «aléatoirement». Enfin, il faut noter que ces deux approches ne sont pas mutuellement exclusives.

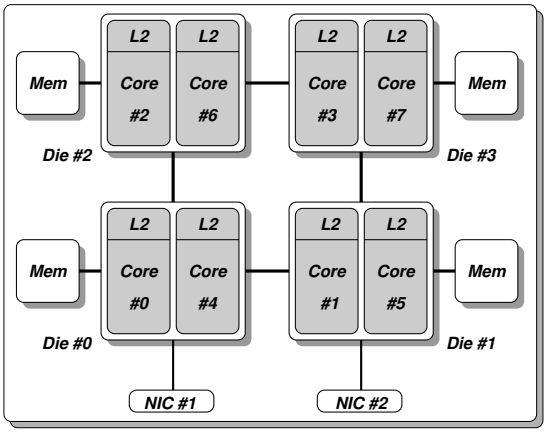


FIGURE 3.10 – Un processeur Opteron.

3.3.2.2 La problème de la numérotation des ressources

En section 3.1.6, nous avons expliqué que les ressources de calculs pouvaient être désignées de façon logique ou bien de façon physique. Or, l'utilisation de cette numérotation physique

9. C'est l'approche que j'ai adoptée en 2009.

$$Machine = \begin{pmatrix} 0 & 100 & 100 & 10 & 1000 & 100 & 100 & 10 \\ 100 & 0 & 10 & 100 & 100 & 1000 & 10 & 100 \\ 100 & 10 & 0 & 100 & 100 & 10 & 1000 & 100 \\ 10 & 100 & 100 & 0 & 10 & 100 & 100 & 1000 \\ 1000 & 100 & 100 & 10 & 0 & 100 & 100 & 10 \\ 100 & 1000 & 10 & 100 & 100 & 0 & 10 & 100 \\ 100 & 10 & 1000 & 100 & 100 & 10 & 0 & 100 \\ 10 & 100 & 100 & 1000 & 10 & 100 & 100 & 0 \end{pmatrix}$$

FIGURE 3.11 – Représentation qualitative du processeur Opteron de la figure 3.10. *Machine(i,j)* représente la vitesse des communications entre les cœurs C_i and C_j.

n'est pas pratique et ne permet pas d'être portable. En effet, deux machines matériellement identiques peuvent être vues différemment par les applications car les ressources matérielles sont numérotées par le BIOS (voire par le système d'exploitation). Une mise à jour logicielle peut donc entraîner un changement dans la numérotation et rendre caduque la politique de placement calculée avant cette mise à jour. La raison justifiant cette variation dans la numérotation est que les architectes des BIOS essaient de proposer une numérotation qui soit utile aux applications ainsi qu'aux systèmes d'exploitation qui sont incapables de s'adapter à la topologie matérielle.

Comme ces applications vont utiliser prioritairement les *premières* ressources de calcul, les constructeurs vont tenter de numéroter en *premier* des ressources répondant bien aux besoins des applications. Cela peut se traduire par l'utilisation de cœurs de différents nœuds NUMA (pour maximiser la bande passante mémoire), par l'utilisation d'un unique thread matériel par cœur (pour éviter que les autres threads ne le perturbent), etc. Par exemple, sur la figure 3.12 les unités physiques d'exécutions (ou PU pour *Processing Units*) sont numérotées horizontalement d'abord par processeur, puis par cœur, puis par nœud NUMA. Cependant, les besoins des applications étant variables, les solutions choisies par les constructeurs le sont également beaucoup. Le schéma de numérotation est donc susceptible de fortement varier d'une machine à l'autre.

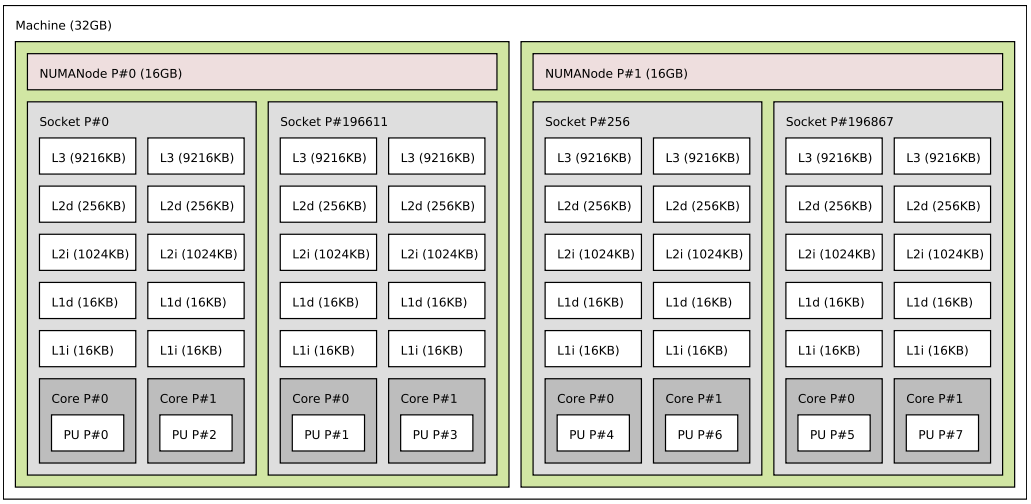


FIGURE 3.12 – Une machine quadriprocesseurs Intel Itanium 2 *Montecito* 9040 bicœurs.

3.3.2.3 Un peu d'histoire : création d'hwloc

Dans mes premiers travaux consacrés au placement (2009), j'ai adopté une approche statique avec une simple matrice décrivant la topologie matérielle (cf. figure 3.11). Les poids dans la matrice indiquaient quant à eux une forme de hiérarchie au niveaux des performances entre les différents cœurs de la machine (approche plutôt qualitative, donc). De plus le choix des poids se devait d'être réfléchi car le problème de plongement de graphe associé aux matrices de topologies virtuelles et matérielles était résolu à l'aide du partitionneur Scotch qui n'accepte pas des valeurs totalement arbitraires pour arriver à effectuer une coupe adéquate.

À cette époque, les valeurs de la matrice de topologie virtuelle sont obtenues à l'aide d'un outil de détection des topologies matérielles intégré dans le logiciel PM^2 [148]. En effet, durant sa thèse, Vincent DANJEAN qui travaillait sur un ordonnanceur mixte de threads, avait développé un élément logiciel permettant la détection de machines SMP. Ce code a d'ailleurs été intégré dans MPICH/Madeleine afin d'activer dynamiquement le support des communications par mémoire partagée pour une meilleure progression des communications. Le code de ce *proto*-hwloc était le suivant :

```
  /*  Determination du nombre de processeurs disponibles */
   # ifdef SOLARIS_SYS
  _nb_processors = sysconf(_SC_NPROCESSORS_CONF);
  # elif defined(LINUX_SYS)
  _nb_processors = sysconf(_SC_NPROCESSORS_CONF);
  # elif defined(IRIX_SYS)
  _nb_processors = sysconf(_SC_NPROC_CONF);
  # elif defined(OSF_SYS)
  _nb_processors = sysconf(_SC_NPROCESSORS_CONF);
  # else
  _nb_processors = 2;
  # endif
  smp_processes = MPID_CH_MAD_SMPSETRANKS();
}
```

Par la suite, Samuel THIBAULT, a continué à travailler sur l'ordonnancement des threads, en s'intéressant plus particulièrement aux machines hiérarchiques multicœurs [175]. Il a donc étendu le code de détection de topologie matérielle pour découvrir les nœuds NUMA, les processeurs, les cœurs ainsi que les threads matériels. Ce module de détection de topologie, toujours intégré dans PM^2, offrait une certaine portabilité mais était incomplet. Brice GOGLIN a amélioré l'implémentation en utilisant de nouvelles sources d'information de topologie, par exemple les fichiers virtuels `sysfs` sous Linux, et a rajouté la détection des mémoires caches. Il a de plus écrit un outil qui permet un affichage (en mode texte) de la topologie de la machine. Un tel exemple d'affichage est donné par la figure 3.13, qui correspond en fait au processeur Opteron de la figure 3.10. C'est à l'aide de cet affichage que je rentrais manuellement des valeurs dans la matrice de topologie matérielle. Mais la structure de données associée était donc statique (voire rigide) ce qui n'était pas satisfaisant. J'ai eu la conviction qu'un tel module pourrait fournir de grands services, non seulement pour travailler sur le placement, mais bien au-delà. J'ai donc émis l'idée de son externalisation afin d'en faire un logiciel autonome. Samuel, Brice et Jérôme CLET-ORTEGA ont entrepris ce travail et *Hardware Locality* (hwloc) était né, d'abord sous le nom de `libtopology` [CI7]. Ce travail a rapidement été remarqué par les développeurs d'Open MPI qui disposaient d'un projet concurrent dénommé PLPA (*Portable Li-*

```
Machine:
 NUMANode + Die: Node#0(8GB) Die#0
  L2Cache + Core + L1Cache + SMTproc : L2#0(1MB) Core#0 L1#0(64kB) CPU#0
  L2Cache + Core + L1Cache + SMTproc : L2#4(1MB) Core#1 L1#4(64kB) CPU#4
 NUMANode + Die: Node#1(8GB) Die#1
  L2Cache + Core + L1Cache + SMTproc : L2#1(1MB) Core#0 L1#1(64kB) CPU#1
  L2Cache + Core + L1Cache + SMTproc : L2#5(1MB) Core#1 L1#5(64kB) CPU#5
 NUMANode + Die: Node#2(8GB) Die#2
  L2Cache + Core + L1Cache + SMTproc : L2#2(1MB) Core#0 L1#2(64kB) CPU#2
  L2Cache + Core + L1Cache + SMTproc : L2#6(1MB) Core#1 L1#6(64kB) CPU#6
 NUMANode + Die: Node#3(8GB) Die#3
  L2Cache + Core + L1Cache + SMTproc : L2#3(1MB) Core#0 L1#3(64kB) CPU#3
  L2Cache + Core + L1Cache + SMTproc : L2#7(1MB) Core#1 L1#7(64kB) CPU#7
```

FIGURE 3.13 – Informations de topologie matérielle sous forme graphique (mécanisme de découverte de topologie de PM^2

nux Processor Affinity) [163] mais dont ils commençaient à percevoir des limites critiques (uniquement sous Linux, nœuds NUMA et caches ignorés). Les deux projets ont fusionné pour aboutir sur hwloc. Depuis, hwloc a également été intégré à MPICH (travail que j'ai effectué dans le cadre de l'Équipe Associée Inria MPI-Runtime) et dans de très nombreux autres logiciels, notamment la plupart des implémentations MPI mais aussi des supports exécutifs, des gestionnaires de ressources ou encore des bibliothèques numériques. Il n'est donc pas galvaudé d'affirmer qu'hwloc est rapidement devenu un logiciel incontournable dans le HPC [10], preuve que l'intuition de départ était la bonne. hwloc a éclipsé des solutions concurrentes antérieures telles que Libnuma [108] ou Pthread sched [151]. Il existe cependant des solutions ad-hoc pour certaines architectures et qui sont donc utilisées dans des situations particulières, comme TopoMgr [16] pour les machines Cray XT/XE et BlueGene d'IBM.

3.3.2.4 Organisation logique des ressources dans hwloc

Le succès de hwloc provient de la connaissance avancée du matériel qu'il propose, mais également de la façon dont il l'expose à travers un ensemble d'outils en ligne de commande mais surtout par le truchement d'une interface de programmation permettant une manipulation concrète de quelque chose d'assez abstrait à l'origine [11]. hwloc représente la topologie matérielle sous la forme d'une structure hiérarchique arborescente, comme schématisé sur la figure 3.14. Les ressources de calcul et mémoire sont organisées en arbre selon un ordre d'inclusion. L'ensemble des objets de même type (tous les nœuds NUMA, tous les processeurs, tous les caches L1i, etc.) sont placés et chaînés sur un même niveau horizontal, ce qui permet un parcours aisé, quand bien même la topologie serait asymétrique (par exemple, si un processeur contient moins de niveaux de cache qu'un autre). Aucun ordre n'est imposé entre les niveaux, ce qui permet d'éviter des problèmes de disposition de ressources. L'application peut simplement demander où se trouve le niveau NUMA, ou quel type d'objet se trouve au-dessus ou au-dessous des *packages*.

Ce sont ces niveaux qui servent d'ailleurs à renuméroter les unités physiques d'exécution de façon logique. Les cœurs dont les numéros logiques se suivent seront donc physiquement proches, contrairement aux numérotations compliquées et variables évoquées en section 3.3.2.2. Chaque objet dans l'arbre contient par ailleurs des attributs spécifiques comme par exemple la taille et l'associativité s'il s'agit d'un cache. La modélisation en arbre est donc

10. Pour s'en convaincre, il suffit de regarder la liste des projets utilisant hwloc sur le site du projet : https://www.open-mpi.org/projects/hwloc/.
11. C'est-à-dire une réification en quelque sorte.

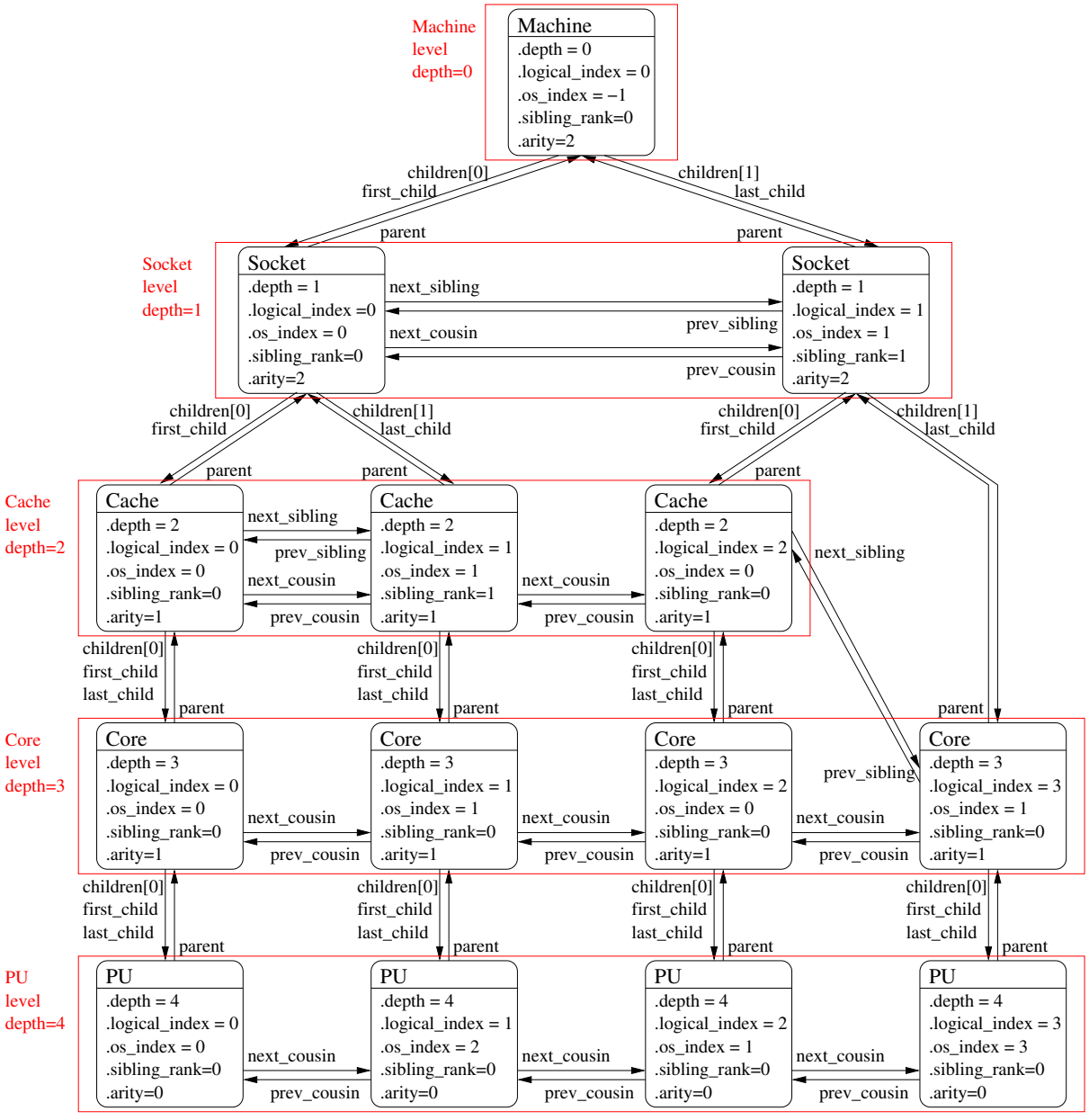


FIGURE 3.14 – Organisation arborescente d'une topologie matérielle dans hwloc.

simple mais n'interdit pas à l'application d'utiliser des informations bas-niveau sur le matériel.

3.3.3 Étape 3 : appariement de la topologie virtuelle avec la topologie matérielle

3.3.3.1 Genèse de TreeMatch

Dans mes premiers travaux sur le placement de 2009, j'ai commencé par collaborer avec François Pellegrini et Sébastien Fourestier pour la résolution du problème de plongement de graphe induit par celui du placement que je cherchais à résoudre. La solution retenue fût celle de l'utilisation du partitionneur Scotch, outil plutôt dédié au partitionnement de graphes génériques de grande taille. À son arrivée dans Runtime, Emmanuel Jeannot fit alors le constat que la méthode utilisée n'était sans doute pas la plus adéquate au regard du problème considéré : Scotch est destiné à des graphes quelconques de grande taille alors que nous visions des arbres de taille bien moindre (cf. section 3.1.5.1). C'est ainsi qu'Emmanuel et moi-même avons décidé de travailler sur la mise au point d'un nouvel algorithme de calcul de placement, dénommé TreeMatch, qui verra le jour peu de temps après [CI8]. Les idées-forces de TreeMatch sont d'une part l'adoption d'une approche **ascendante** là où les solutions existantes étaient plutôt **descendantes** (comme Scotch qui partitionne des graphes de plus en plus petits pour arriver à la solution). D'autre part, TreeMatch fonctionne de façon **qualitative** : en effet c'est la *structure* de l'arbre de topologie matérielle qui est utilisée plutôt que des mesures précises des capacités du matériel sous-jacent. Enfin, le parti-pris adopté a été de s'efforcer d'atteindre des bonnes performances dans le contexte spécifique des arbres, car nous nous concentrions explicitement sur le placement des processus dans des nœuds hiérarchiques multicœurs.

TreeMatch [RI3] est rapidement devenu une référence dans le domaine du placement et nombre de concurrents lui ont emboîté le pas et apparaissent régulièrement. On citera par exemple LibTopoMap [86] qui se base sur de l'isomorphisme de graphe et utilise diverses heuristiques (bipartitionnement récursif, $k-$partitionnement, stratégies gloutonnes et Cuthill McKee [47]) pour implémenter – notamment – l'interface des topologies virtuelles de MPI 2.2 [88]. Plus récemment, on trouve HierTopoMap [209] qui est basé sur du partitionnement récursif, ParMapper qui fait partie de l'environnement TopoMapping [66] et qui détermine de façon parallèle le meilleur placement selon différents critères, APHiD [141] qui propose de placer des processus hiérarchiquement sur des arbres $n-$aires, ClustMap [137] et enfin EagerMap [46]. La thèse de François Tessier que j'ai co-encadrée avec Emmanuel Jeannot a notamment porté sur TreeMatch et a permis d'étendre les champs d'application de TreeMatch à des problématique d'équilibrage de charge ([CI5] et [AI2]).

3.3.3.2 Présentation générale de l'algorithme TreeMatch

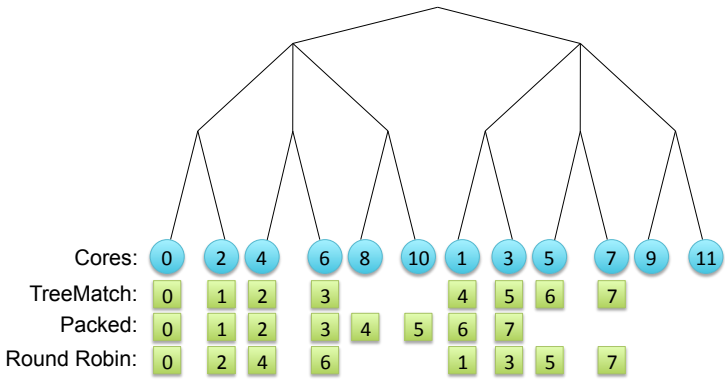
TreeMatch agit en tant que fonction de placement telle que décrite en section 3.1.3. La finalité de TreeMatch est donc l'obtention d'un numéro logique de ressource physique de calcul dans le but d'y affecter une entité logique d'exécution (processus ou thread) ou bien l'obtention d'un nouveau numéro de rang pour effectuer une renumérotation (cas particulier des applications MPI). Comme nous nous plaçons dans le cadre du passage de messages, les métriques à optimiser seront basées sur les coûts de communication associés aux échanges entre les ressources physiques de calcul. TreeMatch optimise notamment le *Hop-Byte* (cf. la définition en 3.1.3).

L'algorithme général est présenté par l'Algorithme 1 dont nous allons montrer le fonctionnement à l'aide d'un exemple. Soient la topologie matérielle représentée par la figure 3.15(a) et

un schéma de communication représenté par la matrice de la figure 3.15(b). La topologie matérielle est équilibrée (i.e. toutes les feuilles sont à la même profondeur) et symétrique (i.e. tous les nœuds de même profondeur ont la même arité). Cette topologie possède quatre niveaux de profondeur et douze feuilles qui correspondent aux unités physiques d'exécution (e.g. des cœurs). La matrice de communication, de taille 8×8, représente les communications effectuées entre chaque paire d'unités virtuelles d'exécution (e.g. des processus). Dans la version présentée ici, il n'y a pas obligatoirement de bijection entre les unités virtuelles et les unités physiques : une injection est suffisante (i.e. pas d'*oversubscribing*) mais la surjection n'est pas obligatoire non plus (i.e. toutes les unités physiques ne sont pas forcément utilisées). Au final le nombre d'unités virtuelles d'exécution est inférieur ou égal au nombre d'unités physiques d'exécution.

Algorithm 1: L'algorithme TREEMATCH

```
   Input: T // L'arbre représentant la topologie
   Input: m // La matrice de communication
   Input: D // La profondeur de l'arbre
1  groups[1..D − 1]=∅ // Regroupement des nœuds à chaque niveau
2  foreach depth← D − 1..1 do // Démarrage par les feuilles
3      p ← order of m
       // Extension de la matrice de communication si nécessaire
4      if p mod arity(T, depth − 1) ≠ 0 then
5          m ←ExtendComMatrix(T,m,depth)
6      groups[depth]←GroupProcesses(T,m,depth)// Regroupement des entités par affinité
7      m ←AggregateComMatrix(m,groups[depth]) // Agrégation des communications des groupes
         d'entités
8  MapGroups(T,groups) // Construction de la permutation
```



(a) Arbre de la topologie matérielle (les carrés représentent les processus assignés par les différentes politiques de placement (cf. section 3.1.2.3)).

Proc	0	1	2	3	4	5	6	7
0	0	1000	10	1	100	1	1	1
1	1000	0	1000	1	1	100	1	1
2	10	1000	0	1000	1	1	100	1
3	1	1	1000	0	1	1	1	100
4	100	1	1	1	0	1000	10	1
5	1	100	1	1	1000	0	1000	1
6	1	1	100	1	10	1000	0	1000
7	1	1	1	100	1	1	1000	0

(b) Schéma (matrice) de communication

FIGURE 3.15 – Données en entrée de l'algorithme TREEMATCH : une représentation de l'architecture et un schéma (matrice) de communication).

TREEMATCH suit une approche ascendante et commence par travailler au niveau des feuilles de l'arbre (soit la profondeur 3 dans notre exemple). À ce niveau, soit k l'arité du niveau supérieur. Dans notre cas, $k = 2$ et divise donc l'ordre $p = 8$ de la matrice m. L'algorithme continue donc en ligne 6 avec l'appel à la fonction `GroupProcesses`. Cette fonction génère tout d'abord la liste de tous les groupes possibles des entités virtuelles d'exécution. La taille de ces groupes est donnée par l'arité k du nœud de niveau supérieur dans l'arbre ($k = 2$ dans notre exemple). Par exemple, il est possible de regrouper l'unité virtuelle numéro 0 avec les unités virtuelles numérotées de 1 jusqu'à 7 et l'unité virtuelle numéro 1 avec les unités virtuelles numérotées de 2 à 7 et ainsi de suite. De façon générale, nous aurons $\binom{8}{2} = 28$ groupes possibles d'unités virtuelles. Comme nous en avons $p = 8$ et qu'elles sont groupées par paires ($k = 2$), il faut déterminer $\frac{p}{k} = 4$ groupes qui n'ont pas d'unités virtuelles communes. Pour trouver ces groupes, un graphe des incompatibilités entre les groupes (ligne 2 de la fonction `GroupProcesses`) est établi.

Function GroupProcesses(T,m,depth)

```
   Input: T //Arbre de la topologie
   Input: m // Matrice de communication
   Input: depth // Niveau courant
1  l ←ListOfAllPossibleGroups(T,m,depth)
2  G ←GraphOfIncompatibility(l)
3  return IndependentSet(G)
```

Function AggregateComMatrix(m,g)

```
   Input: m // Matrice de communication
   Input: g // Liste des groupes d'unités
          (Méta-Unités)
1  n ← NbGroups(g)
2  for i ← 0..(n − 1) do
3      for j ← 0..(n − 1) do
4          if i = j then
5              r[i, j] ← 0
6          else
7              r[i, j] ← Σ_{i1∈g[i]} Σ_{j1∈g[j]} m[i1, j1]
8  return r
```

Deux groupes sont dits *incompatibles* s'ils partagent une unité virtuelle d'exécution (un processus). Par exemple, le groupe {2,5} est incompatible avec le groupe {5,7} puisque le processus numéro 5 est membre des deux groupes. En fait, cela signifie que le processus numéro 5 ne peut pas être assigné à deux unités physiques d'exécution simultanément. Dans ce graphe des incompatibilités, les sommets correspondent aux groupes et les arêtes représentent les incompatibilités entre groupes. Dans la littérature, un tel graphe est le complémentaire du graphe de Kneser [111] et se note *Complémentaire de* $KG_{n,k}$ (cf. figure 3.16). L'ensemble des groupes recherchés est dans un ensemble indépendant de ce graphe, c'est-à-dire un ensemble de groupes qui n'ont aucune arête incidente entre eux. Cette condition est indispensable pour ne pas se retrouver dans une situation d'*oversubscribing* (i.e plusieurs processus par unité physique d'exécution). Une particularité intéressante du complément du graphe de Kneser est que si k divise p, alors les ensembles indépendants maximaux sont de taille $\frac{p}{k}$. Ainsi, n'importe quel algorithme glouton trouvera toujours un ensemble indépendant de la taille souhaitée.

Cependant, tous ces ensembles indépendants ne sont pas de qualité équivalente et dépendent des valeurs contenues dans la matrice. Dans notre exemple, un groupement de l'unité virtuelle 0 avec l'unité virtuelle 5 n'est pas pertinent puisqu'elles n'échangent qu'un faible volume de données. La conséquence de ce groupement sera de reporter un volume important de communications sur le niveau supérieur de l'arborescence, niveau considéré plus coûteux du point de vue des communications. Afin de prendre ce phénomène en considération, le graphe des incompatibilités est pondéré par le volume total de communication du groupe moins son volume de communications internes. Par exemple, en se basant sur la matrice m, la somme des communications du processus 0 est de 1114 et celle du processus 1 est de 2104 pour un total de 3218. Si ces deux entités devaient être regroupées, ce coût de communications sera réduit de 2000, ce qui correspond au volume de données échangées entre les processus 0 et 1, soit un total de $3218 - 2000 = 1218$. L'intérêt d'un groupe est inversement proportionnel à ce coût. Cependant, trouver un tel ensemble indépendant de coût minimal est un problème NP-

Difficile [103]. Par conséquent, des heuristiques sont employées afin de trouver un ensemble indépendant adéquat :

- **plus petite valeur prioritaire** : les sommets sont triés par ordre croissant de poids et un ensemble indépendant maximal est créé de façon gloutonne ;
- **plus petite valeur maximale** : les sommets sont triés par ordre croissant de poids et un ensemble indépendant maximal est déterminé de façon à ce que le sommet de poids maximal soit minimisé ;
- **plus grand poids des voisins prioritaire** :les sommets sont triés de façon décroissante en fonction du poids moyen des sommets voisins. Puis, un ensemble indépendant maximal est déterminé en commençant par les sommets pour lesquels cette valeur est la plus importante [103].

TREEMATCH commence par la première méthode qui est ensuite améliorée à l'aide des deux méthodes suivantes. Il est possible de définir un seuil pour le nombre de groupes au-delà duquel la technique du «plus grand poids des voisins prioritaire» ne sera pas exécutée. Dans notre exemple, indépendamment de l'heuristique utilisée, l'ensemble indépendant de poids minimal est $\{(0,1)\ ,\ (2,3)\ ,\ (4,5)\ ,\ (6,7)\}$. La figure 3.16 illustre cette sélection. La liste des groupes membres de l'ensemble indépendant est affectée au tableau $group[3]$, comme le montre la ligne numéro 6 de l'algorithme TREEMATCH. Cela signifie, par exemple, que les processus 0 et 1 seront affectés à des feuilles partageant le même nœud parent direct.

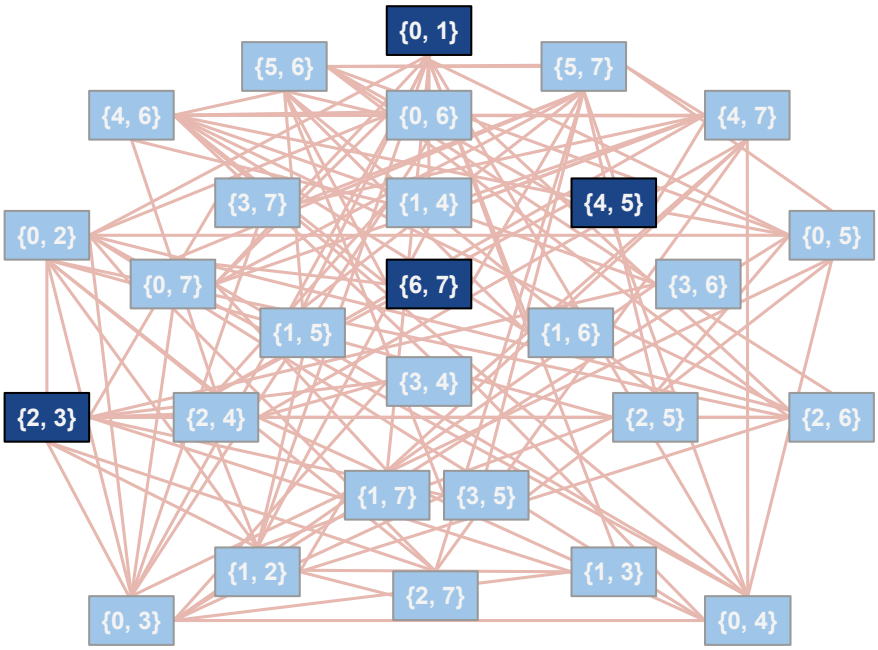


FIGURE 3.16 – Ensemble indépendant de poids minimal du graphe des incompatibilités : *Complémentaire de* $KG_{8,2}$.

L'étape suivante consiste à générer les groupes au niveau 2 de l'arborescence. Pour ce faire, il est nécessaire d'agréger la matrice m avec les communications restantes. La matrice agrégée est calculée dans la fonction `AggregateComMatrix` dont l'objectif est le calcul des communications restantes entre chaque groupe d'entités. Par exemple, entre le premier groupe $(0,1)$ et le deuxième groupe $(2,3)$, le volume de communication est de 1012 et se voit assigné à $r[0,1]$ (cf. la figure 3.17(a)). La matrice r de taille 4×4 (car il y a 4 groupes) est affectée à m (ligne 7 de l'algorithme TREEMATCH). La matrice m correspond dorénavant au schéma de communica-

Méta-Unité	0	1	2	3
0	0	1012	202	4
1	1012	0	4	202
2	202	4	0	1012
3	4	202	1012	0

(a) Matrice agrégée (profondeur 2).

Méta-Unité	0	1	2	3	4	5
0	0	1012	202	4	0	0
1	1012	0	4	202	0	0
2	202	4	0	1012	0	0
3	4	202	1012	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(b) Matrice étendue.

Méta-Unité	0	1
0	0	412
1	412	

(c) Matrice agrégée (profondeur 1).

FIGURE 3.17 – Évolution de la matrice de communication aux différentes étapes de l'algorithme TREEMATCH.

tion entre des *groupes* d'unités virtuelles d'exécution, que nous appellerons des *Méta-unités*. Les étapes suivantes de l'algorithme consistent donc à regrouper ces *Méta-unités* à chaque niveau de l'arbre jusqu'à atteindre la racine.

Function ExtendComMatrix(T,m,depth)

Input: T `//Arbre de la topologie`
Input: m `//Matrice de communication`
Input: depth `// current depth`
1 $p \leftarrow$ order of m
2 $k \leftarrow$arity(T,depth+1)
3 **return** `AddEmptyLinesAndCol`(m,k,p)

L'algorithme continue et décrémente la profondeur à 2. À ce stade, l'arité à la profondeur 1 est $k = 3$ et ne divise pas l'ordre $p = 4$ de la matrice m. Deux groupes artificiels ne communiquant avec aucun autre groupe sont ajoutés grâce à la fonction `ExtendCommMatrix`. Concrètement, deux lignes et deux colonnes remplies de 0 sont rajoutées à la matrice m, augmentant de ce fait son ordre p à 6. Cette étape illustre le cas où le nombre de ressources physiques disponibles est supérieur au nombre d'unités virtuelles d'exécution. Cette nouvelle matrice est présentée en figure 3.17(b). Une fois cette étape effectuée, il devient possible de regrouper les *Méta-unités* en trouvant un ensemble indépendant de poids minimal de taille $\frac{p}{k} = 3$. Les groupes créés sont alors les suivants : $\{(0, 1, 4)\ ,\ (2, 3, 5)\}$, ainsi que montré sur la figure 3.18.

Une fois encore, les communications restantes sont agrégées dans une matrice de taille 2×2 (cf. figure 3.17(c)). L'itération suivante, c'est-à-dire à la profondeur 1, n'offre qu'une unique possibilité pour regrouper les *Méta-unités* : $\{(0, 1)\}$, ensemble qui est affecté à $group[1]$. L'algorithme se poursuit ligne 8 et l'objectif est désormais d'affecter les processus aux cœurs physiques. Ainsi que nous l'avons vu, la première phase de TREEMATCH est ascendante pour grouper les unités virtuelles d'exécution. La seconde phase, en revanche, est descendante de la racine vers les feuilles en parcourant le tableau décrivant la hiérarchie des groupes d'unités virtuelles d'exécution. C'est cette hiérarchie qui va déterminer la fonction d'assignation (i.e. le placement ou la renumérotation). Par exemple, le processus 0 (resp. 1) appartenant à $group[1]$ est affecté à la partie gauche (resp. droite) de l'arbre de la topologie matérielle. Si un groupe

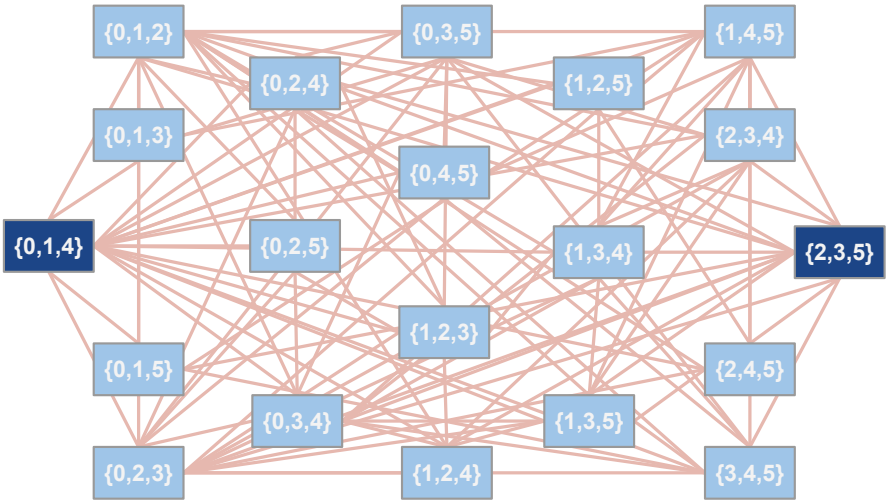


FIGURE 3.18 – Ensemble indépendant de poids minimal du graphe des incompatibilités entre *Méta-unités* (profondeur 2 de l'arbre de la topologie).

correspond à un groupe artificiel, aucune unité ne sera affectée au sous-arbre correspondant. À la fin, les processus numérotés de 0 à 7 sont affectés aux feuilles de l'arbre (i.e. aux unités physiques d'exécution) 0,2,4,6,1,3,5, et 7 respectivement (cf. l'arbre décrit par la figure 3.15(a)).

La politique de placement basée sur TREEMATCH est comparée aux politiques simples de placement *Round-Robin* et séquentielle (*Packed*), telles que définies en section 3.1.2.3 sur la figure 3.15(a). Dans notre exemple, le placement de TREEMATCH est meilleur que celui de *Round-Robin* et que le placement séquentiel : en effet, la politique séquentielle sépare les processus 4 et 5 des processus 6 et 7 alors que ces deux groupes échangent un volume important de données. Il est plus pertinent de les affecter au sein du même sous-arbre. De même, la solution proposée par *Round-Robin* n'est pas satisfaisante.

À noter que l'algorithme fonctionne si le nombre de ressources physiques de calcul est supérieur ou égal au nombre de processus. En revanche, la version de TREEMATCH présentée dans cette section ne gère pas les cas d'*oversubscribing*, c'est-à-dire si le nombre de processus est strictement supérieur au nombre de cœurs disponibles dans l'architecture considérée. Néanmoins, une version plus sophistiquée de TREEMATCH permet de gérer ce cas de figure. Des optimisations de TREEMATCH ainsi que des extensions pour les cas non gérés par la version détaillée ici sont présentées dans [RI3] et la thèse de François TESSIER [193].

3.3.4 Étape 4 : application effective du résultat de la permutation σ

3.3.4.1 Problèmes liés à l'application du placement

Nous avons déjà evoqué dans la section 3.2.1 que le résultat de la permutation [12] pouvait être interprété et appliqué selon deux modalités différentes. Dans la version initiale de ce travail, nous avions choisi une approche basée sur le placement, c'est-à-dire que le résultat fourni par Scotch (à l'époque) était appliqué lors du lancement de l'application parallèle. Deux méthodes sont possibles pour arriver à cela, la génération d'une ligne de commande et la génération d'un fichier appelé le *machinefile* .

12. J'utilise le terme permutation par abus de langage, car nous avons vu que des cas non-bijectifs sont également traités par TREEMATCH.

La génération d'une ligne de commande multi-exécutables `mpiexec` permet, de façon standard, de lancer plusieurs exécutables (potentiellement différents) simultanément. Une propriété très intéressante est que l'ensemble des processus lancés ainsi vont appartenir à la même application et donc la numérotation des processus est contrôlable au moment de leur lancement. Par exemple, supposons que l'on veuille lancer deux programmes `prog1` et `prog2` avec respectivement 2 et 3 processus chacun sur deux machines distinctes. Une commande possible sera alors :

```
mpiexec -np 2 -host toto ./prog1 : -np 3 -host tata./prog2
```

Dans ce cas, nous aurons les processus 0 et 1 (dans `MPI_COMM_WORLD`) qui exécuteront `prog1`, tandis que les processus 2, 3 et 4 exécuteront `prog2`, à supposer que `prog1` et `prog2` utilisent MPI. Si c'est le cas, alors ces deux programmes vont appeler `MPI_Init` et tous les processus vont pouvoir s'échanger des données, ainsi que le requiert le modèle de programmation de MPI. En lançant le même programme plusieurs fois, en jouant sur l'ordre de construction de cette ligne de commande et avec des noms de machines adéquats, il devient alors possible de contrôler exactement le placement des processus de l'application que l'on cherche à déployer.

La génération d'un *machinefile* le *machinefile* est un fichier qui liste les machines potentiellement utilisées par une application parallèle MPI. Il s'agit là d'un paramètre *optionnel* de la commande `mpiexec`, et qui en règle générale assigne les rangs des processus (dans `MPI_COMM_WORLD`, toujours) selon la position dans ce fichier. Par exemple, supposons que le *machinefile* contienne trois noms de machines : *Mann*, *Carpenter* et *Anderson*. Alors le lancement d'un programme avec :

```
mpiexec -np 3 -machinefile ./machinefile ./mon_prog
```

aura pour résultat de placer le processus 0 sur *Mann*, le 1 sur *Carpenter* et le 2 sur *Anderson*. Là encore, en organisant le *machinefile* de façon adéquate, il est possible d'appliquer une politique de placement, comme dans Brandfass *et al.* par exemple [26].

Dans les deux cas, il s'agit de fournir des paramètres bien choisis à la commande de lancement `mpiexec` ou `mpirun` et cela pose un certain nombre de problèmes. Pour la génération de ligne de commande, sa taille est limitée et donc le lancement d'une application possédant un grand nombre de processus ne sera pas toujours envisageable. Dans l'autre cas, l'interprétation du *machinefile* dépend de l'implémentation de MPI utilisée, voire de la version de l'implémentation de MPI. Il n'est donc pas exceptionnel d'avoir à réécrire ce fichier pour mettre en place la politique de placement désirée.

Enfin, comme expliqué en section 3.2.1, ce placement est décidé au moment du lancement de l'application et n'est plus modifiable par la suite, ce qui serait pourtant intéressant pour les applications dont le schéma de communication varie au cours de l'exécution (e.g. les applications de maillage adaptatif).

3.3.4.2 Choix de la fonction de renumérotation

Ces problèmes techniques et de portabilité de la méthode nous ont poussés à considérer une autre façon d'appliquer un placement de processus. Ainsi, l'examen du standard MPI montre qu'un certain nombre de fonctions de topologies virtuelles (censées décrire le schéma de communication de l'application, donc) étaient capables d'effectuer une opération de renumérotation, en créant un nouveau communicateur au sein duquel les processus membres voient leurs rangs potentiellement modifiés, selon des critères propres à l'implémentation de MPI. En particulier, il est tout à fait possible de renuméroter les processus sur la base de leur

affinité matérielle, de façon à mettre en correspondance la topologie virtuelle avec la topologie matérielle de l'architecture sous-jacente.

Cette approche permet de modifier la renumérotation dynamiquement en appelant plusieurs fois la fonction MPI choisie au cours de l'exécution de l'application, ce qui n'était pas permis avec l'approche basée sur le placement. En revanche, le code d'une application doit être modifié ce qui n'est pas nécessaire dans le cas du placement. Notre choix s'est porté sur la fonction `MPI_Dist_graph_create` dont le prototype est le suivant :

```
int  MPI_Dist_graph_create(MPI_Comm comm_old,
                           int n,
                           const int sources[],
                           const int degrees[],
                           const int destinations[],
                           const int weights[],
                           MPI_Info info,
                           int reorder,
                           MPI_Comm *comm_dist_graph);
```

Les paramètres `n`, `sources`, `degrees`, `destinations` et `weights` caractérisent la topologie virtuelle (qui peut donc correspondre au schéma de communication applicatif) tandis que `reorder` permet d'activer la renumérotation. `comm_dist_graph` est le nouveau communicateur dans lequel les rangs des processus membres sont renumérotés par l'implémentation MPI.

Les raisons de ce choix sont multiples : il s'agit de la fonction de topologie virtuelle la plus générique de MPI et de plus cette version est celle qui a été ajoutée au standard 2.2 en 2009/2010 et qui règle des problèmes de passage à l'échelle constatés dans l'interface précédente des topologies virtuelles génériques. Il serait de plus possible d'appliquer le même traitement aux autres fonctions de topologie virtuelle dans MPI, avec un véritable support de la renumérotation, ce qui est rarement fait en pratique [194].

3.3.4.3 Implémentation et emploi de `MPI_Dist_graph_create`

Modes de calcul de la renumérotation Nous avons donc implémenté notre propre version de la fonction `MPI_Dist_graph_create` permettant désormais l'utilisation du paramètre `reorder`. Cette version, disponible dans la version actuelle d'Open MPI [13], utilise effectivement le graphe de topologie virtuelle qui lui est passé en argument. Les informations de topologie matérielle sont quant à elles obtenues dynamiquement grâce à hwloc et servent à construire une structure de données utilisée par TREEMATCH. TREEMATCH va alors déterminer non pas un placement mais de nouveaux numéros de rangs pour les processus du communicateur d'origine (le premier paramètre de la fonction `MPI_Dist_graph_create`). Deux modes de renumérotation ont été implémentés dans notre version de `MPI_Dist_graph_create` :

— **le mode centralisé** : avec ce mode (celui par défaut), un seul processus-maître centralise les données des topologies virtuelle et matérielle avant de calculer avec TREEMATCH la renumérotation de l'ensemble des processus applicatifs. Ces nouveaux numéros de rang sont ensuite distribués par le processus-maître à tous les autres avec une opération collective (`MPI_Scatter`). Ensuite, un partitionnement est effectué grâce à `MPI_Comm_split` avec pour clef (`key`) ce nouveau numéro de rang et une couleur (`color`) identique pour tous, ce qui permet de créer un communicateur similaire à celui de dé-

13. Version 4.0.1 au moment de la rédaction de ce document.

part mais au sein duquel les processus voient leurs rangs modifiés [14]. Ce mode de fonctionnement prend en compte l'ensemble des informations ce qui permet d'obtenir une renumérotation pertinente quelle que soit le placement initial des processus. La seule contrainte étant que ces derniers doivent êtres liés (*bindés*) à des unités physiques d'exécution.

— **le mode partiellement distribué** : avec ce mode, on choisit autant de processus-maîtres que de nœuds de calcul dans la machine-cible. Chaque maître utilise les données locales à son nœud (i.e. les topologies virtuelle et matérielle) sans avoir besoin d'échanger avec d'autres processus, ce qui réduit les communications inter-nœuds qui étaient présentes dans le mode centralisé. Chaque maître utilise alors TREEMATCH pour calculer une renumérotation locale à son nœud. Ce nouveau numéro local est ensuite modifié pour en déduire une valeur globale utilisable au sein de la machine tout entière. Comme dans le cas centralisé, un nouveau communicateur avec renumérotation est créé grâce à un appel à `MPI_Comm_split` avec les paramètres de clef et de couleur adéquats. Le mode partiellement distribué permet une répartition du calcul de renumérotation sur plusieurs processus (TREEMATCH n'étant pas un algorithme distribué) ce qui accélère ce calcul mais au prix d'une permutation potentiellement moins intéressante puisque seules des informations locales sont utilisées. Nous sommes donc tributaires de la répartition initiale des processus sur les nœuds, ce qui n'était pas le cas dans le mode centralisé. Cependant, une répartition initiale cherchant à minimiser les communications inter-nœuds (e.g. Brandfass *et al.* [26]) serait idéalement complémentée par ce mode partiellement distribué.

Modification des applications MPI De nouvelles applications écrites avec MPI pourraient tirer parti de cette version améliorée de `MPI_Dist_graph_create` directement. Pour les applications MPI pré-existantes, des modifications sont en revanche nécessaires. L'ampleur de ces modifications peut rester limitée si une seule opération de renumérotation est exécutée au debut de l'application (par exemple). La figure 3.19 donne un exemple concret (il s'agit du début du code d'un noyau NAS modifié afin d'utiliser la renumérotation). Cet exemple montre comment remplacer le communicateur `MPI_COMM_WORLD` par un autre communicateur dans lequel les communications seront optimisées pour l'architecture sous-jacente. On remarquera l'utilisation d'une fonction `read_pattern` permettant de remplir les tableaux passés en arguments à `MPI_Dist_graph_create` et décrivant le comportement de l'application. Cette fonction utilise les données stockées dans un fichier produit par notre outil de traces, dont un extrait est donné en section 3.3.1.3. En particulier, il est possible d'indiquer à cette fonction quelle métrique utiliser (volume global des données ou bien nombre de messages). Nous rappelons toutefois que normalement, c'est à l'utilisateur qu'incombe la tâche de fournir le schéma de communication applicatif.

3.3.4.4 Un exemple de résultat expérimental

Nous allons maintenant détailler des résultats obtenus avec une application MPI modifiée afin de lui faire utiliser de la renumérotation de processus. Des expériences et résultats complémentaires sont disponibles dans [CI6] et [RI3]. La plate-forme matérielle utilisée est identique à celle décrite en section 3.1.2.1 : une grappe de 32 nœuds interconnectés par un réseau Infiniband (HCA : Mellanox Technologies MT26428, ConnectX IB QDR). Chaque nœud possède deux processeurs INTEL XEON NEHALEM X5550 quadricœurs à 2,66 GHz. Au niveau de la

14. `MPI_SIMILAR` sera retourné par la fonction `MPI_Comm_compare` si l'on devait comparer ces communicateurs.

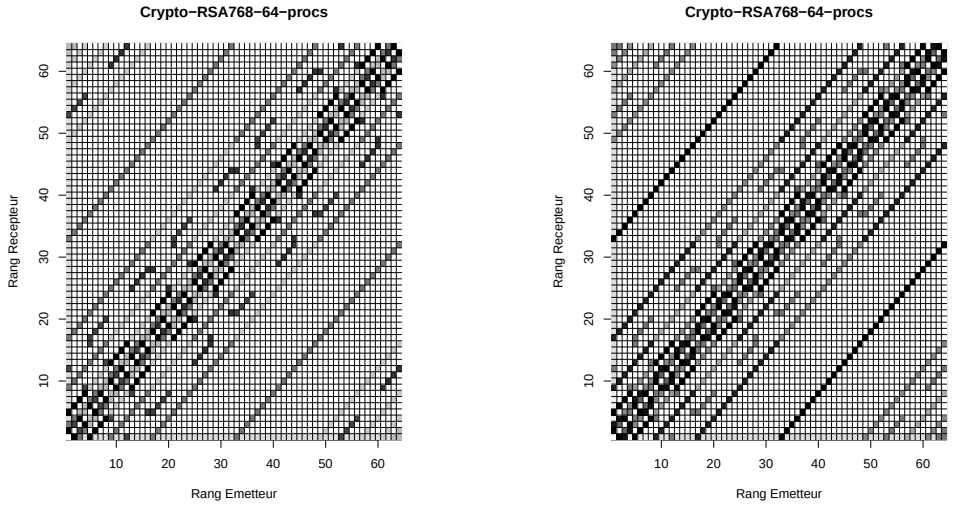
```
MPI_Init(NULL,NULL);
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );

if(argc > 1){
    if (rank == 0) {
    int *sources = NULL;
    int *degrees = NULL;
    int *destinations = NULL;
    int *weights  = NULL;
    int reorder = 1;

    pattern_file = argv[1];
    metric = atoi(argv[2]);
    /* read_pattern allocates the arrays */
    read_pattern(pattern_file,size,&destinations,&weights, &sources,&degrees,metric);

    MPI_Dist_graph_create(MPI_COMM_WORLD,comm_size, sources, degrees, destinations,
                          weights, MPI_INFO_NULL, reorder, &comm_topo);
    }
    else
      MPI_Dist_graph_create(MPI_COMM_WORLD, 0, NULL,NULL,NULL,NULL,
                            MPI_INFO_NULL, reorder, &comm_topo);

    if (comm_topo != MPI_COMM_NULL) {
      comm_to_use = comm_topo;

      /* Get the new rank in the new communicator */
      MPI_Comm_rank( comm_to_use, &rank);
    }
    else
      comm_to_use = MPI_COMM_WORLD;
}
else
  comm_to_use = MPI_COMM_WORLD;

/* Use comm_to_use in the rest of the code instead of MPI_COMM_WORLD  */
```

FIGURE 3.19 – Intégration de `MPI_Dist_graph_create` dans un code MPI pré-existant.

mémoire, chaque cœur possède ses caches L1 et L2 tandis que le cache L3 est partagé par les quatre cœurs d'un processeur. Un nœud possède 24 Go de mémoire DDR3 (1,33 Ghz).

Le nombre RSA-768 à 232 chiffres et 768 bits a été factorisé en Décembre 2009 ([109] et [110]). Ce travail a mobilisé une grande puissance de calcul et mis à contribution plusieurs algorithmes et applications. Une étape en particulier est chargée de trouver des dépendances entre les lignes d'une matrice creuse en utilisant l'algorithme de Block Wiedemann. Nous avons mené des expériences sur une version simplifiée de cet algorithme qui présente une caractéristique intéressante : les communications sont l'élément dominant de cette phase. De plus, dans la version que nous avons utilisée, la répartition entre calculs et communications est tracée durant l'exécution. De façon habituelle, les communications occupent plus de la moitié du temps total d'exécution. La perspective d'une réduction du temps passé dans les communications est donc très intéressante. L'application codant cet algorithme n'utilise pas normalement la renumérotation et nous l'avons donc modifiée selon les principes détaillés en section 3.3.4.3. Cette application a été exécutée avec 64 processus et pour 100 itérations. Les politiques de placement *Round-Robin* et *séquentielle* sont comparées avec une renumérotation basée sur les métriques *Volume global de donnée* et *Nombre de messages*. Les schémas de communication de l'application pour ces métriques sont montrés par la figure 3.20. On remarquera que la métrique *Nombre de messages* donne un schéma légèrement plus précis que l'autre mé-

trique. Dans le cas de la renumérotation, le placement initial des processus a été fait selon la politique *Round-Robin* [15].



(a) Volume total de données (b) Nombre de messages

FIGURE 3.20 – Schémas de communication pour RSA 768.

Les résultats obtenus sont résumés par la table 3.1 : pour chaque politique de placement ou de renumérotation considérée, le temps total d'exécution, la répartition entre communications et calculs et le ratio de communications intra-nœuds sont indiqués. La politique *Round-Robin* est celle pour laquelle les résultats sont les plus mauvais. Cela peut s'expliquer par le fait que quasiment toutes les communications sont inter-nœuds, moins rapides que les communications intra-nœuds. La politique séquentielle est plus efficace pour cette application avec un taux de communications intra-nœuds assez important, ce qui permet de diminuer significativement le temps passé dans les communications et *in fine* dans l'application. Il convient de noter que cette politique *séquentielle* est celle utilisée normalement par l'application est que cette dernière a été fortement optimisée pour ce cas de figure. Quant à la renumérotation, on peut constater que la métrique de volume des données améliore les résultats de *Round-Robin* mais pas ceux de la politique séquentielle.

Placement	Temps total d'exécution	Répartition Communications/Calcul	Ratio de communications intra-nœuds
Round-Robin	372s	Comm 66% Calcul 34%	*négligeable*
Séquentiel	221s	Comm 39% Calcul 61%	62%
Renumérotation (volume de données)	277s	Comm 52% Calcul 48%	21%
Renumérotation (nombre de messages)	**216 s**	Comm **38%** Calcul **62%**	**63%**

TABLE 3.1 – RSA-768 : résultats pour cent itérations

Le taux de communications intra-nœuds ne permet pas une diminution suffisante du temps de communication (par rapport au temps de calcul). Les résultats les plus intéressants sont

15. Ce qui n'a en pratique aucune conséquence sur le résultat.

ceux obtenus par la renumérotation basée sur le nombre de messages qui arrive à améliorer les performances obtenues par le placement séquentiel. Le résultat peut paraître modeste mais il faut rappeler que le placement d'origine des processus est *séquentiel* (et que l'application est optimisée pour ce cas) et que tout est fait de façon *automatique sans que l'utilisateur n'ait à se soucier des détails du matériel*. La seule chose qu'il se doit de fournir est le schéma applicatif mais nous avons déjà vu que les traces obtenues suite à une exécution préliminaire étaient déjà suffisantes pour gagner en performances. De plus, cette application a été conçue nativement pour exploiter très finement l'architecture-cible, il est donc difficile d'arriver à en améliorer les performances et pourtant, c'est ce que nous sommes parvenus à faire et de façon automatique et transparente pour l'utilisateur.

3.4 Au-delà du déploiement : localité et allocation de ressources de calcul

La section 3.3 a détaillé un ensemble de travaux dont l'objectif est l'amélioration des communications dans les applications parallèles par une prise en compte effective de la localité. Cependant, si les performances d'une application sont conditionnées par son déploiement et son affectation sur les diverses ressources de calcul, elles le sont également par l'attribution de ces mêmes ressources. En effet, sur certains systèmes, l'affectation d'un processus sur *un* nœud de calcul mal placé par rapport à l'ensemble est susceptible d'impacter très négativement les performances globales des applications [112](de l'ordre de 30% !). Partant de ce constat et étant donnés les résultats positifs obtenus avec TREEMATCH dans les domaines du placement/renumérotation de processus et de l'équilibrage de charge prenant en compte les topologies matérielles, nous avons décidé d'étudier l'impact sur les performances applicatives d'une allocation de ressources prenant en compte non seulement les aspects matériels mais également le comportement de l'application à déployer et exécuter sur la partition allouée. Dans la dernière section de ce chapitre, nous allons donc examiner les problèmes posés par l'allocation de ressources, avant de décrire la solution proposée pour améliorer les performances.

3.4.1 Allocation de ressources dans les machines parallèles

Les machines parallèles de grande taille sont construites pour répondre à des besoins – en termes de puissance de calcul – exprimés par des applications scientifiques toujours plus gourmandes. Il est cependant rare qu'une seule application occupe l'intégralité de la plate-forme. La conséquence est donc que dans un but d'exploitation optimale de cette dernière, il est impératif qu'un partage des ressources de calcul soit instauré. Ce partage est effectué temporellement (une plate-forme n'est en règle générale pas dédiée à une unique application) mais également spatialement : de multiples utilisateurs vont donc être amenés à devoir cohabiter et partager les ressources de calcul, c'est-à-dire de façon explicite les unités de calcul disponibles (i.e. les nœuds de calcul, les cœurs, etc.) mais aussi et de façon plus implicite les ressources réseau comme par exemple les *switches*. Une plate-forme peut ainsi être partagée par un grand nombre d'utiliseurs simultanément, ce qui exclut d'emblée un accès interactif. Afin de contrôler l'accès à la plate-forme et d'assurer une répartition équitable des ressources, les utilisateurs soumettent donc leurs requêtes (en termes de ressources et calcul et de temps d'utilisation) à un système appelé le *gestionnaire de ressources* (ou *Resource and Job Manager System*) qui remplit trois fonctions :

— la centralisation de toutes les requêtes provenant des utilisateurs de la plate-forme ;
— l'allocation des ressources les plus pertinentes vis-à-vis de la requête formulée ;

— le déploiement et l'exécution de l'application (ou *job*) de l'utilisateur sur les ressources sélectionnées et allouées.

3.4.2 Choix de métrique et approche retenue.

Des multiples critères doivent être optimisés par les gestionnaires de ressources et ces critères sont amenés parfois à rentrer en conflit les uns avec les autres. Il faut donc effectuer des arbitrages selon des métriques, par exemple le nombre de *jobs* exécutés sur une certaine unité de temps ou encore le ratio de l'utilisation des ressources de la plate-forme. Cependant, l'optimisation de ces métriques dépend grandement du point de vue adopté : les administrateurs d'une machine parallèle possèdent en général une vue d'ensemble du système qu'ils gèrent, ce qui est rarement le cas des utilisateurs, d'où les différences. La métrique que nous pensons la plus pertinente pour un utilisateur est la durée de présence du *job* dans le système, c'est-à-dire le temps passé entre la soumission de ce dernier et la fin de son exécution. En effet, un utilisateur souhaite en règle générale obtenir les résultats de son application le plus tôt possible. Si nous supposons qu'une application est déjà très optimisée pour la machine sur laquelle elle va être déployée, par exemple en utilisant une implémentation de MPI offrant des communications très efficaces ou encore en exploitant la localité avec les méthodes et outils détaillés dans les sections précédentes (comme le placement ou la rénumérotation de processus), elle reste tributaire de l'allocation des ressources proposée par le RJMS. En particulier, aucune garantie n'est donnée que cette allocation sera compatible avec le schéma de communication de l'application à déployer. Ainsi, la prise en compte de ce schéma, non seulement pour le déploiement mais également pour l'allocation des ressources va permettre une réduction encore plus importante du temps de communication de l'application. Ainsi nous avons intégré notre algorithme de calcul de placement/renumérotation TREEMATCH dans un gestionnaire de ressources très répandu, SLURM [212] [16]. L'objectif est d'utiliser TREEMATCH pour déterminer une allocation de ressources pertinente, en se basant non seulement sur les informations de topologie matérielle, mais également sur les informations de topologie virtuelle (i.e. le schéma de communication applicatif). Là encore, notre approche est *qualitative*, c'est-à-dire que c'est la structure du matériel qui est exploitée plus que ses caractéristiques numériques comme la latence ou le débit. De plus, ces informations nous sont fournies directement et automatiquement par hwloc sans que l'utilisateur n'ait à le faire.

3.4.3 Topologies virtuelles et matérielles dans les gestionnaires de ressources

L'utilisation des informations de topologie virtuelle et matérielle pour la réservation et l'allocation des ressources n'est pas nouvelle est à déjà été proposée notamment dans le cas des grilles de calcul. Santos *et al.* [176] modélise le problème dans ce contexte et propose une solution prenant en compte les caractéristiques des réseaux d'interconnexion ainsi que les besoins des applications. Koala [184] et Liu *et al.* [125] travaillent également dans ce contexte des grilles afin de sélectionner la ressource adéquate, en l'occurence la grappe présentant les meilleures caractéristiques pour réduire l'impact des communications à grande distance (i.e. utilisant le réseau *WAN*). Cependant, ces approches considèrent les grappes dans leur ensemble et ne vont pas explorer les niveaux internes de hiérarchie ni prendre en compte les effets NUMA.

Plus récemment, certains travaux visent des classes d'applications particulières, par exemple celles basées sur *MapReduce*. C'est le cas de TARA [118] qui utilise une description de l'application pour allouer ses ressources. Cependant, cette approche manque de généricité et ne prend pas en compte les aspects matériels.

16. Travail effectué par Adèle VILLIERMET durant sa thèse.

Nous avons vu précédemment que la mise en correspondance de la topologie virtuelle avec la topologie matérielle était susceptible d'améliorer les performances applicative (cf. Bathelé *et al.* [15] et la section 3.1.5.2 plus généralement). Les caractéristiques du réseau peuvent ainsi être prises en compte par le RJMS pour trouver une allocation pertinente en sélectionnant des nœuds appartenant à un même *switch* (ou au même niveau dans le réseau) pour éviter là encore des communications trop coûteuses (cf. Navaridas *et al.* [149]).

HTCondor (connu autrefois sous le nom de Condor) utilise un système de *matchmaking* [171] lui permettant d'apparier les besoins applicatifs aux ressources disponibles. Cependant, ce comportement applicatif en tant que tel n'est pas utilisé pour ce *matchmaking* et qui de toute façon cible les grappes et réseaux de stations de travail sans regarder la structure interne des nœuds.

La plupart des RJMS actuels possèdent des options ou des modules permettant une prise en compte des caractéristiques du matériel de façon à améliorer les communications dans les applications. Par exemple, SLURM (évoqué dans la section précédente), propose de minimiser le nombre de *switches* réseau utilisés dans l'allocation. Ainsi la localité des communications est mieux respectée puisque les *switches* situés dans des niveaux plus profonds de la hiérarchie sont moins «coûteux» que ceux situés à des niveaux plus hauts. Le même genre d'allocation *topo-aware* est proposé par Altaïr [157] (ex-PBS Pro), Grid Engine[153] et LSF [29]. Fujitsu propose également quelque chose, mais uniquement pour son réseau d'interconnexion propriétaire Tofu [62]. La différence entre SLURM et les solutions citées ci-dessus est que la sélection proposée est de type *best-fit* tandis que les autres sont de type *first-fit*.

D'autres RJMS proposent des options permettant un placement sophistiqué des processus applicatifs. Torque [42] propose un placement à l'intérieur des nœuds NUMA. OAR [30] utilise quant à lui une représentation flexible des ressources qui permet de placer les processus applicatifs sur la hiérarchie interne des nœuds de calcul. Cependant, dans les deux cas, il s'agit plus de placement [17] que d'allocation et pour cette dernière seule la topologie du réseau d'interconnexion est véritablement prise en compte et les caractéristiques internes des nœuds ne sont pas exploitées.

Cependant, comme le montre Pascual *et al.* [156], une allocation essayant de respecter la contigüité des ressources (pour une meilleure localité des communications) se traduit par une fragmentation plus importante dans le système, ce qui d'un point de vue des administrateurs n'est pas vraiment souhaitable et peut se révéler négatif pour les autres utilisateurs. Deveci *et al.* [51] se penche d'ailleurs sur le placement de processus dans le cadre d'allocations de ressources non contigües. Vydyanathan *et al.* [205] propose quant à lui un algorithme de *backfilling* prenant en compte la localité dans les applications structurées en graphes de tâches dont la pondération des arêtes correspond à des communications. Lucarelli *et al.* [128] étudie également l'impact de la localité et de la contiguité des ressources sur le *backfilling* dans l'allocation.

Albing *et al.* [5] étudie les topologies toroïdales et montre comment les nœuds de calcul pourraient être mis en correspondance avec une liste undimensionnelle afin de conserver des informations de localité. Ce travail utilise notamment des courbes de remplissage d'espace, comme celles de Hilbert. Cet article décrit également les stratégies d'allocation implémentées dans le système propriétaire de Cray, Application Level Placement Scheduler (ALPS) [44]. Des stratégies similaires sont implémentées dans SLURM qui supporte ALPS. Li *et al.* ([123] et[122]) étudient les allocations pour les machines de de type Blue Gene d'IBM (avec une topologie d'interconnexion toroïdale, donc).

Enfin, Yang *et al.* [211] se place dans un contexte de topologie en tore avec une stratégie d'ordonnancement de *jobs* basée sur la localité et qui s'efforce d'optimiser non seulement les performances de l'application mais également du système tout entier. Là encore, ce travail

17. Rappelons que les RJMS sont parfois également responsables du déploiement des applications ...

essaye de préserver de la contigüité entre les nœuds de calcul.

Tous ces travaux exploitent des propriétés des topologies matérielles ou s'efforcent de fournir des allocations qui respectent une certaine localité pour optimiser les communications de l'application. Cependant, le schéma de communication applicatif n'est pas pris en compte (ou alors de façon très exceptionnelle).

3.4.4 Allocation de ressources *virtual topology-aware*

Notre approche consiste donc à prendre en compte pour l'allocation des ressources non seulement les informations relatives au matériel sous-jacent (da façon qualitative), mais également et surtout le comportement de l'application pour essayer de maintenir au maximum une bonne localité. En pratique nous avons mis au point une nouvelle option de sélection dans le *plugin* `cons_res` de SLURM où l'algorithme de *best-fit* habituel est remplacé par notre algorithme TREEMATCH. En réalité, nous avons besoin de trois informations : outre le schéma de communication (qui sera donné sous la forme d'une matrice d'affinité, cf. exemple plus loin) et la topologie matérielle, nous devons également indiquer quelles sont les ressources déjà utilisées et qui ne peuvent plus être assignées pour la requête en cours.

La matrice d'affinité est donnée au moment de la soumission du *job* via une option de distribution de la commande `srun` :

```
srun -m TREEMATCH=/comm/matrix/path cmd
```

Le démon contrôleur de SLURM enregistre alors ce chemin dans la structure de données interne de SLURM décrivant le *job*, ce qui rend alors la matrice accessible à TREEMATCH pour calculer son allocation. Les informations de topologie matérielle sont fournies au démon contrôleur via un nouveau paramètre dans son fichier de configuration qui est le chemin sur la description de la topologie matérielle (`TreematchTopologyFile=/topology/file/path`). Cependant cette topologie décrit en fait l'ensemble de la machine-cible et il est donc indispensable de fournir une liste de *contraintes* à TREEMATCH, c'est-à-dire l'ensemble des ressources déjà occupées et non-affectables pour la requête en cours. Dans la version de TREEMATCH décrite en section 3.3.3, nous avons montré la version fonctionnant quand toutes les ressources sont disponibles. La version avec *contraintes* est décrite dans la thèse de François TESSIER [193] et permet une utilisation de TREEMATCH dans le cas où des ressources (par exemple des cœurs de calcul) sont indisponibles. Ces contraintes nous sont fournies par SLURM qui possède la vision globale du système et utilise des structures de type *bitmap* pour décrire l'occupation de la machine qu'il gère.

Cependant, il est nécessaire de traduire la description de la topologie faite par SLURM dans un format compréhensible par TREEMATCH. En effet, TREEMATCH utilise sa propre représentation : les cœurs sont les entités matérielles sélectionnables et un numéro global au sein du système leur est assigné. Pour faire en sorte que SLURM adopte un comportement similaire dans le cas d'une utilisation de TREEMATCH pour la sélection des ressources, nous devons utiliser le *plugin* `cons_res` avec le bon niveau de granularité, c'est-à-dire `SelectTypeParameters=CR_CPU` ou bien `CR_Core`. Dans ce cas, le *bitmap* de SLURM correspond à des cœurs et cette structure décrit précisément les entités matérielles déjà utilisées par d'autres applications. Cependant, les numéros des cœurs sont toujours relatifs aux nœuds de calcul et donc locaux et une traduction est nécessaire (selon une formule du type de celle donnée en section 3.1.2.3 par exemple). Une fois que TREEMATCH a déterminé les cœurs à allouer, une traduction vers le *bitmap* de SLURM est alors effectuée.

L'utilisation de TREEMATCH introduit un surcoût lors du calcul de la partition à allouer à l'application et ce surcoût croît avec la taille de la plate-forme cible. Afin de réduire ce surcoût,

nous avons implémenté une solution alternative qui cherche d'abord un sous-arbre convenable dans la topologie matérielle globale. C'est ce sous-arbre qui va ensuite être utilisé pour calculer l'allocation. Plus d'informations sur cette méthode et ses performances sont disponibles dans [CI3] et [RI1].

3.4.5 Un exemple pratique

Nous allons montrer sur un exemple relativement simple le bénéfice apporté par une prise en compte du comportement de l'application pour la réservation de ses ressources. Nous faisons l'hypothèse que ce comportement (schéma de communication par exemple) est connu au moment de la soumission du *job* dans le système. Encore une fois, soit l'utilisateur le connait et le fournit ou bien, il est récupéré via un profilage de l'application avec des outils et techniques tels que décrits en section 3.3.1. Nous supposons enfin que ce schéma de communication ne varie pas entre l'exécution préliminaire (pour déterminer le schéma, justement) et les exécutions suivantes, ce qui est le cas pour un large panel d'application parallèles (algèbre linéraire dense, stencils, etc.).

Dans notre example, nous allons comparer trois approches :

— l'utilisation de SLURM «classique», qui est toutefois capable de prendre en compte une partie de la topologie matérielle pour l'allocation de ressources ;
— l'allocation des ressources avec SLURM puis un déploiement de l'application utilisant TREEMATCH. Pour rappel, c'est SLURM qui déploie les application à l'aide de sa commande `srun` en lieu et place des habituels `mpirun/mpiexec` ;
— l'allocation des ressources avec SLURM, mais en utilisant TREEMATCH (et le schéma de communication applicatif donc) pour la sélection de ces dernières. Ensuite, c'est toujours SLURM qui procède au déploiement des processus applicatifs.

Les différences entre ces trois approches sont exposées par la figure 3.21. Dans ce cas de figure, nous avons une machine composée de 6 nœuds possédant 2 cœurs chacun mais le nœud numéro 3 (cœurs 6 et 7) est indisponible, utilisé par une autre application par exemple. Une nouvelle application est soumise et fait une requête pour réserver quatre nœuds (8 processus au total, groupés en paires).

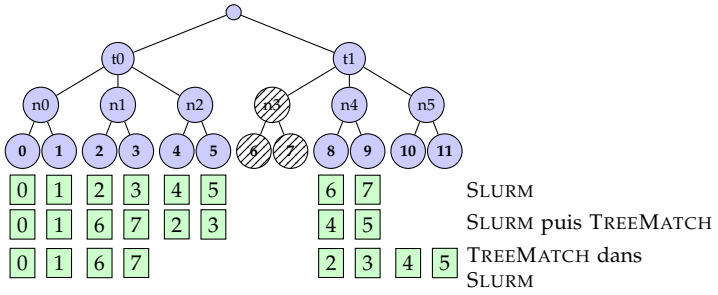


FIGURE 3.21 – Arbre de la topologie de 6 nœuds (2 unités physiques d'exécution chacun) avec le nœud n3 non disponible

La matrice d'affinité de cette application est donnée par le tableau 3.2.

Si SLURM doit allouer des ressources pour ces 8 processus, il va alors chercher le plus petit nombre de *switches* auxquels sont reliés des nœuds permettant à la requête d'être comblée : c'est le fonctionnement *topo-aware* de SLURM. Dans ce cas, l'ensemble de l'arbre va devoir être utilisé. Ensuite, les processus sont assignés à des cœurs de façon séquentielle (au sens donné en section 3.1.2.3). Les nœuds numéro 0, 1, 2 et 4 seront donc alloués pour ce *job* et les processus placés en conséquence par `srun`. Il est évident qu'une telle allocation est coûteuse du point de

Proc.	0 -1	2 - 3	4-5	6-7
0-1	0	**20**	0	**2000**
2-3	**20**	0	**1000**	0
4-5	0	**1000**	0	**10**
6-7	**2000**	0	**10**	0

TABLE 3.2 – Matrice d'affinité des 8 processus (4 paires) : les nombres correspondent à la quantité de données ou au nombre de messages échangés entre les processus applicatifs.

vue des communications car des paires de processus sont réparties sur les différents nœuds sans qu'aucune optimisation ne soit mise en place.

Il est alors possible d'utiliser TREEMATCH pour remédier à cette situation en se basant sur la matrice d'affinité et en plaçant les processus de façon plus sophistiquée. Ainsi, la paire 0-1 sera placée sur le nœud numéro 0, la paire 6-7 sur le nœud numéro 1, la paire 2-3 sur le nœud 2 et la paire 4-5 sur le nœud 4, ce qui est la meilleure répartition compte tenu de l'allocation disponible. Cependant, la paire 2-3 communique beaucoup avec la paire 4-5 et avec cette allocation, le traffic va passer par la racine de l'arbre, ce qui pour une métrique de type *Hop-Byte* n'est guère intéressant.

En revanche, si TREEMATCH se charge de calculer l'allocation en se basant sur la matrice d'affinité, le résultat sera amélioré du point de vue des communications : la paire 0-1 sera placée sur le nœud 0, la paire 6-7 sur le nœud 1, la paire 2-3 sur le nœud 4 et enfin la paire 4-5 sur le nœud numéro 5 car le nœud numéro 3 est indisponible. Toutes les communications entre les paires 2-3 et 4-5 seront plus locales (deux sauts au lieu de quatre) et leur coût réduit. Cependant, ainsi qu'énoncé précédemment, la solution proposée entraîne une possible augmentation de la fragmentation du système global. Plus de détails sur notre approche utilisant TREEMATCH ainsi que des résultats sur des historiques d'exécution d'applications parallèles sur de grands système (e.g. Curie) sont disponibles dans [AI1], [CI3] et [RI1].

3.5 Conclusion

Dans ce chapitre, nous avons expliqué pourquoi la problèmatique de la localité avait revêtu depuis toujours une importance particulière dans le monde du HPC. Cette importance s'est accrue avec l'émergence de nouvelles classes d'architectures. Nous avons ensuite montré comment nos travaux avaient permis de remettre cette problématique au goût du jour ainsi que les réponses que nous y avons apportées. Il est important de noter que ces travaux ont inspiré ou permis l'émergence d'autres travaux dans l'équipe et que cette thématique est devenu un axe de recherche privilégié de l'EPC TADaaM qui possède de plus une visiblité forte au niveau international concernant ces aspects. Ainsi nous sommes devenus des acteurs de référence pour tout ce qui concerne les topologies matérielles c'est-à-dire les moyens de les découvrir et les exploiter tant au niveau des supports exécutifs qu'au niveau applicatif.

Bilan Le premier élément que nous souhaitons mettre en avant est la pertinence des travaux menés, aussi bien du vue thématique que du *timing*. En effet, je pense que nous avons correctement appréhendé l'importance du problème de la localité et que les solutions proposées sont tout-à-fait appropriées et comblent des manques pour certaines situations. C'est en particulier le cas de hwloc, qui s'est imposé comme un élément incontournable pour qui souhaite obtenir et exploiter effectivement des informations sur le matériel.

Ensuite, si les implémentation de MPI arrivent à optimiser les communications pour les grappes de machines hiérarchiques complexes multicœurs, ce n'est pas toujours suffisant car l'implémentation de MPI n'est pas systématiquement en capacité de prendre toutes les décisions pertinentes en lieu et place de l'utilisateur. Ce dernier peut espérer des gains supplémentaires, à la condition de fournir des informations supplémentaires à l'implémentation pour pouvoir agir effectivement.

Des solutions sont possibles, nous l'avons bien vu avec notre approche basée sur la renumérotation utilisant notre implémentation de la fonction `MPI_Dist_graph_create`. Cependant, si cette solution est intéressante, elle possède un défaut majeur : l'approche retenue n'est pas standard puisque la fonction que nous avons choisie pourrait très bien être implémentée différemment dans d'autres versions de MPI [18] Il faudrait donc trouver des moyens standards pour arriver à un tel résultat. Or, dans sa version actuelle, MPI ne permet pas de récupérer et d'exploiter explicitement des informations de topologie matérielle. Seules des extensions ou introduction de nouvelles fonctionnalités permettraient d'arriver à atteindre ce but. Ce sera d'ailleurs l'objet du chapitre suivant.

Discussion Néanmoins, certaines questions mériteraient d'être encore développées. En premier lieu, la génération du schéma de communication applicatif mériterait d'être revue afin d'éviter une exécution préliminaire complète. Une approche basée sur la simulation d'un squelette extrait de l'application nous paraît constituer une solution intéressante.

Également, nous aimerions explorer la voie de l'analyse statique qui permettrait la génération de l'information voulue dès la compilation. Cependant, il semblerait que la quantité d'informations pouvant être extraite soit assez limitée. Il faudrait donc être capable de quantifier le lien entre la précision de l'information fournie et les gains de performances obtenus. Ainsi, si une information peu précise est déjà capable d'induire des gains substantiels, cette approche serait valable. Une telle étude serait à entreprendre, et Bordage *et al.* [25] s'est déjà penché sur un problème un peu similaire, c'est-à-dire celui du lien entre métriques d'optimisation et gains de performances.

Il pourrait alors être intéressant d'établir une classification des applications et de créer ainsi une bibliothèque de schémas applicatifs génériques utilisables pour fournir des informations quand il n'est pas possible d'obtenir un schéma précis pour une application donnée. Un telle bibliothèque pourrait être utilisée aussi bien dans le cas de l'allocation de ressources que pour le déploiement d'applications ou la renumérotation de processus. Les travaux de Ma *et al.* [131] sur les distances et similarités de schémas applicatifs pourraient de ce point de vue être appliqués pour la constitution de cette bibliothèque.

Ensuite, malgré la mise en place d'un système de profilage utilisant des compteurs matériels, nous ne sommes pas encore parvenus à établir une corrélation précise entre les gains de performance obtenus grâce au placement ou à la renumérotation et l'utilisation de certaines ressources, les mémoires caches en particulier. Il demeure encore difficile de savoir dans quelle mesure cela participe à l'accélération des communications intra-nœuds.

Enfin il est important de préciser que les solutions et approches retenues ne sont pas confinées au paradigme du passage de message. Nous avons utilisé MPI car cela constitue un environnement d'exéprimentation, de validation et de diffusion très important pour nos travaux mais il est loin d'être le seul. Ainsi, comme nous l'avons évoqué à plusieurs reprises, le paradigme basé sur des threads serait tout indiqué. L'étude des applications hybrides serait un bon point de départ, d'autant plus que nous avons mis en place une collaboration avec le CERFACS pour la mise au point d'outils efficaces de placement d'applications hybrides MPI + OpenMP (cf. section 4.2.6).

18. Ce qui est le cas en pratique car le *reordering* n'est quasiment jamais mis en place dans les bibliothèques MPI.

Chapitre 4

De nouvelles abstractions pour exploiter la localité dans les applications

Dans le chapitre 2, nous avons exposé comment les implémentations de MPI arrivaient (plus ou moins) à prendre en compte les topologies matérielles et à utiliser leurs caractéristiques pour améliorer les performances des communications. Il s'agit pour l'essentiel d'optimiser l'implémentation elle-même de façon à ce qu'elle rende de meilleurs services (i.e. des communications plus rapides) aux applications qui l'utilisent. La prise en compte de la localité et de la topologie matérielle n'est qu'incidente et implicite par l'utilisation de la bibliothèque MPI optimisée à dessein. Cette approche n'est pas vraiment portable car un changement de version de MPI entraînera vraisemblablement une perte des bénéfices apportés. De plus, les implémentations ont certes atteint un haut degré de sophistication (cf. section 2.5), mais ne sont pas en capacité de prendre toutes les décisions (d'autant plus qu'elles ne possèdent pas l'intégralité des informations nécessaires pour cette prise de décision) car elle ne s'occupent que des communications dans la pile HPC.

Dans le chapitre 3, nous avons alors montré qu'il existe de nombreux travaux cherchant à pallier ce problème en travaillant *autour* des implémentations MPI, c'est-à-dire au niveau de l'écosystème HPC de façon plus globale. Cette approche permet de s'affranchir de certaines limitations de MPI et offre des résultats intéressants, mais la portabilité reste encore problématique car le plus souvent il s'agit de solutions particulières. Quant aux mécanismes proposés dans les implémentations de MPI (par exemple, notre implémentation de `MPI_Dist_graph_create`, cf. section 3.3.4), ils n'offrent là encore aucune garantie de portabilité. Une possibilité est néanmoins offerte par un logiciel tel qu'hwloc, qui propose une interface de consultation et de récupération des informations matérielles et que les applications pourraient utiliser. Le logiciel hwloc étant disponible dans de nombreux logiciels de la pile HPC (gestionnaires de ressources, gestionnaires de processus, implémentations de MPI, etc.), il pourrait être considéré comme un standard *de facto* [1]. Cependant, hwloc est un outil qui selon moi n'est pas destiné aux développeurs d'applications, ou alors de façon marginale, pour ceux désireux d'optimiser très finement leur code pour une architecture bien particulière. En ce sens, l'interface d'hwloc se destine plutôt aux concepteurs d'outils systèmes, de supports exécutifs ou autres bibliothèques de communication.

Il me paraît toutefois déraisonnable de continuer à travailler dans un écosystème où les applications continuent à faire l'économie de modifications pour prendre en compte la localité (par exemple en utilisant uniquement une allocation et un déploiement *topo-aware*). Pour autant, les développeurs d'applications devraient avoir accès à des mécanismes et abstractions leur permettant de travailler sur les aspects «localité» de leur code sans être forcés d'avoir une vision très (trop ?) détaillée de l'architecture qui, au final, pourrait se révéler inexploitable. Nous avons donc acquis la conviction qu'un travail au niveau du standard MPI lui-même – c'est-à-dire au niveau de la définition de l'interface – était nécessaire pour atteindre cet ob-

1. Ce qui est le cas de MPI par ailleurs.

jectif. Trouver alors le bon niveau d'abstraction est impératif : il s'agit d'une question d'équilibre entre être capable de fournir des informations exploitables pratiquement sans pour autant noyer l'utilisateur sous les détails. C'est en particulier vrai pour un standard comme MPI avec son fameux agnosticisme matériel. Ainsi, notre objectif est de de proposer des abstractions permettant de structurer le code applicatif pour une meilleure exploitation de la localité. Tout ce que nous proposons doit néanmoins rester compatible avec ce qui a été montré jusqu'à présent (i.e. utilisation de bibliothèques MPI optimisées, déploiement d'application selon une politique de placement intelligente, etc.).

Dans la section 4.1 nous allons d'abord montrer comment le standard MPI s'est un peu adapté (ou est en passe de s'adapter) pour offrir des moyens d'intégrer la localité dans les applications et éventuellement passer d'un modèle de programmation plat (cf. section 2.3.3) à quelque chose de plus hiérarchique et extensible. La section 4.2 détaillera une proposition de modification de MPI, en cours de discussion actuellement et qui permettrait d'atteindre certains des objectifs. Modifier un standard comme MPI n'est pas chose aisée et nous expliquerons en section 4.3 les démarches effectuées et notamment comment nous avons créé au sein du Forum MPI un groupe de travail dédié aux topologies matérielles. Nous évoquerons les discussions actuelles et les pistes envisagées. Enfin, la section 4.4 concluera ce chapitre et nous donnera l'occasion d'en tirer le bilan.

4.1 Mécanismes disponibles et évolutions de l'écosystème MPI

MPI a été créé au mitan des années 90 et n'a cessé d'évoluer depuis, en particulier depuis les dix dernières années avec l'arrivée des versions MPI 3.X et des discussions actuelles pour MPI 4.0. De plus, les travaux présentés dans ce document couvrent une large période de temps et faire une «photographie» de la situation n'est guère facile, d'autant plus que les discussions se poursuivent toujours à l'heure actuelle. Dans cette section, nous allons détailler l'existant (depuis les débuts de MPI et les ajouts successifs) pour voir ce qui est concrètement utilisable pour les applications cherchant une meilleure prise en compte de la localité et de la topologie matérielle.

4.1.1 Topologies virtuelles et renumérotation

4.1.1.1 Types de topologies virtuelles

Il existe dans MPI le concept de *topologie virtuelle*. Nous avons beaucoup utilisé ce terme jusqu'à présent comme un synonyme de *comportement applicatif* ou de *schéma de communication applicatif*. Une topologie virtuelle dans MPI décrit des relations entre des processus applicatifs, relations en termes d'échanges de données car MPI est avant toute chose une interface conçue pour gérer les communications. Ainsi que nous l'avons évoqué en section 2.4.5, deux types de topologies virtuelles sont à l'heure actuelle disponibles dans MPI : les topologies cartésiennes et les topologies génériques de graphe[2]. Les premières sont bien évidemment des cas particuliers des secondes et leur présence se justifie par le nombre important d'applications structurées ainsi. Il n'est pas déraisonnable de penser que d'autres topologies virtuelles pourraient être introduites dans le futur. Par exemple, les applications de type *stencil* utilisent une topologie géométrique mais qui n'est pas couverte par le type cartésien (les échanges sur

2. En réalité il existe *trois* types de topologies virtuelles : cartésiennes; graphes et graphes distribués. Cette dernière catégorie a été introduite dans MPI 2.2 pour palier les problèmes de passage à l'échelle de l'interface de graphe «classique».

les diagonales sont exclus) et qui est un cas particulier là encore d'une topologie générique de graphe.

4.1.1.2 Structuration des application à l'aide des topologies virtuelles

Ces topologies virtuelles peuvent aider à structurer une application, mais le lien avec la topologie matérielle n'est pas garanti, ainsi que l'indique le texte du standard [3] :

> *"A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI. The description of the virtual topology, on the other hand, depends only on the application, and is machine-independent."*

En pratique surgissent deux problèmes :

— Premièrement, peu de bibliothèques MPI proposent une implémentation des topologies virtuelles prenant en compte l'architecture matérielle de façon effective, ainsi que le constate Träff en 2002 [194]. Il n'existe à l'heure actuelle que peu de travaux se penchant sur la question : ceux pionniers (1997 !) d'Hatazaki [81] qui utilisent le *reordering* pour des architectures spécifiques, ceux de Träff en 2002 (avec une hiérarchie à deux niveaux) et ensuite en 2011 mes travaux [CI6] et ceux de Rashti *et al.* [172] qui visent les grappes de nœuds hiérarchiques multicœurs. Au regard des vingt-cinq années d'existence de MPI, cela est bien peu ;

— Deuxièmement, quand bien même la bibliothèque MPI proposerait une implémentation exploitant le matériel sous-jacent, cela ne serait pas standard : la citation ci-dessus est très claire à ce sujet. Cela pose alors un problème d'adoption par les applications.

Les fonctions de topologies virtuelles dans MPI produisent un *communicateur* auquel est attaché une information de topologie virtuelle (cartésienne ou graphe) dans lequel les rangs des *MPI Process* sont modifiés pour refléter le schéma de communication choisi. C'est en effectuant ce calcul de renumérotation que la topologie matérielle peut être prise en considération afin de mettre en correspondance ces topologies (virtuelle et matérielle). Les travaux basés sur la renumérotation procèdent ainsi. Il faudrait donc garantir que ce calcul de renumérotation soit effectué de façon standard ce qui est difficilement possible car cela imposerait l'implémentation du mécanisme, voire l'algorithme pour la mise en correspondance des topologies virtuelles et matérielles.

Le problème reste entier si l'on cherche à utiliser les fonctions bas-niveau de l'interface des topologies virtuelles dans MPI, à savoir `MPI_Cart_map` et `MPI_Graph_map` puisque selon le texte du standard :

> `MPI_Cart_map` *computes an "optimal" placement for the calling process on the physical machine. A possible implementation of this function is to always return the rank of the calling process, that is, not to perform any reordering."*

D'une part la renumérotation conserve son caractère optionnel, mais de plus elle repose entièrement sur l'implémentation [4]. L'utilisateur se retrouve une fois de plus sans réelle garantie quant au calcul de la rénumérotation.

Une solution serait alors de laisser l'utilisateur calculer une renumérotation selon ses propres critères (en particulier des critères matériels) et ensuite de procéder à la création de la topologie virtuelle MPI en utilisant en paramètre d'entrée le communicateur avec la rénumérotation.

3. Citation extraite du début du chapitre 7 consacré aux topologies virtuelles.
4. Les guillemets autour du mot *optimal* sont d'ailleurs fort éloquents.

Ceci est possible grâce à la fonction `MPI_Comm_split` qui permet un partitionnement et une renumérotation du communicateur d'entrée en utilisant ses paramètres de *couleur* et de *clef*. Il suffirait d'imposer la même couleur à tous les *MPI Process* (pas de partitionnement donc) et de mettre en tant que *clef* le rang prévu dans la renumérotation pour créer un communicateur adéquat. Ce communicateur va alors servir pour la création de la topologie virtuelle :

```
MPI_Comm incomm, reordercomm, topocomm;
int color = MY_COLOR;
int key = 0;

/* Calcul de la clef = renumérotation */
key = savant_calcul(infos_matérielles, schéma_de_comm());

/* création du communicateur avec renumérotation */
MPI_Comm_split(incomm,
               color,
               key,
               &reordercomm);

/* creation de la topologie virtuelle utilisant la renumérotation précédente */
MPI_Comm_cart_create(reordercomm,
                     ndims,    /* donné par schéma_de_comm() */
                     dims,     /* donné par schéma_de_comm() */
                     periods,  /* donné par schéma_de_comm() */
                     0,        /* pas de renumérotation nécessaire ici !*/
                     &topocomm);
```

Nous obtenons ainsi une méthode permettant à l'utilisateur de contrôler exactement sa renumérotation et de créer une topologie virtuelle (cartésienne dans l'exemple, mais la topologie de graphe fonctionnerait de même) sans dépendre de l'implémentation de la fonction de topologie dans MPI. Cependant, bien que séduisante, cette approche voit un écueil perdurer : l'absence de méthode portable pour récupérer les informations de topologie matérielle (d'où provient le paramètre `infos_matérielles` de la fonction `savant_calcul` ?). Utiliser hwloc ? Possible, certes, mais comme nous l'avons expliqué en introduction dans ce chapitre, pas complètement portable et *a priori* hors de portée pour le chimiste ou le physicien écrivant son application.

4.1.1.3 Localité via les communications collectives de voisinage

Nous avons déjà evoqué l'importance des opérations de communications collectives dans MPI. Ces opérations sont appelées «collectives» car les fonctions implémentant de telles opérations doivent être appelées par l'ensemble des processus MPI appartenant au communicateur pour lequel l'opération a lieu [5]. Dans le cas d'une application structurée avec une topologie virtuelle (surtout dans le cas de graphes/graphes distribués), décrivant le schéma de communication, ces opérations collectives ne sont pas les plus adaptées. Afin de permettre à l'implémentation MPI de mieux optimiser les échanges de messages en utilisant le schéma de communication (connu et décrit via la topologie virtuelle), de nouvelles fonctions ont fait leur apparition : les collectives de voisinage (ou *Neighborhood Collective Communications*). En particulier, Hoefler *et al.* [87] montre comment une implémentation de MPI peut calculer un ordonnancement optimisé des échanges de messages grâce à des opérations de ce type. Ces opérations prennent en argument un communicateur auquel une information de topologie est

5. Pour autant, cela ne signifie pas que tous les processus vont activement échanger des données lors de cette opération et cela n'induit aucune espèce de synchronisation entre les processus participants.

rattachée, qui va être effectivement exploitée. Ainsi, le modèle «plat» de MPI est légèrement remis en cause et une certaine forme de localité peut être exploitée. Les échanges entre voisins s'en trouvent facilités et directement (re)connus au niveau de l'implémentation.

4.1.2 Support du partage de l'espace d'adressage

Le passage de messages n'a pas toujours été perçu comme un paradigme de programmation parallèle pertinent par certains acteurs du HPC. En particulier, les constructeurs de machines parallèles de type SMP (avec notamment un espace d'adressage global pour les processus), ont longtemps considéré que dans un tel contexte, faire du passage de message introduisait un surcoût réduisant les performances et était donc dispensable. Or, avec l'arrivée massive des grappes de nœuds uniprocesseurs où le passage de messages était indispensable (contexte de mémoire physiquement distribuée), ce problème du surcoût ne s'est pas posé de façon trop prégnante. L'arrivée des grappes de machines bi- ou quadriprocesseurs n'a pas bouleversé fondamentalement la donne, notamment parce que des implémentations offrant un très bon niveau de performances pour de telles architectures (cf. section 2.5) ont fait leur apparition, ce qui n'a fait que repousser le problème. De plus, ainsi que nous l'avons expliqué en section 3.1.5.5, les modèles hybrides mélangeant MPI avec des threads offraient des performances en-deçà des attentes.

Ce n'est qu'avec l'apparition des grappes de nœuds multicœurs que les questions du surcoût de MPI et de la pertinence de l'utilisation d'un parallélisme intra-nœud à base de threads couplé à des échanges de messages pour la partie inter-nœuds vont se poser avec une acuité renouvelée. Il faudra cependant un temps d'adaptation pour comprendre que le maximum de performances n'est pas obligatoirement atteint avec la solution utilisant un seul processus MPI par nœud et autant de threads que de cœurs dans le nœud. Un travail de sensibilisation mené dans les centres de calcul est dès lors nécessaire, comme le montre la figure 4.1 (extraite de [82]). Trouver la bonne granularité entre processus MPI et threads n'est pas trivial et demande souvent d'effectuer des tests pour déterminer d'une part le bon nombre total de tâches et d'autre part le bon ratio entre processus MPI et threads sur chaque nœud. Jérôme CLET-ORTEGA proposera d'ailleurs dans sa thèse [38] (co-encadrée avec Raymond NAMYST) un environnement permettant d'automatiser ces calculs. Il convient de noter que ce problème ne se pose que pour les implémentations qui utilisent des processus lourds en tant que *MPI Process* et que celles basées sur des threads ne sont pas touchées puisque les différents processus MPI se partagent en pratique un même espace d'adressage, ce qui permet des optimisations intéressantes pour les échanges de données intra-nœuds. D'après l'étude de 2017 de l'*Exascale Computing Project* [13], 79% des applications utilisent déjà ou prévoient l'utilisation de threads afin d'éviter le surcoût lié à MPI dans le cas de communications intra-nœuds. Il s'agit donc d'une problématique importante pour les années à venir.

Cependant, utiliser les threads ne constitue pas l'unique solution pour éviter le surcoût du passage de messages dans un contexte intra-nœud. Hoefler *et al.* [89] propose en effet un modèle hybride différent : MPI + MPI. Il s'agit d'un modèle basé sur des processus MPI dans lequel c'est l'application elle-même qui va gérer les différents types de communications (i.e. intra- et inter-nœuds) sans déléguer cette tâche à l'implémentation MPI. Ainsi il est possible au sein d'un nœud de court-circuiter toute la pile logicielle de MPI. Dans le cas de communications par mémoire partagée, il devient possible d'utiliser de simples opérations *load* et *store*, clairement plus efficaces que des `MPI_Send/MPI_Recv`.

Évidemment, tout cela repose sur la possibilité pour les applications de déterminer qu'un certain degré de localité existe entre les processus, en particulier qu'ils s'exécutent sur un même nœud. Le standard MPI propose deux mécanismes complémentaires pour arriver à un

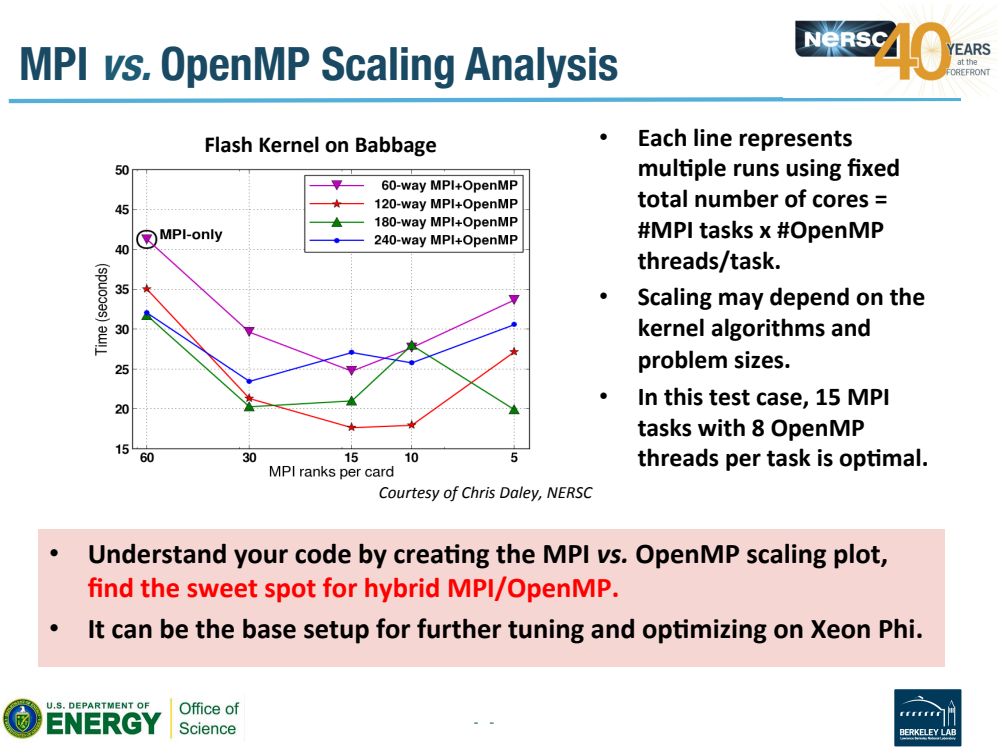


FIGURE 4.1 – Les recommandations du NERSC en cas d'utilisation d'un modèle hybride de programmation.

tel résultat :

— la fonction `MPI_Comm_split_type` appelée avec la valeur de `split_type` prédéfinie `MPI_COMM_TYPE_SHARED` [6] permet la création de communicateurs dont les processus membres ont la possibilité de créer des zones de mémoire partagée. Cependant, la méthode de création de telles zones n'est pas spécifiée et n'importe quelle méthode habituelle peut potentiellement être employée (`mmap`, segments System V, etc.), ce qui nuit à la portabilité du code applicatif ;
— une fonction de création de zones de mémoire partagée, qui va justement pallier le problème de portabilité rencontré avec l'utilisation de `MPI_Comm_split_type`. Cette fonction, `MPI_Win_allocate_shared` a été introduite récemment (MPI 3.1, soit la version la plus récente du standard).

Cependant, cette méthode, qui permet effectivement de mieux exploiter les capacités des machines multicœurs en éliminant le surcoût lié au passage de messages, ne permet pas de vraiment tirer parti de la hiérarchie mémoire complexe (surtout au niveau des caches). En effet, ce mécanisme n'offre aucun moyen de comprendre si des ressources mémoires (autre que la mémoire principale) sont partagées (ou non) entre processus MPI.

6. C'est d'ailleurs la *seule* valeur actuellement définie dans MPI, mais nous y reviendrons en section 4.2.

4.1.3 Sessions MPI et ensembles de processus

Les *sessions* [90] sont une nouveauté dans MPI mais techniquement ne font pas encore partie du standard. Les sessions sont en discussion au sein du Forum MPI depuis 2016 et seront selon toute vraisemblance acceptées dans la prochaine mouture du standard [7]. À l'origine, les sessions ont été proposées pour permettre une meilleure composabilité entre différentes bibliothèques utilisant toutes MPI et qui sont regroupées (cas de couplage de codes par exemple). Dans une telle situation peut se poser un problème d'initialisation (de MPI) car les différentes bibliothèques n'ont pas forcément conscience de la présence d'autres entités utilisant aussi MPI. Or, le standard stipule que `MPI_Init` et `MPI_Finalize` ne doivent être appelées qu'une seule fois au cours de l'exécution de l'application. Il faut donc faire des contorsions pour arriver à quelque chose de fonctionnel, mais dans le cas d'applications MPI multithreadées, les problèmes sont potentiellement insolubles.

Les sessions permettent de résoudre ce problème en créant des ensembles de processus (des *psets*) qui peuvent être utilisés pour créer des groupes puis des communicateurs. Avec les sessions, les communicateurs par défaut habituels `MPI_COMM_WORLD` et `MPI_COMM_SELF` n'existent plus au démarrage de MPI et doivent être explicitement construits si nécessaire. Les *psets* sont des objets relativement légers, encore plus que les groupes et définitivement plus que les communicateurs, ce qui permet un certain passage à l'échelle et réduit l'empreinte mémoire de l'implémentation MPI. Des informations peuvent être associées à des *psets* et en particulier des informations relatives au matériel partagé par des processus membres d'un *pset*. Cette information est dénommée URI, pour *Universal Resource Identifier*.

La figure 4.2 donne un exemple d'une application composée d'un couplage de deux codes (`app://ocean` et `app://atmos`), comprenant respectivement trois et deux processus (`mpi://SELF`). Ces différents processus sont physiquement répartis sur des nœuds distincts (`location://rack/17` et `location://rack/23`). L'ensemble de ces processus est considéré comme une seule application MPI (`mpi://WORLD`), qui est en fait un unique *job* du point de vue du gestionnaire de ressources responsable du déploiement (`job://12492`). Cette notion d'URI permet les interactions entre MPI et les différents éléments de l'écosystème HPC comme les gestionnaires de ressources, les gestionnaires de processus, etc. En effet, ces derniers seraient capables d'attribuer des URI reconnus et utilisables au niveau de la bibliothèque MPI. Bien que `MPI_COMM_WORLD` et `MPI_COMM_SELF` soient absents, les URIs `mpi://SELF` et `mpi://WORLD` sont obligatoires car ils doivent permettre justement d'éventuellement créer ces communicateurs si besoin est. Ainsi, un logiciel de découverte de topologie matérielle, comme hwloc pourrait créer tout un ensemble d'URIs correspondant aux différents niveaux de hiérarchie mémoire, comme `hwloc://package`, `hwloc://L3`, `hwloc://L2`, etc. D'un certain point de vue, cela constitue une généralisation de la seule fonction dans MPI qui permet d'obtenir des informations sur le matériel sous-jacent, `MPI_Get_processor_name`.

L'avantage de l'approche *sessions* + URIs est qu'elle présente des ensembles de processus sans formuler d'hypothèses de hiérarchie ce qui permet de représenter n'importe quel type de topologie matérielle. En revanche, cela impose à l'utilisateur de bien connaître le matériel pour arriver à l'exploiter correctement, ce qui d'un point de vue abstraction pèche quelque peu. Au final, cette solution est prometteuse lorsque l'on cherche à obtenir une représentation explicite de la topologie matérielle et à exprimer des degrés de localité entre ensembles de processus. Il convient toutefois de noter qu'il n'existe pas encore à l'heure actuelle d'implémentation des *sessions*, hormis sous forme de prototype.

7. MPI 4.0, attendu dans quelques mois.

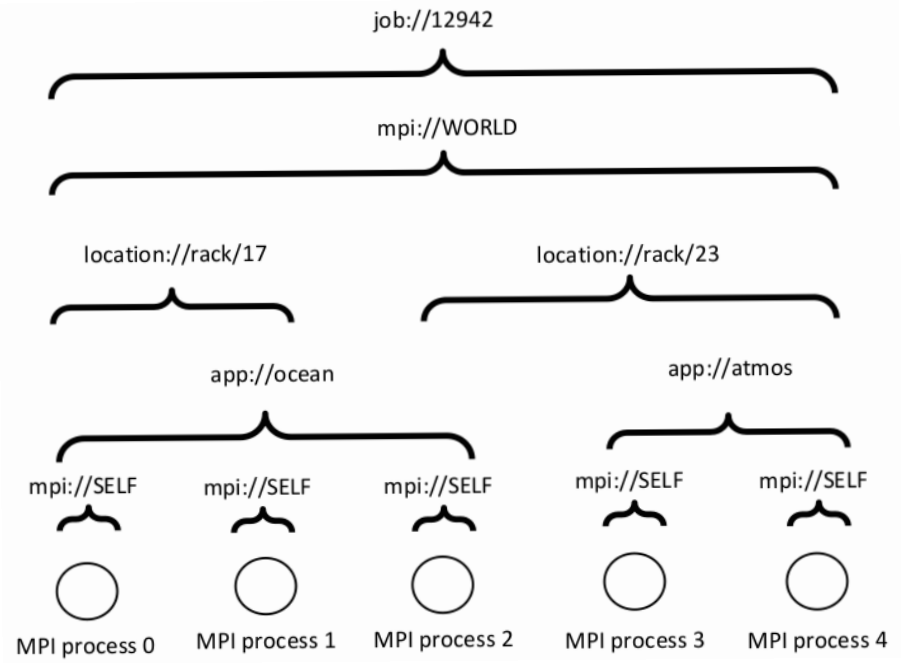


FIGURE 4.2 – Un exemple d'application MPI avec plusieurs sessions simultanées (figure extraite de la propostion soumise au MPI Forum pour les sessions).

4.1.4 Gestionnaires de processus/supports exécutifs

Le gestionnaire de processus est devenu au fil du temps un élément de plus en plus important dans l'écosystème MPI. En effet, il est le responsable du déploiement des applications et génère puis transmet des informations importantes pour le bon déroulement de l'application. À titre d'exemple, c'est très souvent le gestionnaire/lanceur de processus qui s'occupe d'affecter des identifiants uniques aux processus MPI, identifiants qui seront utilisés pour leur donner leur numéro de rang dans `MPI_COMM_WORLD`. Dans le cas des sessions, ce sont encore ces gestionnaires de processus qui vont créer les URIs disponibles au début de l'application (cf. figure 4.2). De plus, ces gestionnaires font non seulement le lien entre la bibliothèque MPI et le système cible (ce qui comprend surtout le système d'exploitation mais parfois aussi le matériel), mais également avec d'autres entités de la pile HPC comme le gestionnaire/allocateur de ressources matérielles.

Ainsi que nous l'avons expliqué en section 3.2.2.3, l'émergence puis la généralisation des grappes de nœuds multicœurs a considérablement complexifié la tâche dévolue au gestionnaire de processus, qui doit être capable de pouvoir gérer beaucoup plus finement le placement des processus MPI sur les différentes unités physiques de calcul afin de gagner en performances (cf. [CI10]). Le temps où fournir un simple *machinefile* pour en déduire les numéros de rang et le placement est révolu (comme dans Brandfass *et al.* [26]). Cela est d'autant plus vrai lorsque ce sont des applications hybrides qui doivent être déployées car les contraintes du second modèle de programmation doivent pouvoir être prises en compte [8]. Néanmoins, le fait que cet élément ne fasse pas partie du standard rend les choses très difficiles : il faut souvent réviser le déploiement des applications en raison de changements d'options ou de leur sémantique au niveau des gestionnaires de processus.

Le second problème (outre l'aspect non-standard de la chose) provient surtout du total

8. Par le truchement de variables d'environnement bien souvent, ce qui n'est pas des plus dynamique.

manque d'interactions entre l'application MPI et le gestionnaire de processus. S'il existe clairement de fortes interactions (et c'est obligé) entre l'implémentation MPI et le gestionnaire, rien n'est remonté au niveau de l'application proprement dite. La seule concession faite par le standard reste la présence de la fonction `MPI_Get_processor_name`, qui peut éventuellement fournir des informations de topologie matérielle à l'application, mais cela ne peut aller au-delà de la connaissance de l'existence d'un certain nombre de nœuds ou du partage possible de mémoire entre processus. Là encore, une partie importante de la hiérarchie n'est pas exploitée (ni exploitable).

4.1.5 Solutions *ad-hoc*

Constatant le manque actuel de mécanismes de description et d'exploitation des topologies matérielles dans MPI, certains constructeurs se sont résolus à proposer non pas une implémentation de MPI optimisée pour leur système propre (un peu à la manière de ce qui a été – rarement – proposé pour les topologies virtuelles) mais de nouvelles fonctions permettant aux applications de se structurer en prenant en compte l'architecture matérielle cible.

C'est notamment le cas de Fujitsu qui propose des extensions de l'interface MPI pour son réseau d'interconnexion Tofu (un Tore 6D) [63]. Le support exécutif (pour le déploiement des applications) est également capable de fournir des ressources organisées selon une topologie matérielle particulière. Ainsi la commande `pjsub` possède une option permettant de préciser le nombre de dimensions souhaitées [9] pour le Tore (uni-, bi- ou tridimensionnel). Le tableau 4.1

Nom de la fonction	**Description/Résultat produit**
`FJMPI_Topology_get_dimension`	Nombre de dimensions fournies à `MPI_COMM_WORLD`
`FJMPI_Topology_get_shape`	Forme (1D, 2D, 3D) donnée à `MPI_COMM_WORLD`
`FJMPI_Topology_rank2x`	Coordonnée X du rang appelant
`FJMPI_Topology_rank2xy`	Coordonnée XY du rang appelant
`FJMPI_Topology_rank2xyz`	Coordonnée XYZ du rang appelant
`FJMPI_Topology_x2rank`	Rang à partir de la coordonnée X
`FJMPI_Topology_xy2rank`	Rang à partir de la coordonnée XY
`FJMPI_Topology_xyz2rank`	Rang à partir de la coordonnée XYZ
`FJMPI_Topology_cart_reorder`	Valeur déterminant le rang dans une topologie cartésienne
`FJMPI_Topology_sys_rank2xyzabc`	Coordonnées de Tofu à partir du rang
`FJMPI_Topology_sys_xyzabc2rank`	Rang à partir des coordonnées Tofu
`FJMPI_Topology_rel_rank2xyzabc`	Coordonnées relatives de Tofu à partir du rang
`FJMPI_Topology_rel_xyzabc2rank`	Rang à partir des coordonnées relatives de Tofu

TABLE 4.1 – Extensions topologiques de Fujitsu pour le réseau toroïdal Tofu.

montre l'interface qui a été développée. IL est clair que si une telle interface fournit aux applications de précieux services, cette approche n'est pas du tout généralisable. Cela démontre néanmoins qu'il existe des applications qui souhaitent utiliser la topologie matérielle sous un angle «cartographique» c'est-à-dire avec des informations précises quant à la localisation des processus MPI dans la machine. Là encore, il s'agit d'une approche qui privilégie une connaissance fine de la plate-forme cible, ce qui n'est pas systématiquement le cas des développeurs.

9. Une *shape* dans la terminologie adoptée par Fujitsu.

4.2 Vers une exploitation de la localité avec les communicateurs MPI

Il ressort de l'exposition des mécanismes décrits précédemment qu'il n'existe pas de moyen standard avec un degré suffisant d'abstraction répondant aux besoins applicatifs vis-à-vis de la localité et de l'exploitation des topologies matérielles. Rappelons que ces topologies sont de plus en plus complexes et exhibent de nombreux niveaux de hiérarchie à l'intérieur des nœuds de calcul et que cette hiérarchie va vraisemblablement gagner encore plus en complexité dans les années qui viennent [10]. Le défi est donc de proposer des mécanismes à la fois pertinents, répondant correctement aux besoins applicatifs, et suffisamment pérennes pour éviter d'avoir à refaire le même travail dans le futur. Ce dernier point est loin d'être anecdotique : le MPI Forum n'est pas une instance de discussion réputée être très réactive (ce qui parfois est utile) et le processus d'acceptation de nouvelles fonctionnalités peut parfois pâtir d'une inertie mâtinée d'une certaine dose de conservatisme. Il est donc exclu de proposer quelque chose devant être modifié dans quelques années ou présentant un caractère potentiellement obsolète. En revanche, effectuer des ajouts reste acceptable.

4.2.1 Représentation d'une topologie matérielle avec des communicateurs MPI

L'examen du standard MPI nous a convaincus rapidement qu'il était inutile d'introduire de nouvelles abstractions dans MPI. En effet, certains objets pré-existants sont à même, selon nous, de pouvoir répondre aux besoins formulés. En particulier, les communicateurs MPI, que l'on peut voir en première approximation comme des groupes de processus, sont capable de représenter différents niveaux de hiérarchie physique présents dans la plate-forme cible. De plus, ils sont utilisés systématiquement dans les applications MPI, avec *a minima* `MPI_COMM_WORLD`. Les développeurs sont donc habitués à la présence ou la manipulation des communicateurs MPI. Également il n'est pas inutile de rappeler que les topologies virtuelles se basent aussi sur l'utilisation de communicateurs, auxquels une information est rattachée. Enfin, l'approche «sessions» ne rend pas les communicateurs caducs pour autant.

4.2.1.1 Précédents travaux et tentatives

L'idée d'employer des communicateurs MPI n'est pas nouvelle et cette approche a déjà été tentée dans divers contextes. J'avais déjà utilisé dans ma thèse [Th1] un ensemble de communicateurs pour représenter les différents réseaux physiques d'interconnexion présents dans une grappes de grappe. En revanche, je proposais de gérer la hiérarchie mémoire intra-nœud en interne dans l'implémentation, ce qui n'est plus satisfaisant avec les machines actuelles ainsi que nous l'avons expliqué.

De façon similaire, mais pour les grilles de calcul, MPICH-G2 [104] présente ses quatre niveaux de hiérarchie à l'utilisateur à l'aide de communicateurs dédiés pour aider à structurer l'application. Ainsi, passer par un communicateur plutôt qu'un autre permet de clairement déterminer quel type de réseau va être utilisé et au final de donner des indications sur le niveau de performances possible. Toujours pour les grilles, López *et al.* [127] propose des extensions de l'interface MPI pour aider à structurer l'application. De façon intéressante, cette interface présente des similitudes avec celle de hwloc (mais lui est bien antérieure), avec des fonctions telles que `HMPI_Get_parent`, `HMPI_Get_sons`, `HMPI_Get_siblings`, ou `HMPI_Get_level`. Une notion de `hrank` est également introduite permettant de naviguer dans les niveaux topologiques. Cependant ces deux solutions ne s'intéressent pas aux niveaux intra-nœuds.

10. L'arrivée des mémoires non volatiles et des disques SSD brouillent les frontières établies jusqu'à présent et tendent vers la création d'un continuum que les développeurs vont bien devoir exploiter à l'avenir.

Un certain nombre de travaux pour les niveaux intra-nœuds existent cependant, mais ont été proposés plutôt à l'époque de la première génération d'implémentation de MPI, ce qui fait qu'ils n'ont sans doute pas eu l'impact qu'ils auraient dû. Ainsi, Träff [196] a proposé assez tôt une idée similaire de communicateurs topologiques afin de mieux utiliser la hiérarchie mémoire intra-nœud. Il propose une hiérarchie avec inclusion non stricte et un nombre prédéfini de niveaux la composant. HMPI [115] présente également une hiérarchie de communicateurs pour aider à mieux structurer une application. Il s'agit cependant d'une bibliothèque externe et n'est donc pas portable. Plus récemment (2011), l'idée d'introduire des communicateurs nommés (par exemple, `MPI_COMM_SOCKET`, `MPI_COMM_L3` ,etc) a été émise devant le Forum MPI mais rapidement écartée : en effet, un tel nommage est très rigide et exposé à trois risques : 1) d'une part un nom pourrait disparaître (c'est déjà le cas avec le terme *socket* qui est remplacé par *package* par les fabricants de processeurs) 2) d'autre part, l'apparition de nouveaux noms obligerait à un nouvel effort de standardisation (et sans doute régulièrement), ce qui est déraisonnable et 3) deux noms identiques sont susceptibles de désigner des ressources différentes selon les architectures considérées.

Toutes ces solutions présentent au développeur des communicateurs qui sont créés au moment de l'initialisation et leur nombre ainsi que leur dénomination est fixé par l'implémentation de MPI. La configuration est donc relativement statique et demande une adaptation lorsqu'il faut changer de plate-forme.

Enfin, cette notion de regroupement n'a pas été étudiée que dans MPI et le passage de messages. Dans l'équipe Runtime Samuel THIBAULT a utilisé un concept similaire, appelé *bulles*, afin de placer dynamiquement des threads sur les différents cœurs d'une machine hiérarchique. Il a implémenté cela dans son ordonnanceur Bubble Sched [175], destiné à être utilisé comme support exécutif pour une bibliothèque OpenMP. Dans le cas de Bubble Sched, les threads sont regroupés au sein de *bulles* selon leur affinité et les bulles sont réparties sur les différentes ressources de calcul. Des bulles peuvent contenir d'autres bulles pour indiquer des liens d'affinité encore plus forts. Pour la répartition, il faut «percer» les bulles et le placement est effectué de façon *top-down* jusqu'à ce qu'il n' y ait plus de bulles mais plus que des threads à placer. Il est aussi possible de recréer des bulles ou de les fusionner : cela correspond à la fin d'une section parallèle OpenMP quand on ne souhaite pas recréer de nouveaux threads systématiquement mais utiliser un *pool* de threads disponibles.

4.2.1.2 Principes des communicateurs topologiques

L'idée principale est donc de pouvoir accéder aux différents niveaux topologiques de la plate-forme cible, aussi bien inter-nœuds qu'intra-nœuds via une hiérarchie de communicateurs MPI. Au sein de cette hiérarchie chaque communicateur correspond à une ressource physique partagée par tous les processus MPI membres du communicateur. Par exemple, si un processus partage un cache L2 ainsi qu'un cache L3 avec d'autres processus, il appartiendra simultanément au communicateur regroupant tous les processus partageant ce cache L2 et au communicateur regroupant tous les processus partageant ce cache L3.

Selon nous, ces communicateurs ne devraient pas dépendre de caractéristiques rigides comme une profondeur fixe de la hiérarchie ni de noms fixes pour les différents niveaux. L'accès à ces niveaux va permettre de structurer les applications pour mieux prendre en compte la localité et mieux exploiter le matériel sous-jacent. Nous voyons en fait deux publics distincts pouvant tirer parti d'une fonctionnalité de ce type :

— d'une part les développeurs ayant déjà une bonne connaissance de la plate-forme pour laquelle ils écrivent (ou adaptent) leur code. Dans ce cas, il est possible de vouloir accéder directement à un niveau topologique bien précis ;

— d'autre part les développeurs désireux de structurer leur application en prenant en compte la localité au sens large, sans se préoccuper des détails du matériel, ni du placement des processus applicatifs. L'utilisation des communicateurs topologiques permet de déterminer des affinités entre processus et plus le communicateur se situe profondément dans la hiérarchie, plus cette affinité (du point de vue du matériel) est forte, c'est à dire que les processus sont localisés sur des entités physiques de calcul proches physiquement parlant. Là encore, nous formulons une hypothèse du type de celle exprimée en section 2.2.2, à savoir que la vitesse des communications augmente avec la profondeur du niveau. Cette hypothèse n'est sans doute pas vraie dans tous les cas, mais elle l'est pour une grande majorité des machines actuelles et la complexification de la hiérarchie mémoire ne va pas la remettre fondamentalement en cause. Des applications de type multiplication hiérarchique de matrices [167] ou multiplication parallèle matrice creuse - vecteur [20] constituent de très bonnes candidates pour une utilisation des communicateurs topologiques telle que nous venons de la décrire.

Également, les communicateurs vont permettre l'utilisation des opérations de communication collectives. Dans le cas qui nous occupe, ces collectives pourront être optimisées non seulement par la bibliothèque MPI si elle possède des collectives *topo-aware* mais également par le fait que les échanges de données seront d'autant plus rapides que le niveau topologique sera bas. Il s'agit d'un aspect supplémentaire pour encourager une structuration des applications prenant plus en compte la localité.

4.2.2 Extensions proposées du standard MPI

Depuis sa première version en 1994, le standard MPI n'a cessé de croître en termes de fonctions et de fonctionnalités disponibles. Cela pose d'ailleurs un problème puisque les développeurs d'applications sont de moins en moins capables d'appréhender l'éventail complet des possibilités offertes par l'interface MPI, ni les subtilités découlant des implémentations. Il me semble qu'à l'avenir vont apparaître de plus en plus de *power users* de MPI, c'est-à-dire des gens faisant l'interface entre les développeurs des implémentations proprement dites et les développeurs d'applications. Ces intermédiaires vont pouvoir analyser les besoins applicatifs et proposer les outils adéquats pour les applications. Des bibliothèques intermédiaires, basées sur MPI et offrant des fonctionnalités n'ayant pas pour vocation à être intégrées directement dans MPI devraient voir le jour également. Ainsi, notre approche est d'introduire le strict minimum pour ne pas surcharger davantage une interface déjà bien fournie et de préférence d'étendre des mécanismes pré-existants. Nous allons maintenant décrire les différentes fonctions et fonctionnalités que nous nous proposons de rajouter.

4.2.2.1 Création des communicateurs topologiques

L'étude du standard MPI montre qu'il existe quelques fonctions permettant la création de nouveaux communicateurs :

— `MPI_Comm_create` permet la création d'un communicateur à partir d'un groupe de processus (au sens MPI) ;
— `MPI_Comm_dup` permet de créer une copie d'un communicateur passé en argument ;
— `MPI_Comm_split` permet la création d'un ensemble de *sous-communicateurs* disjoints, partitionnant de fait le communicateur d'origine (également appelé communicateur *parent*). Ce k-partitionnement du communicateur est effectué selon un paramètre `color` : tous les processus indiquant la même couleur se retrouveront membres du même sous-communicateur résultat. Ces processus peuvent même être renumérotés dans le sous-communicateur grâce au paramètre `key`.

Notre approche repose sur la création de communicateurs représentant une certaine localité s'exprimant sous la forme du **partage de ressources physiques** entre processus membres. Il serait alors tentant de vouloir partitionner un communicateur tel que `MPI_COMM_WORLD` (bon candidat mais pas obligatoire) pour arriver au résultat désiré : l'information de partage serait véhiculée par l'argument de *couleur* (`color`) de `MPI_Comm_split`. Cependant, le résultat ne serait pas celui escompté, car si la même couleur est fournie à cette fonction par des processus, alors ils se retrouveront tous membres d'un sous-communicateur résultat. Prenons par exemple le cas de processus placés sur différents nœuds physiques (un processus étant assigné à un cœur dédié) qui possèdent plusieurs caches L3. Si nous voulons obtenir autant de communicateurs que de caches L3, il n'est pas possible de donner une seule couleur pour tous les caches sinon *tous* les processus seront dans le sous-communicateur topologique associé. L'idéal serait de fournir une *unique* couleur, mais que l'interprétation de cette dernière soit contextuelle et variable selon les processus appelants. Or, il se trouve que le standard MPI fournit justement une telle fonction : `MPI_Comm_split_type`, décrite ci-après.

Extension de `MPI_Comm_split_type` Nous avons déjà évoqué cette fonction dans la section 4.1.2 consacrée aux constructions de MPI permettant le partage d'espace d'adressage entre processus. Cette fonction partitionne le communicateur d'entrée en sous-communicateurs disjoints en se basant sur la valeur assignée au paramètre `split_type`. La différence avec `MPI_Comm_split` est que pour une même valeur de `split_type`, il est possible d'obtenir des sous-communicateurs distincts alors dans le cas de `MPI_Comm_split`, tous les processus possédant la même couleur se retrouveront membres du même communicateur de sortie. Le prototype de cette fonction est le suivant :

```
int  MPI_Comm_split_type(MPI_Comm oldcomm,
                         int split_type,
                         int key,
                         MPI_Info info,
                         MPI_Comm *newcomm);
```

Avec pour paramètres :

- IN `oldcomm` : communicateur d'entrée à partitionner
- IN `split_type` : type de processus à regrouper
- IN `key` : contrôle du rang dans le communicateur de sortie
- IN `info` : informations pour guider le partitionnement
- OUT `newcomm` : communicateur de sortie

Dans la version actuelle de MPI, une seule valeur pour `split_type` est définie : `MPI_COMM_TYPE_SHARED`, qui permet de créer des communicateurs où les processus seraient capables de partager une partie de leurs espaces d'adressage respectifs[11]. Le standard MPI indique que les implémentations ont la possibilité de définir de nouvelles valeurs de `split_type` qui leur sont propres afin mettre en place des comportements spécifiques (c'est d'ailleurs ce qui est fait dans Open MPI, cf. section 4.2.5.2). La flexibilité intrinsèque de cette approche est contrebalancée par son manque total de portabilité. Nous proposons donc d'enrichir la liste actuelle de valeurs possibles pour le paramètre `split_type` en y ajoutant une nouvelle : `MPI_COMM_TYPE_HW_SUBDOMAIN`.

11. Cette valeur est très souvent utilisée pour créer des communicateurs correspondants aux différents nœuds de calcul, mais cela est incorrect de façon absolue. En effet si des nœuds sont interconnectés par un réseau de type SCI (Scalable Coherent Interface) ou bien par une technologie de type NUMALink, alors le partage d'espace d'adressage devient possible entre nœuds.

Modes de partitionnement Le paramètre `info`[12] peut être utilisé (de façon optionnelle) pour guider le partitionnement. Ainsi si la clef `mpi_hw_subdomain_type` est définie dans `info`, le partitionnement est dit **guidé**. Il est **non-guidé** dans le cas contraire. Dans le cas *guidé*, la valeur de la clef `mpi_hw_subdomain_type` est une chaîne de caractères désignant le nom de la ressource matérielle pour laquelle on souhaite créer des sous-communicateurs, par exemple «L3», «package» ou «NUMANode». Comme le nommage présente des problèmes de standardisation, nous ne proposons pas que MPI impose un ensemble de valeurs prédéfinies (cf. la discussion en 4.2.4). La seule valeur prédéfinie est `mpi_shared_memory`, qui produit le même résultat qu'un appel à `MPI_Comm_split_type` avec pour `split_type` la valeur `MPI_COMM_TYPE_SHARED`. Le mode *non-guidé* peut être utilisé dans des applications structurées hiérarchiquement telles que décrites dans [20] ou [167][13].

Dans le cas *non-guidé*, le partitionnement est guidé par la structure du matériel (obtenue via un outil adéquat). Le sous-communicateur produit doit correspondre au *niveau suivant* dans la hiérarchie réprésentant la plate-forme cible. Ce sous-communicateur peut alors être utilisé pour des appels suivants à `MPI_Comm_split_type` pour produire d'autres sous-communicateurs correspondants à des ressources situées de plus en plus profondément dans la hiérarchie matérielle (des exemples sont détaillés en section 4.2.3).

Propriétes des sous-communicateurs Les sous-communicateurs produits à l'aide de la valeur `MPI_COMM_TYPE_HW_SUBDOMAIN` doivent se conformer aux règles suivantes **dans le cas non-guidé** :

- si plusieurs niveaux sont potentiellement sélectionnables comme résultat du partitionnement, alors le niveau *le plus bas dans la hiérarchie* est choisi. Cela est destiné à éviter de créer des communicateurs redondants et identiques (donc inutiles). Par exemple, si un processus est sur un nœud (niveau «Node») et veut faire une opération de partitionnement (non-guidée) et que le niveau suivant correspond à la fois à «NUMANode», «Package» et «L3» (le nœud possède plusieurs *NUMANodes* composés chacun d'un unique *Package* et d'un unique cache L3), alors le sous-communicateur produit correspondra au niveau de cache L3 ;
- chaque sous-communicateur produit doit être inclus strictement dans le communicateur-parent. C'est-à-dire qu'un appel à
  `MPI_COMM_COMPARE(comm,newcomm,result)`
  retournera obligatoirement
  `result = MPI_UNEQUAL`
  Cette propriété assure la non-redondance dans les sous-communicateurs produits et créé une condition d'arrêt en cas d'utilisation récursive de la fonctionnalité. Il s'agit d'une différence importante avec la proposition de Träff [196] ;
- Si aucun communicateur valide ne peut être créé, la fonction retourne `newcomm = MPI_COMM_NULL`. Un seul cas est susceptible de produire un tel résultat : quand le dernier niveau (le plus bas) de la hiérarchie a été atteint. En particulier, l'assignation (*binding*) d'un process peut modifier la «vision» que ce dernier possède de la hiérarchie matérielle et par conséquent, il est possible que ce dernier niveau change en fonction du *binding*. Par exemple, si un processus est lié à un cache L3, aucune information ne sera disponible (et retournée) en-dessous de ce niveau car le processus est susceptible d'utiliser n'importe quel cache L2. Du point de vue du processus, le dernier

12. Dans MPI, un objet `info` sert à stocker un ensemble de couples (*clef*, *valeur*) où la valeur d'une clef est toujours une chaîne de caractères.

13. Une évalution de performances de la multiplication de matrice hiérarchique utilisant nos communicateurs est disponible dans [RI2].

niveau est donc celui du cache L3, quand bien même un niveau L2 existerait encore en-dessous.

Ainsi, dans le cas non-guidé, les communicateurs forment une hiérarchie qui réprésente celle du matériel. Un processus membre du communicateur de niveau n fait également partie des communicateurs des niveaux 0 à $n-1$. Enfin, il est important de noter qu'à aucun moment nous ne faisons de distinction entre la hiérarchie du réseau et la hiérarchie mémoire interne aux nœuds de calcul.

Dans le cas **guidé**, étant donné qu'un partitionnement est effectué à un niveau donné, ce qui n'implique pas de hiérarchie *a priori*, les propriétés sont un peu différentes. De plus, il est nécessaire d'avoir un comportement cohérent entre l'utilisation de `MPI_COMM_TYPE_SHARED` et de `MPI_COMM_TYPE_HW_SUBDOMAIN` avec `mpi_shared_memory` pour valeur du paramètre `info`. Ainsi `newcomm` = `MPI_COMM_NULL` dans les cas suivants : 1) le niveau explicitement requis (ou un niveau équivalent dans le cas de redondances) n'existe pas dans la hiérarchie ou n'est plus disponible (par exemple, des ressources sont éteintes/allumées dynamiquement en cours d'exécution) ; 2) le dernier niveau (le plus bas) de la hiérarchie a été atteint. La remarque faite dans le cas non-guidé et concernant le lien entre assignation du processus aux ressources et la définition de niveau le plus bas s'applique ici encore.

Ainsi, dans le cas guidé, il est possible que le communicateur retourné soit identique au communicateur d'entrée et soit non strictement inclus dans celui-ci.

4.2.2.2 Création des communicateurs-maîtres

Étant donné que tous les sous-communicateurs créés sont disjoints, la communication entre ces objets va mobiliser des processus du niveau supérieur, c'est-à-dire appartenants au communicateur ayant servi au partitionnement. Cependant, il nous paraît important de sélectionner un seul processus par sous-communicateur qui va se charger des échanges avec les autres sous-communicateurs (s'ils existent). C'est ce que nous appelons le communicateur-maître car tous ses membres sont chacun maître d'un sous-communicateur. Gropp [77] émet une idée similaire mais ne l'implémente pas en pratique. Ces communicateurs-maîtres forment également une forme de hiérarchie qui complémente celle des communicateurs basés sur la topologie matérielle. Il pourrait alors être intéressant d'avoir une fonction qui créé à la fois le sous-communicateur et le communicateur-maître associé :

```
int MPI_Comm_hsplit_with_roots(MPI_Comm oldcomm,
                               MPI_Info info,
                               MPI_Comm *newcomm,
                               MPI_Comm *rootscomm);
```

Avec :

IN `oldcomm` : communicateur d'entrée
IN `info` : informations supplémentaires
OUT `newcomm` : sous-communicateur
OUT `rootscomm` : communicateur maître

`newcomm` est le même sous-communicator créé par un appel à `MPI_Comm_split_type` avec `MPI_COMM_TYPE_HW_SUBDOMAIN_TOPOLOGY` comme valeur pour le paramètre `split_type`. `rootscomm` est le communicateur-maître contenant tous les processus-maîtres des `newcomm`. Un communicateur-maître valide est retourné uniquement si le processus-maître de `oldcomm` appelle cette fonction et MPI_COMM_NULL est retourné dans le cas des autres processus. Cette fonction n'utilise pas les paramètres habituels de partitionnement comme la *couleur* et la *clef* car elle est spécifique au contexte des communicateurs topologiques et donc

la valeur de couleur/`split_type` est implicite. Quant à la clef, nous utilisons le rang dans le communicateur d'entrée.

4.2.2.3 Récupération des informations d'un niveau topologique

Le mode *guidé* permet de produire des communicateurs correspondants à une demande précise, ce qui n'est pas le cas du mode *non-guidé*. Une fois que des communicateurs sont construits selon ce mode de fonctionnement, le développeur peut vouloir obtenir des informations à leur sujet (e.g. pour faire de la distribution de données). Cette récupération d'informations peut se faire avec la fonction suivante :

```
int MPI_Comm_get_hw_subdomain_info(MPI_Comm comm,
                                   int *num_comms,
                                   int *index,
                                   char *type);
```

Avec :

IN `comm` : communicateur
OUT `num_comms` : nombre de communicateurs de niveau topologique similaire
OUT `index` : index («rang») du communicateur
OUT `type` : type de ressource que le communicateur représente

Les informations fournies sont les suivantes :

— `num_comms` est un entier dont la valeur est le nombre de communicateurs de même niveau topologique et possédant le même communicateur parent ;
— `index` est interprétable comme une sorte de «rang» pour chaque communicateur. Ces numéros sont contigus et compris entre 0 et `num_comms`−1.
— `type` est une chaîne de caractères donnant des informations sur la ressource que représente le communicateur.

Ces informations sont en fait stockées dans le communicateur au moment de sa création dans un objet `info` qui lui est associé par le truchement de couples (*clef*,*valeur*). La création et récupération de cet objet `info` propre au communicateur requiert donc l'utilisation des fonctions dédiées `MPI_Comm_set_info` et `MPI_Comm_get_info`.

4.2.3 Fonctionnement des communicateurs topologiques par l'exemple

Nous allons maintenant montrer sur des exemples concrets le fonctionnement des communicateurs topologiques. Seul le mode *non-guidé* sera détaillé dans cette section étant donné que le fonctionnement du mode *guidé* est trivial. Pour les exemples, nous allons supposer qu'une application est déployée sur un ensemble de nœuds de calcul possédant une hiéarchie mémoire similaire à celle décrite par la figure 4.3. Chaque nœud est composé de deux *NUMA-Nodes* avec un unique *Package* et quatre cœurs par *Package*. Chaque *Package* possède également son propre cache de niveau 3 et un cache de niveau 2 est partagé par paire de cœurs du *Package* (nous aurons donc quatre caches de niveau 2 dans le nœud).

La création de la hiérarchie des communicateurs topologiques peut s'effecter en appelant le code suivant :

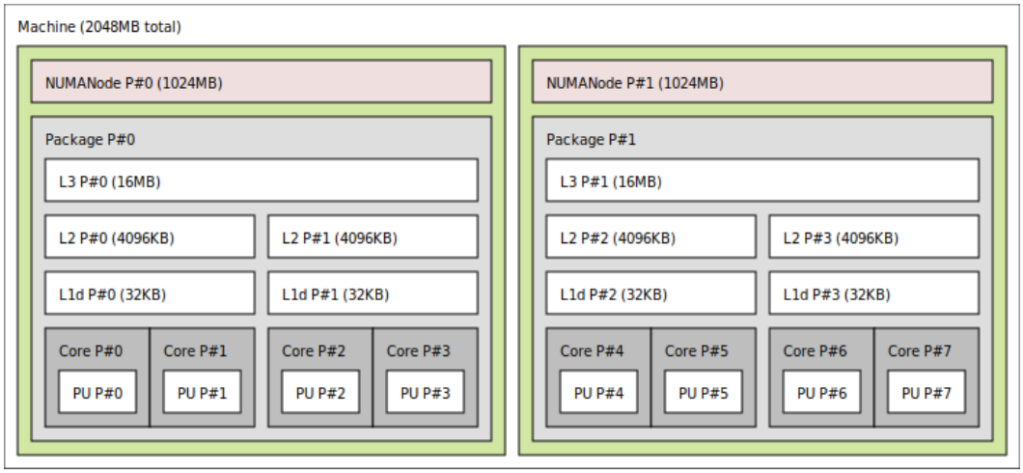


FIGURE 4.3 – Un exemple de nœud de calcul hiérarchique.

```
#define MAX_NUM_DOMAINS 32

MPI_Comm hcomm[MAX_NUM_DOMAINS];
int rank, domain_num = 0;

hcomm[domain_num] = MPI_COMM_WORLD;

while((hcomm[domain_num] != MPI_COMM_NULL)
   && (domain_num < MAX_NUM_DOMAINS-1)){
  MPI_Comm_rank(hcomm[domain_num],&rank);
  MPI_Comm_split_type(hcomm[domain_num],
                      MPI_COMM_TYPE_HW_SUBDOMAIN,
                      rank,
                      MPI_INFO_NULL,
                      &hcomm[domain_num+1]);
  ++domain_num;
}
```

Nous supposons que la constante `MAX_NUM_DOMAINS` est choisie de façon à ce que le tableau de communicateurs `hcomm` soir correctement dimensionné. Cependant, notre proposition ne fait aucune espèce d'hypothèse sur la profondeur de la hiérarchie matérielle ni sur la nature du matériel que chaque sous-communicateur est censé représenter. Nous ne fournissons pas de fonction permettant de connaître la profondeur totale de la hiérarchie car cela demanderait de parcourir deux fois la structure hwloc. Cependant, cette profondeur n'est pas très grande en pratique et l'utilisation d'un tableau alloué statiquement ne pose pas de réel problème pour l'empreinte mémoire.

4.2.3.1 Cas des politiques d'affectation homogènes

Dans notre premier exemple, nous lançons une application comportant 32 processus MPI sur une grappe de quatre nœuds ($N_k, k \in [0..3]$) tels que décrits par la figure 4.3. De plus, chaque processus est assigné à un cœur qui lui est propre. Par exemple, le processus P_x (de rang x dans `MPI_COMM_WORLD`) est assigné au cœur numéro i du nœud numéro k où $x = 8k+i$

($k \in [0..3]$ et $i \in [0..7]$) [14].

Création de la hiérarchie des communicateurs À l'exécution du morceau de code donné ci-dessus, une hiérarchie de communicateurs est créée en plusieurs étapes, avec une étape par appel à la fonction `MPI_Comm_split_type` :

— **étape 1 :** les processus étant localisés sur quatre nœuds distincts, `MPI_COMM_WORLD` recouvre ces nœuds. Comme les sous-communicateurs doivent être strictement inclus dans le communicateur-parent, de nouveaux communicateurs correspondants au niveau «Machine» seront créés pour chaque nœud. C'est-à-dire :

$\texttt{newcomm}[0] = \{P_{8k}, P_{8k+1}, \ldots, P_{8k+6}, P_{8k+7}\}$, pour chaque nœud N_k avec $k \in [0, 3]$

Dans notre exemple, c'est un résultat équivalent à un appel à `MPI_Comm_split_type` avec la valeur `MPI_COMM_TYPE_SHARED` pour l'argument `split_type`. Cependant, l'implémentation pourrait décider de créer un niveau intermédiaire entre ce niveau et `MPI_COMM_WORLD` correspondant à un niveau de *switches* réseau. Comme nous faisons l'hypothèse que tous les nœuds sont connectés au même *switch*, un tel niveau ne peut être créé.

Pour des questions de simplicité et sans perte de généralité, nous allons désormais nous concentrer uniquement sur les sous-communicateurs créés pour le nœud N_0 dans la description des étapes suivantes.

— **étape 2 :** les processus étant localisés sur le même nœud, $\texttt{newcomm}[0]$ correspond au nœud entier. Comme nous imposons une règle d'inclusion stricte pour les sous-communicateurs, le prochain sous-communicateur de la hiérarchie doit forcément correspondre à un niveau plus «petit» que le niveau «Machine». Le prochain niveau possible dans la topologie matérielle cible est donc le niveau NUMANode/Package/L3. Nous choisissons le niveau *le plus bas* en cas de redondances, c'est-à-dire le niveau «L3». Ainsi, deux nouveaux sous-communicateurs sont produits, un par cache L3 présent au sein du nœud. Plus précisément, ces communicateurs ont pour membre les processus suivants :

$\texttt{newcomm}[1] = \{P_{4i}, P_{4i+1}, P_{4i+2}, P_{4i+3}\}$, pour chaque cache L3$_i$ avec $i \in [0, 1]$

— **étape 3 :** à cette étape, nous partitionnons les communicateurs produits lors de l'étape précédente (correspondants aux caches L3/Package/NUMANode) qui deviennent les communicateurs parents. Puisque que deux caches L2/L1 sont présents pour chaque *Package*, deux nouveaux sous-communicateurs sont créés. Une nouvelle fois, le niveau le plus bas est choisi, ce qui correspond au cache L1. Pratiquement, un nouvel appel à `MPI_Comm_split_type` avec le communicateur `newcomm[1]` produira 4 communicateurs pour le nœud n_0 avec les processus suivants :

$\texttt{newcomm}[2] = \{P_{2i}, P_{2i+1}\}$, pour chaque cache L1$_i$ avec $i \in [0, 3]$

— **étape 4 :** un total de 8 nouveaux communicateurs sont créés puisque deux cœurs (ou *Processing Unit*) se partagent un cache L1. Chaque sous-communicateur correspond en fait à un unique processus :

$\texttt{newcomm}[3] = \{P_i\}$, pour chaque cœur C_i avec $i \in [0, 7]$

Notons que si P_i effectue l'appel suivant :

```
MPI_Comm_compare(newcomm[3],MPI_COMM_SELF,&result)
```

alors `result` aura pour valeur `MPI_CONGRUENT`.

— **étape 5 :** le bas de hiérarchie a été atteint lors de l'étape précédente, et aucun sous-communicateur valide n'est produit. Ainsi, tous les appels suivants à `MPI_Comm_split_type` résulteront en :

$\texttt{newcomm}[4] =$ `MPI_COMM_NULL` pour chaque cœur C_i avec $i \in [0, 7]$

14. Une politique de placement séquentielle comme définie en section 3.1.2.3.

Création des communicateurs-maîtres Dans le code présenté précédemment, `MPI_Comm_split_type` pourrait être remplacé par `MPI_Comm_hsplit_with_roots` et une seconde hiérarchie de communicateurs sera alors produite, celle des communicateurs-maîtres. Le communicateur `rootscomm[i]` est donc créé durant la même étape que `newcomm[i]`.

— **étape 1 :** Puisque 4 sous-communicateurs sont créés (un pour chaque nœud, cf. ci-dessus), le communicateur-maître contient 4 processus. Nous conservons l'ordre d'origine des processus dans les sous-communicateurs, ce qui signifie que le processus-maître est celui de plus petit rang. Ainsi :

$$\texttt{rootscomm}[0] = \{P_{8k}\}, \text{ avec } k \in [0..3]$$

Comme dans l'exemple précédent, nous ne regardons désormais dans les étapes suivantes que le fonctionnement au sein d'un unique nœud (comportement symétrique entre ces nœuds).

— **étape 2 :** Deux sous-communicateurs ont été créés (un pour chaque cache L3), un seul communicateur-maître est créé (par nœud), contenant deux processus :

$$\texttt{rootscomm}[1] = \{P_0, P_4\}$$

On notera que le processus-maître de `rootscomm[1]` fait également partie de `rootscomm[0]`.

— **étape 3 :** Chaque communicateur «L3» est divisé en 2 sous-communicateurs «L1». Par conséquent, deux communicateurs-maîtres sont créés à cette étape et chacun possède 2 processus :

$$\texttt{rootscomm}[2] = \{\mathbf{P_0}, P_2\} \text{ pour le cache L3 numéro } 0)$$
$$\texttt{rootscomm}[2] = \{\mathbf{P_4}, P_6\} \text{ pour le cache L3 numéro } 1)$$

Évidemment, les processus-maîtres des deux communicateurs `rootscomm[2]` font partie de `rootscomm[1]`.

— **étape 4 :** Nous appliquons à nouveau le même fonctionnement qu'à l'étape précédente, chaque communicateur «L1» est divisé en 2 sous-communicateurs «Cœur». Au total, 4 communicateurs-maîtres sont créés à cette étape, avec deux processus chacun :

$$\texttt{rootscomm}[3] = \{\mathbf{P_{2i}}, P_{2i+1}\} \text{ pour le cache L1 de numéro } i \text{ avec } i \in [0..3]$$

Là encore, les processus-maîtres de `rootscomm[3]` font partie d'un des deux communicateurs `rootscomm[2]`.

— **étape 5 :** la fin de la hiérarchie étant atteinte, il n'y a plus de communicateurs-maîtres qui sont créés. Ainsi :

$$\texttt{rootscomm}[4] = \texttt{MPI\_COMM\_NULL} \text{ (8 fois, une par processus)}$$

4.2.3.2 Gestion des politiques d'affectation hétérogènes

Ce second exemple va nous permettre de mettre en évidence le caractère flexible de notre proposition et de souligner sa capacité à pouvoir gérer des cas plus compliqués que l'exemple trivial exposé précédemment. En effet, dans le cas d'une politique hétérogène d'affection des processus aux unités de calcul, la création de communicateurs topologiques ne pose aucun problème et reflète cette disposition [15]. Dans l'exemple que nous décrivons, 8 processus sont déployés sur un unique nœud possédant 8 cœurs de calcul avec une politique d'affectation différente pour chacun d'entre-eux. La figure 4.4 montre une telle situation pour les 8 processus de `MPI_COMM_WORLD` : les processus de rangs 0 et 1 sont liés aux cœurs 0 et 1 respectivement, tandis que les processus 2 et 3 sont liés non pas à des cœurs précis, mais au cache L2 numéro 1. Cela signifie que pendant l'exécution de l'application, ces processus peuvent occuper tantôt le cœur 2, tantôt le cœur 3. Les processus 4, 5, 6 et 7 sont liés au NUMANode numéro 1, ils peuvent donc utiliser n'importe lequel des cœurs dépendants de ce NUMANode (i.e. les cœurs 4, 5, 6 ou 7). Là encore, rien n'empêche les processus de changer de cœur durant l'exécution de

15. Même si en pratique de tels cas devraient se révéler très rares, voire inexistants.

l'application.

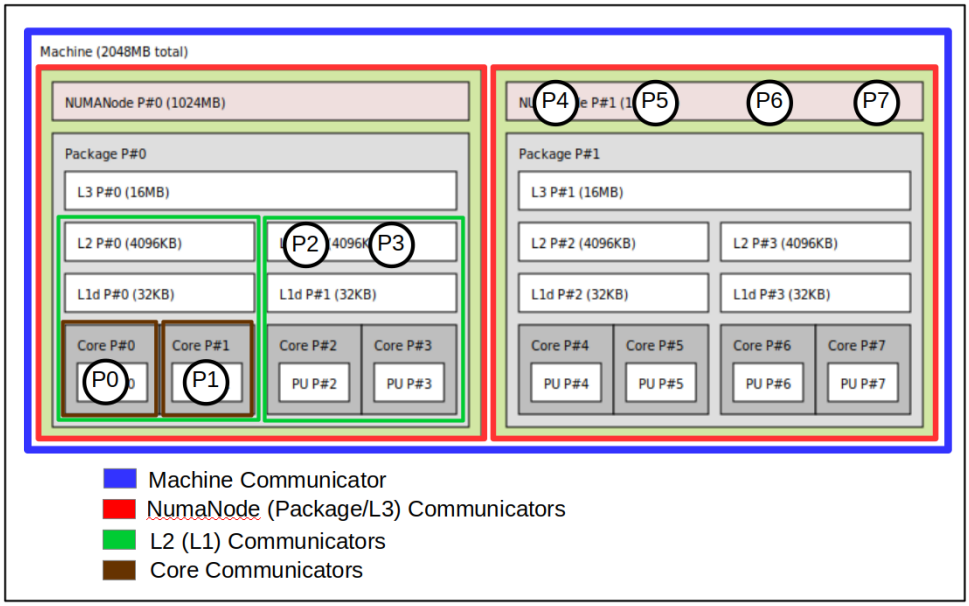


FIGURE 4.4 – Exemple d'affectation hétérogène de processus sur des unités de calcul

— **étape 1 :** Tous les processus sont placés sur le même nœud, donc MPI_COMM_WORLD correspond au nœud dans son entièreté. Ce premier communicateur correspond au niveau «Machine» (de couleur bleue sur la figure 4.4). Nous utilisons ce communicateur-parent pour le partitionnement et étant donné que le prochain niveau dans la hiérarchie matérielle est le niveau NUMANode/Package/L3, deux nouveaux sous-communicateurs sont créés car il y a deux caches L3 par machine. Tout comme dans l'exemple précédent, c'est le niveau le plus bas qui est choisi (L3).
$\texttt{newcomm}[0] = \{P_{4i}, P_{4i+1}, P_{4i+2}, P_{4i+3}\}$ pour chaque cache L3 numéro i avec $i \in [0, 1]$
Ces communicateurs apparaissent en rouge sur la figure 4.4.
— **étape 2 :** Durant cette étape, les communicateurs `newcomm[0]` sont traités de façon différente car les processus membres possèdent des politiques d'affectation différentes. Dans le cas du communicateur représentant le cache L3 numéro 0 (partie de gauche sur la figure 4.4), étant donné que tous les processus sont liés à au moins un cache L1, la même situation que dans l'étape 2 de l'exemple homogène se produit. En revanche, le cas du cache L3 numéro 1 (partie de droite sur la figure 4.4) est bien différente puisqu'aucune information de hiérarchie (relative aux processus) ne peut être déterminée car tous les processus sont susceptibles de bouger d'un cœur à l'autre pendant l'exécution de l'application. Aucun nouveau sous-communicateur n'est donc créé. Nous aurons alors :
$\texttt{newcomm}[1] = \{P_{2i}, P_{2i+1}\}$ pour chaque cache L1 de numéro i avec $i \in [0, 1]$
$\texttt{newcomm}[1] = \texttt{MPI\_COMM\_NULL}$ pour chaque cache L1 de numéro i avec $i \in [2, 3]$
Ces communicateurs sont de couleur verte sur la figure 4.4.
— **étape 3 :** À partir de cette étape, seuls des communicateurs valides peuvent être partitionnés, c'est-à-dire les communicateurs correspondants aux caches L1 dépendants du cache L3 numéro 0. Étant donné que les processus de rangs 0 et 1 sont liés à des cœurs, il est possible de créer de nouveaux sous-communicateurs pour ce niveau de hiérarchie. Cependant, les processus de rangs 2 et 3 n'étant pas liés à des ressources situées plus «profondément» que le cache L1, il n'est pas possible de créer de nou-

veaux sous-communicateurs pour eux. Une fois de plus, seuls deux nouveaux sous-communicateurs sont alors produits :

$$\texttt{newcomm}[2] = \{P_i\} \text{ pour chaque cœur numéro } i \text{ avec } i \in [0, 1])$$
$$\texttt{newcomm}[2] = \texttt{MPI\_COMM\_NULL} \text{ pour chaque cœur numéro } i \text{ avec } i \in [2, 3]$$

Ces sous-communicateurs sont de couleur marron sur la figure 4.4.

— **étape 4 :** Pour cette dernière étape, le bas de la hiérarchie est atteint pour les processus de rangs 0 et 1. Pour tous les autres processus, la fin de hiérarchie a été rencontrée lors d'une précédente étape. Donc :

$$\texttt{newcomm}[3] = \texttt{MPI\_COMM\_NULL} \text{ pour chaque cœur de numéro } i \text{ avec } i \in [0, 1]$$

En ce qui concerne la création des communicateurs-maîtres (appel de `MPI_Comm_hsplit_with_roots` en lieu et place de `MPI_Comm_split_type`), moins seront créés et ils posséderont moins de processus membres :

— **étape 1 :** comme deux sous-communicateurs sont créés (un par cache L3), le communicateur-maître contient deux processus :

$$\texttt{rootscomm}[0] = \{P_0, P_4\}$$

— **étape 2 :** aucun communicateur-maître n'est créé pour le cache L3 numéro 1 et celui du cache L3 numéro 0 contient deux processus :

$$\texttt{rootscomm}[1] = \{\mathbf{P_0}, P_2\} \text{ pour le cache L3 numéro } 0$$
$$\texttt{rootscomm}[1] = \texttt{MPI\_COMM\_NULL} \text{ pour le cache L3 numéro } 1$$

À nouveau, nous vérifions que le processus-maître du communicateur-maître valide `rootscomm`[1] fait partie de `rootscomm`[0].

— **étape 3 :** seul le communicateur-maître correspondant au cache L1 numéro 0 peut être créé, car les processus de rangs 2 et 3 ne sont pas liés à des ressources plus «profondes» que le cache L1 du nœud. D'où :

$$\texttt{rootscomm}[2] = \{\mathbf{P_0}, P_1\} \text{ pour le cache L1 numéro } 0$$
$$\texttt{rootscomm}[2] = \texttt{MPI\_COMM\_NULL} \text{ pour le cache L1 numéro } 1$$

Le processus-maître de `rootscomm`[2] fait partie de `rootscomm`[1].

— **étape 4 :** le fond de la hiérarchie est atteint, deux communicateurs invalides sont alors créés durant cette étape, ainsi que déjà fait lors d'étapes précédentes :

$$\texttt{rootscomm}[3] = \texttt{MPI\_COMM\_NULL}$$
deux fois, pour les processus de rangs 0 et 1

Cet exemple montre la flexibilité de notre proposition qui est capable de gérer convenablement des cas «pathologiques». En effet, la politique de placement et d'affectation n'a aucune d'incidence sur notre approche.

4.2.4 Discussion sur l'approche proposée

Nous proposons d'utiliser des objets MPI pré-existants – les communicateurs – pour structurer les applications et permettre d'exploiter au mieux les ressources matérielles disponibles dans la plate-forme cible. Les communicateurs sont souvent employés dans les applications MPI et ils font partie du paysage de la programmation avec MPI de façon presque *naturelle* (au-delà de `MPI_COMM_WORLD`). Ainsi, utiliser notre proposition demanderait peu d'efforts, conceptuellement parlant. Les communicateurs créés ne sont pas nommés de façon prédéterminée par le standard lui-même : ainsi il n'y aura pas besoin de modifications pour ajouter de nouveaux noms ou bien retirer ceux devenus obsolètes. Selon nous, il n'est pas nécessaire de disposer de *noms standardisés*, mais plutôt de *moyens standardisés* pour accéder au panel des noms disponibles (et fournis via un logiciel de détection de topologie matérielle comme hwloc par exemple) et les manipuler. C'est pour des raisons similaires que nous ne dépendons pas d'une profondeur déterminée pour la hiérarchie. En créant *récursivement* de nouveaux sous-

communicateurs (mode *non-guidé*), les développeurs sont capables de récupérer tous les objets nécessaires successivement jusqu'à atteindre le bas de la hiérarchie matérielle.

Certains développeurs seront plus enclins à utiliser le mode *guidé* qui permet de créer des sous-communicateurs correspondants à un niveau spécifique de la hiérarchie matérielle. Cependant, le fait qu'un niveau puisse porter le même nom au sein de deux architectures distinctes tout en y désignant deux ressources différentes est un problème potentiel. De la même façon, il n'est pas garanti que le niveau choisi soit disponible ou bien que le nom puisse désigner de façon pérenne la même ressource au cours du temps [16]. Une autre utilisation possible de l'argument `info` pour le mode *non-guidé* serait de donner des informations quantitatives pour créer les sous-communicateurs plutôt que d'utiliser la structure du matériel. Il serait ainsi possible de produire des voisinages au sein desquels les performances lors des échanges inter-processus sont connues *a priori*. Introduire une *fonction de distance* avec plusieurs métriques (I/O, réseau, mémoire, etc.) est une alternative envisageable. Ces solutions, parce qu'elles suivent une approche quantitative, fonctionnent dans un contexte quelconque (i.e. pas nécessairement hiérarchique).

Avec notre solution, il est possible de créer *une* hiérarchie de communicateurs qui correspondent à des niveaux matériels (un partage des ressources du niveau plus exactement), mais cela n'est pas *la* hiérarchie complète. En effet, il nous semble inutile de fournir une vision complète et exhaustive car toutes les informations fournies ne seront pas utilisées ou utilisables. En ce sens, une implémentation qui ne fournirait que certains niveaux et de notre point de vue totalement acceptable, du moment que les caractéristiques des communicateurs soient conformes avec les propriétés données en section 4.2.2.1. La seule conséquence de ce manque de niveaux serait l'impossibilité d'optimiser complètement le code de l'application.

Une autre approche serait d'utiliser les *ensembles de processus* (cf. les sessions en section 4.1.3) car la création d'un communicateur est une opération collective (un consenus est nécessaire pour déterminer de façon univoque l'identificateur de contexte à utiliser pour le communicateur) tandis que les *psets* sont des objets locaux. Il est toujours possible de créer un groupe à partir d'un *pset* puis un communicateur à partir d'un groupe. C'est une piste que nous allons explorer dans le cadre du groupe de travail dédié aux sessions car elle permettrait d'alléger le fardeau de l'implémentation dans la création de communicateurs. Cependant, le mode *guidé* permet déjà d'opérer une sélection et de ne créer que le communicateur désiré.

Le principal reproche que nous pouvons formuler à l'encontre de notre solution est le fait que seules des architectures organisées hiérarchiquement sont gérées par le mode récursif *non-guidé*. Cependant, il faut tempérer cette critique par le fait que bon nombre d'architectures sont actuellement organisées ainsi et que cette tendance n'est pas remise en cause. Il est possible de trouver des machines (des nœuds) qui ne sont pas *totalement* hiérarchiques et dans ce cas, une partie de l'information de topologie physique sera probablement manquante. Il reste cependant le mode *guidé* qui ne formule aucune hypothèse sur le matériel. Également, il me semble préférable de disposer d'une information partielle plutôt que d'aucune information du tout.

Le problème est identique dans le cas des topologies réseaux non-hiérarchiques comme les tores par exemple. Est-ce que le partitionnement devrait être effectué selon une dimension particulière ? Le paramètre `info` pourrait être utilisé afin de communiquer cette information. Cependant, si une forme de hiérarchie est possible, alors notre solution sera en mesure de l'exploiter.

Enfin, un atout important de la solution proposée est qu'elle considère la topologie de façon globale et elle ne fait pas de distinction entre les niveaux inter-nœuds et intra-nœuds, ce qui est conceptuellement satisfaisant.

16. Les *Sockets* renommées en *Package* est un bon exemple.

4.2.5 Implémentations existantes

Une première présentation de la proposition d'extension de `MPI_Comm_split_type` a été faite en Février 2017 lors d'une réunion du Forum MPI à Portland (États-Unis). Afin de préparer cette réunion, nous avons implémenté une preuve de concept sous la forme d'une bibliothèque externe, appelée Hsplit (*Hardware Split*) qui nous a permis notamment de montrer les gains potentiels dans les cas de communications collectives et d'applications structurées hiérarchiquement (e.g. multiplication hiérarchique de matrices). Cependant, il existait déjà depuis 2014 dans Open MPI des extensions de la fonction `MPI_Comm_split_type`. MPICH propose également depuis 2018 des extensions dont le fonctionnement recouvre partiellement celui de Hsplit. Nous allons donc décrire ces implémentations en indiquant sur les similitudes et les différences par rapport à notre proposition.

4.2.5.1 La bibliothèque *Hardware Split* (Hsplit)

Nous avons commencé le développement de la bibliothèque Hsplit à la fin de l'année 2016 afin de préparer dans les meilleures conditions possibles la réunion du Forum MPI de Février 2017 à Portland. Initialement, nous devions faire une implémentation au sein d'une bibliothèque MPI puisque le cœur de la proposition concerne la définition de nouvelles valeurs de *split_type* pour la fonction `MPI_Comm_split_type`. Cependant, afin d'accélerer le développement nous avons choisi de créer une bibliothèque externe constituant une surcouche à MPI. Cette approche a pour inconvénient d'introduire un surcoût pour certaines fonctions. En effet, nous sommes obligés d'échanger des données entre processus, données qui nous seraient fournies nativement par le support exécutif de la bibliothèque MPI dans le cas d'une intégration complète. L'avantage est néanmoins que les développeurs qui sont contraints d'évoluer dans un environnement rigide de travail (compilateurs bien définis, version de MPI particulière, etc.) peuvent utiliser Hsplit à loisir. C'est notamment le cas de nos utilisateurs actuels du CERFACS (cf. section 4.2.6) qui sont obligés d'employer des outils Intel® et donc Intel®MPI. Si l'objectif est d'intégrer à court/moyen terme Hsplit dans Open MPI ou encore MPC, le maintien sous forme de bibliothèque externe nous paraît pertinent tant que les fonctionnalités de base de Hsplit ne sont pas encore acceptées dans le standard MPI.

Hsplit est véritablement indépendante de toute implémentation et n'utilise que des fonctions du standard MPI quand le passage de messages est requis (échanges d'informations, création de communicateurs, etc.). Toute la partie de découverte de la topologie matérielle repose quant à elle sur hwloc [CI7] pour la partie intra-nœud et netloc ([71], [24]) pour la partie inter-nœuds lorsque cela est possible (ou supporté).

Hsplit implémente donc la proposition détaillée en section 4.2.2, c'est-à-dire le support de la nouvelle valeur de *split_type* `MPI_COMM_TYPE_HW_SUBDOMAIN` avec les deux modes de fonctionnement déjà evoqués en section 4.2.2.1 :

— le mode **guidé** où un couple clef-valeur (*keyval*) est passé par l'argument `info` et permet de désigner le niveau particulier de la hiérarchie que l'on souhaite partitionner en sous-communicateurs. Les noms que nous utilisons alors sont dérivés de type hwloc pour les niveaux intra-nœuds et de la forme `Net_levelX` pour les niveaux inter-nœuds ;

— le mode **non-guidé**, c'est-à-dire sans utiliser de nombre ou de dénomination fixe pour les niveaux de la hiérarchie matérielle (c'est la structure du matériel, renvoyée par hwloc qui nous permet de faire le partitionnement). En cas de niveaux redondants, le niveau le plus bas dans la hiérarchie est utilisé pour la création du sous-communicateur résultat.

Conformément aux exemples détaillés en section 4.2.3, Hsplit gère le cas où l'assignation des processus sur les ressources matérielles est hétérogène. De plus, nous ne faisons pas l'hypo-

thèse que tous les nœuds de calcul sont structurés identiquement. Une description plus complète de l'interface ainsi que des évaluations de performances de Hsplit sont disponibles dans Jeannot *et al.* [CI4] et Goglin *et al.* [RI2].

Création des communicateurs topologiques Nous allons donner quelques détails concernant l'implémentation de cette fonctionnalité dans Hsplit. Quand la valeur `MPI_COMM_TYPE_HW_SUBDOMAIN` est donnée en paramètre à la fonction `MPI_Comm_split_type`, nous allons en réalité utiliser la fonction générique `MPI_Comm_split` pour effectuer le k-partitionnement en sous-communicateurs disjoints. Les deux paramètres importants pour réaliser cette opération sont la **couleur** (*color*) et la **clef** (*key*) [17] Pour la clef, nous allons utiliser le rang du processus appelant dans le communicateur-parent. Ainsi, l'ordre des processus est préservé dans le sous-communicateur résultat. Il reste à déterminer la couleur adéquate. Nous rappelons que si la couleur est définie à `MPI_UNDEFINED`, alors aucun communicateur valide ne sera créé pour le processus appelant (i.e. le résultat sera `MPI_COMM_NULL`). L'algorithme 2 montre comment cette couleur est déterminée :

Algorithm 2: Algorithme de choix de la couleur pour le partitionnement.

```
1 color ← MPI_UNDEFINED
2 obj = get_ancestor(comm); // Récupération de l'objet le plus
  profond contenant tous les bindings des processus
3 foreach idx← 0 .. (obj->arity) do // Recherche de la couleur adéquate
4     if calling_process->cpuset ⊆ obj->children[idx]->cpuset then
5         color ← idx
6         break
7 MPI_Comm_rank(comm,&rank)
8 MPI_Comm_split(comm,color,rank,newcomm)
```

Les étapes sont les suivantes :

— la première opération est de déterminer l'objet hwloc dans la hiérarchie qui comprend tous les processus membres du communicateur-parent à partitionner. hwloc propose une fonction permettant de trouver l'objet le plus profond (hiérarchiquement parlant) en cas de niveaux équivalents et redondants (ligne 2) ;

— une fois que cet objet est obtenu, nous pouvons chercher parmi ses «enfants» pour trouver celui auquel est lié le processus appelant (ligne 3). Si le *cpuset* [18] du processus appelant fait partie du cpuset de l'enfant considéré, c'est que le processus est lié à cet enfant (ligne 4). Les enfants sont numérotés *logiquement* et séquentiellement par hwloc et la couleur sera donc l'index de l'enfant trouvé (ligne 5). Si aucun objet n'est trouvé, la couleur reste fixée à la valeur `MPI_UNDEFINED`, ce qui produira un communicateur invalide (`MPI_COMM_NULL`) lors de l'appel à `MPI_Comm_split` ;

— comme indiqué précédemment, la clef est le rang du processus appelant dans le communicateur-parent (ligne 7) ;

— enfin, une opération de partitionnement est effectuée avec les bonnes couleur et clef (ligne 8).

17. cf. la fonction `MPI_Comm_split` dans le standard MPI.

18. Les *cpusets* constituent le moyen utilisé par hwloc pour décrire la localisation d'un objet (un ensemble de threads matériels inclus dans l'objet) ou l'affectation des processus (l'ensemble des threads matériels où les processus sont susceptibles de s'exécuter.

Si le communicateur produit est valide, trois couples *(key, value)* sont définis dont les valeurs peuvent être récupérées par un appel à la fonction `MPI_Comm_get_hw_subdomain_info` :

- `MPI_COMM_HW_SUBDOMAIN_TYPE` : la valeur correspondant à cette clef est un nom permettant de renseigner le développeur quant au type de ressource que représente le communicateur. En pratique, ce sont des noms dérivés de hwloc ainsi qu'expliqué en début de section, grâce à la fonction `hwloc_obj_type_snprintf`. Ainsi, un certain degré de portabilité est possible si d'autres implémentations MPI utilisent hwloc [19] ;
- `MPI_COMM_HW_SUBDOMAIN_NUM` : la valeur correspondant à cette clef est le nombre total de sous-communicateurs produits lors d'un partitionnement à un niveau donné de la hiérarchie [20] ;
- `MPI_COMM_HW_SUBDOMAIN_RANK` : la valeur correspondant à cette clef est le «rang» [21] d'un sous-communicateur parmi l'ensemble des sous-communicateurs créés lors d'un partitionnement à un niveau donné de la hiérarchie.

Les deux derniers couples *(key,value)* sont utilisables pour distribuer des données aux différentes parties de l'architecture matérielle. Toutes ces paires *(key,value)* sont stockées dans le communicateur à l'aide de la fonction `MPI_Comm_set_info`. De plus, les noms des clefs ne sont pas communiqués directement à l'utilisateur qui récupère les informations pertinentes à l'aide d'une fonction dédiée.

Création des communicateurs-maîtres Ainsi que nous l'avons expliqué en section 4.2.2, il est possible de créer une seconde hiérarchie de communicateurs qui vont faciliter les échanges éventuels entre les communicateurs créés lors du partitionnement d'un niveau matériel. Il est possible de procéder à cette création avec les fonctions existantes de MPI, mais il serait plus efficace qu'une telle opération soit effectuée par la bibliothèque MPI directement qui pourrait mettre à profit les informations de topologie matérielle fournies par l'élément logiciel idoine (hwloc ou autre). Là encore, il s'agit de trouver la bonne *couleur* à donner en argument d'une opération de type `MPI_Comm_split`, comme le décrit l'algorithme 3 :

- la première partie est identique à l'algorithme 2 (lignes 1 à 8) ;
- afin de créer le communicateur-maître, il faut trouver la couleur adéquate : seuls les processus possédant le rang 0 dans un sous-communicateur vont posséder une couleur valide (i.e. différente de `MPI_UNDEFINED`). Ce choix est dicté par le fait que dans un communicateur, les rangs des processus sont séquentiels et contigus. Un communicateur valide possède donc au moins un processus de rang 0 (lignes 10 et 11) ;
- si le processus appelant est maître de son communicateur, il doit fournir une couleur pour l'opération `MPI_Comm_split`. Comme dans le cas précédent, la clef sera son rang dans le communicateur-parent. La couleur choisie est l'index logique dans hwloc de l'ancêtre de l'objet trouvé pour créer le sous-communicateur (ligne 12). En effet, il est indispensable de faire une distinction entre les processus qui partagent le niveau *précédent* dans la hiérarchie. Cette information nous est donnée par cet index logique de l'ancêtre ;
- un opération de partitionnement est effectuée (ligne 13).

Les paires *(key,value)* définies précédemment sont également positionnées lors de cet appel.

Implémentation de `MPI_Comm_get_hw_subdomain_info` Cette fonction est triviale à implémenter car son rôle consiste juste à récupérer les informations associées aux sous-communi-

19. Ce qui est le cas d'Open MPI, de MPICH et MVAPICH.
20. Ce nombre correspond à la taille du communicateur-maître associé.
21. Par analogie avec le concept de rang pour les processus MPI.

Algorithm 3: Algorithme de choix de la couleur pour les partitionnements en sous-communicateurs et communicateur-maître.

```
1  color ← MPI_UNDEFINED
2  obj = get_ancestor(comm); // Récupération de l'objet le plus
   profond contenant tous les bindings des processus
3  foreach idx← 0 .. (obj->arity) do // Recherche de la couleur adéquate
4      if calling_process->cpuset ⊆ obj->children[idx]->cpuset then
5          color ← idx
6          break
7  MPI_Comm_rank(comm,&rank)
8  MPI_Comm_split(comm,color,rank,newcomm)
9  color ← MPI_UNDEFINED
10 MPI_Comm_rank(newcomm,&newrank)
11 if newrank = 0 then
12     color ← (obj->logical_index)
13 MPI_Comm_split(comm,color,rank,rootscomm)
```

cateurs et contenues dans les couples *(key, value)* détaillés ci-dessus. Ces informations sont transmises à l'utilisateur sous une forme exploitable. L'intégralité des informations étant stockée dans un objet MPI *info*, un appel à `MPI_Comm_get_info` suffit.

Nous avons fait le choix de ne pas utiliser les fonctions `MPI_Comm_set_name` et `MPI_Comm_get_name` pour stocker/récupérer la valeur de la clef `MPI_COMM_HW_SUBDOMAIN_TYPE` car il n'est pas possible d'avoir plusieurs noms par ce biais. De plus, le communicateur pourrait déjà posséder un nom qu'il serait fâcheux de remplacer.

Support des topologies réseaux avec netloc Nous nous sommes focalisés sur les niveaux intra-nœuds dans les paragraphes précédents. Comme nous souhaitons donner des moyens aux développeurs pour optimiser leur code en exploitant la *structure* du matériel pour améliorer la localité, la hiérarchie présente en-dehors des nœuds n'a pas à être traitée différemment de la hiérarchie intra-nœud. Dans le modèle de notre proposition, cette distinction est volontairement absente, pour favoriser un continuum topologique et hiérarchique. Le problème qui se pose à nous est alors de trouver des informations exploitables pour étendre l'implémentation de `MPI_Comm_split_type` vue précédemment. Il existe un panel de solutions pour trouver de telles informations, avec des outils spécifiques dédiés à une technologie réseau particulière ou encore en ayant recours à des mesures quantitatives pour en déduire une possible topologie matérielle. Dans notre cas, nous avons *naturellement* utilisé l'extension de hwloc dédiée : netloc [24]. Plus particulièrement c'est la fonction suivante qui nous permet de récupérer les informations souhaitées :

```
int netloc_get_network_coords(int *nlevels,int *dims,int *coords)
```

Avec :

- `nlevels` qui représente le nombre de niveaux de *switches* dans la hiérarchie réseau ;
- `dims[i]` qui représente le nombre de *switches* disponibles au niveau i de cette hiérarchie. `dims` est donc un tableau de taille `nlevels` ;
- `coords[i]` représente les coordonnées (le «rang») du processus appelant dans le niveau i de la hiérarchie réseau. `coords` est également un tableau de taille `nlevels`.

Le dernier niveau de la hiérarchie réseau n'est pas pris en compte car il ne s'agit que d'un numéro de port dans le dernier *switch*. Étant donné que tous les nœuds de calcul sont connectés à un *switch* via un port dédié (et donc avec un numéro de port dédié), cette information ne nous est pas utile et est redondante avec le niveau «Nœud» géré dans le cas intra-nœud.

4.2.5.2 Extension de `MPI_Comm_split_type` dans Open MPI

Open MPI propose une approche ressemblant fortement à celle des communicateurs nommés vus en section 4.2.1.1. Le standard MPI stipulant que les implémentations sont libres de définir leurs propres valeurs pour l'argument `split_type` de la fonction `MPI_Comm_split_type`, Open MPI ne s'en prive pas avec une douzaine de nouvelles valeurs définies telles que :

- `OMPI_COMM_TYPE_CLUSTER`
- `OMPI_COMM_TYPE_CU`
- `OMPI_COMM_TYPE_HOST`
- `OMPI_COMM_TYPE_BOARD`
- `OMPI_COMM_TYPE_NODE` (alias de `MPI_COMM_TYPE_SHARED`)
- `OMPI_COMM_TYPE_NUMA`
- `OMPI_COMM_TYPE_SOCKET`
- `OMPI_COMM_TYPE_L3CACHE`
- `OMPI_COMM_TYPE_L2CACHE`
- `OMPI_COMM_TYPE_L1CACHE`
- `OMPI_COMM_TYPE_CORE`
- `OMPI_COMM_TYPE_HWTHREAD`

Ainsi, il n'est possible que de créer des communicateurs correspondants à des niveaux intra-nœuds, et la hiérarchie réseau n'est pas disponible. Les noms donnés proviennent en réalité de l'outil de détection de topologie matérielle utilisé pour l'implémentation, hwloc en l'occurence. Cependant, cette approche présente les mêmes défauts que les communicateurs nommés (rédhibitoires selon moi) :

- l'utilisation de noms fixes pour désigner différents niveaux topologiques pose des problème de pérennité. L'arrivée de nouveaux niveaux ou matériels nécessite le rajout de nouveaux noms (par exemple, les GPU ne sont pas présents ici). De même, certains noms peuvent être amenés à tomber en désuétude, voire disparaître. Cette approche est sans doute encore acceptable pour une implémentation, mais pas une une interface standardisée comme MPI ;
- une solution serait de ne définir qu'un certain nombre de niveaux, mais le problème du choix de ces noms demeure entier : qu'est-ce qui justifie que tel nom sera pris et pas tel autre ? Les points de vue et les avis divergent fortement selon les architectures visées et trouver un consensus semble difficile ;
- les relations hiérarchiques entre niveaux ne sont pas identifiables à première vue (l'ordre dans lequel sont listés les noms semble indiquer la hiérarchie effective) : cette version de la fonction `MPI_Comm_split_type` implémente uniquement le mode *guidé* décrit dans notre proposition.

De plus, comme ces noms sont spécifiques à Open MPI, aucune portabilité ne saurait être garantie, ce qui peut constituer un frein sérieux à l'adoption d'une telle fonctionnalité avec une dépendance forte de l'application envers la bibliothèque MPI utilisée pour son développement. Enfin, cette implémentation de `MPI_Comm_split_type` n'est d'ailleurs plus maintenue et n'a pas connu de mise à jour depuis 2014 (par exemple, `OMPI_COMM_TYPE_SOCKET` devrait être remplacé par `OMPI_COMM_TYPE_PACKAGE`).

4.2.5.3 Extension de `MPI_Comm_split_type` dans MPICH

Plus récemment a été implémenté dans MPICH une version de `MPI_Comm_split_type` qui se rapproche très fortement de notre proposition. En effet, MPICH propose d'introduire une seule nouvelle valeur de `split_type` afin de réaliser un partitionnement topologique matériel du communicateur-parent (et qui n'est pas forcément `MPI_COMM_WORLD`, bien entendu). Cette nouvelle valeur a pour nom `MPIX_COMM_TYPE_NEIGHBORHOOD`. MPICH n'implémente comme Open MPI que le mode *guidé* et utilise deux clefs pour donner des informations sur la façon dont le partitionnement doit être effectué :

— la clef `SHMEM_INFO_KEY` indique qu'un partitionnement intra-nœud est demandé. Dans ce cas, les valeurs possibles pour cette clef sont des chaînes de caractères (des noms) qui sont obtenues à partir des noms des types hwloc correspondants. Par exemple, le nom `package` est dérivé du type hwloc `HWLOC_OBJ_PACKAGE`, `core` est dérivé de `HWLOC_OBJ_CORE`, etc. Il est possible de spécifier le partage de GPU avec le nom `gpu`, ce qui n'est pas encore possible dans Hsplit (ni dans Open MPI).

— la clef `NETWORK_INFO_KEY` indique quant à elle un partitionnement au niveau du réseau d'interconnexion. Des informations sur les *switches* sont obtenues grâce à netloc, l'extension dédiée de hwloc. Il est possible de créer des communicateurs selon la technologie d'interconnexion utilisée (`hfi`, `ib`, `eth`, etc.) ou bien selon une dimension donnée dans le cas d'une topologie toroïdale par exemple.

L'utilisation de deux clefs distinctes marque une forme de frontière entre les niveaux inter-nœuds et intra-nœuds, ce qui n'est pas le cas dans notre proposition [22]. Cette version de `MPI_Comm_split_type` est très proche conceptuellement de celle disponible dans Hsplit. Le mode *non-guidé* pourrait être intégré sans trop de difficultés et certaines fonctionnalités comme le partitionnement sur une dimension dans le cas de tores sont intéressantes et mériteraient d'être transférées dans Hsplit.

4.2.6 Une solution portable pour le placement dynamique d'applications hybrides ?

Hsplit est disponible librement sur le gitlab d'Inria [23]. Ce logiciel est actuellement utilisé pour effectuer un placement dynamique de processus MPI et de threads dans le cadre d'applications hybrides. Nous avons été contactés en 2018 par des membres du CERFACS qui cherchaient un moyen simple d'«isoler» certains processus vis-à-vis des ressources de calcul qu'ils utilisent, ce qui correspond exactement aux communicateurs topologiques. Nous avons donc entamé une collaboration fructueuse dont le résultat est le logiciel *Hybrid Interactive Placement of Processes in Palm and Oasis* (Hippo) [24].

4.2.6.1 Où l'on retrouve le problème du placement ...

Le problème que le CERFACS cherche à résoudre concerne le déploiement d'applications hybrides MPI + OpenMP avec du couplage de code : une application fait appel successivement à plusieurs noyaux de calcul qui possèdent chacun leur propre politique de placement et leur propre répartition entre processus MPI et threads OpenMP pour des performances optimales. Par exemple, le noyau numéro 1 fonctionnera optimalement en MPI pur tandis que le noyau numéro 2 fonctionnera optimalement avec 1 processus MPI par nœud et n threads OpenMP (en supposant que les nœuds possèdent n cœurs de calcul) et le noyau numéro 3 aura les

22. Hsplit fait une différence du point de vue de l'implémentation car les outils utilisés selon les niveaux diffèrent, mais conceptuellement, aucune différence ne devrait être faite.
23. https://gitlab.inria.fr/hsplit/hsplit
24. Oasis et PALM/OpenPALM sont deux logiciels développés par le CERFACS pour faire du couplage de codes.

meilleures performances avec 2 processus MPI par nœuds et $\frac{n}{2}$ threads OpenMP. Ces noyaux peuvent être exécutés séquentiellement ou parallèlement. Une contrainte forte vient de plus se rajouter : l'impossibilité de modifier le code de ces noyaux, qui sont fournis tels quels au CERFACS qui ne bénéficie d'aucun degré de liberté. Également, l'environnement de développement est constitué exclusivement des outils Intel®, que ce soit au niveau du compilateur ou des bibliothèques MPI et OpenMP et des versions spécifiques sont parfois employées sans possibilité de changement. Au final, les outils utilisés doivent offrir de fortes garanties de pérénité. On retrouve la plupart des problèmes propres aux applications hybrides ou composées, ainsi qu'évoqués en section 3.2.3.2.

En l'état actuel, il n'y a pas de solution véritablement satisfaisante à ce problème car la politique de placement et d'affectation des processus MPI est statique, déterminée via des paramètres passés à `mpiexec`. MPI ne fournit rien pour modifier ceci dynamiquement pendant l'application. Quant aux threads OpenMP, leur placement est contrôlé par la définition de variables d'environnement guidant le support exécutif d'OpenMP. Dès lors, la seule solution est de définir des politiques de placement et d'affectation de processus MPI et de threads OpenMP qui soient à la fois compatibles entre elles et susceptibles de convenir le mieux possible à l'ensemble des noyaux de calcul. Cependant, cette approche du «plus petit commun dénominateur» n'est pas très satisfaisante ni conceptuellement, ni en termes de performances. La figure 4.5 montre un exemple du fonctionnement recherché par le CERFACS avec trois

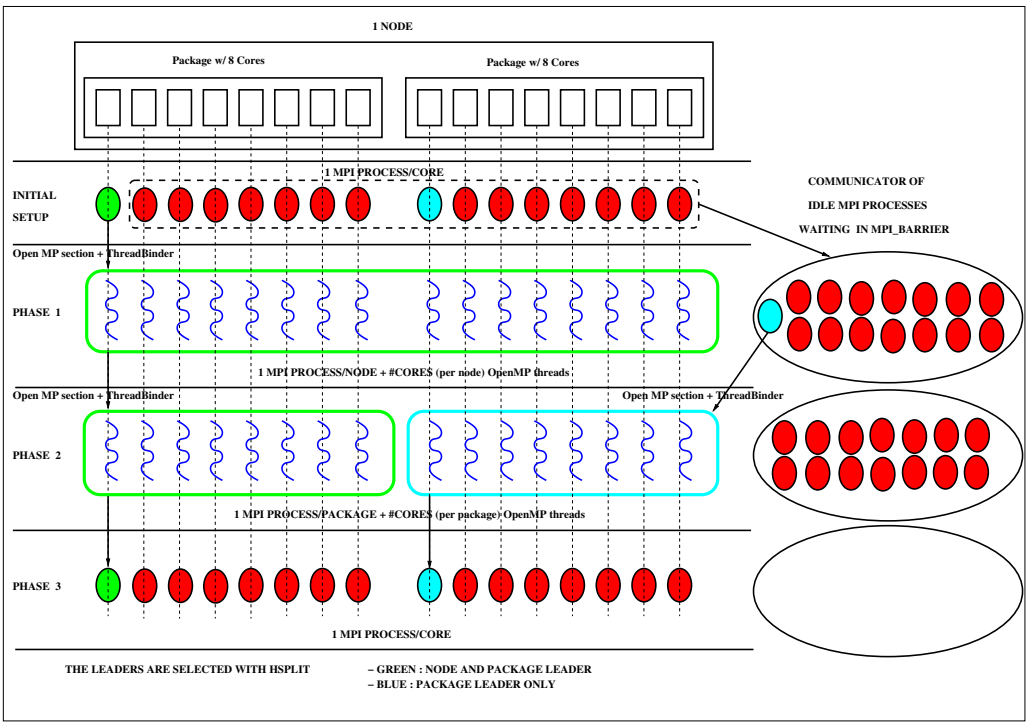


FIGURE 4.5 – Cas d'utilisation du CERFACS

phases distinctes (Full OpenMP, hybride, MPI pur).

4.2.6.2 Hsplit + hwloc = Hippo

La solution mise en place par le CERFACS reposait donc exclusivement sur un paramétrage spécifique des outils Intel®, et sur des calculs explicites de numéros de cœur pour obtenir quelque chose de moyennement satisfaisant au final. Les besoins du CERFACS étaient

partiellement couverts par notre proposition de communicateurs topologiques et son implémentation Hsplit : en effet, Hsplit ne permet pas de modifier l'affectation (*binding*) des threads, mais hwloc en revanche le peut. Nous avons donc utilisé hwloc pour couvrir le reste des besoins en termes de placement et de redistribution. Le résultat se concrétise sous la forme du logiciel Hippo, qui permet de mettre en place dynamiquement avant chaque noyau de calcul une politique de placement et d'affectation adaptée. De plus, cela est possible très simplement en paramétrant le code à l'aide de quelques mots-clefs au lieu de connaître les subtilités du paramétrage des supports exécutifs Intel®. Enfin, la solution implémentée n'est pas spécifique à une version de MPI/OpenMP particulière et donc utilisable dans un contexte bien plus générique.

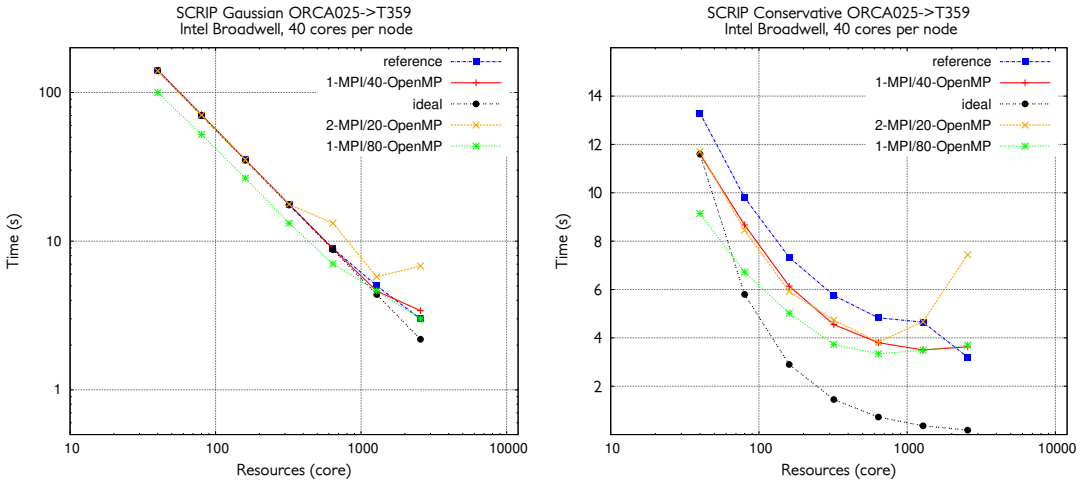
Pourquoi faire du placement ? La communauté des modélisateurs du climat est morcelée (physique des océans, physique de l'atmosphère, surfaces continentales, glaces de mers, glaciers continentaux, biogéochimie marine, etc.). Certains domaines partagent des modèles similaires, mais dans la plupart, on préfère développer et utiliser un outil propre. En particulier, la discrétisation spatiale choisie est celle qui procure le plus d'avantages. Par exemple, en océanographie, il est courant de construire une grille tripolaire pour éviter d'avoir une singularité au pôle Nord, on place deux pôles sur des points continentaux où il n'y a pas de calculs (sur le Canada et la Sibérie).

Tous les spécialistes de ces domaines doivent prescrire des conditions aux limites à leur modèle (par exemple les températures de l'océan en surface dans le cas d'un modèle d'atmosphère). Ces conditions sont construites à partir d'observations, ou de résultats d'autres modèles, et sont lues depuis des fichiers. Cependant, il est courant de vouloir les faire interagir avec les valeurs de surface des modèles (par exemple, la couche limite atmosphérique modifie les températures de surface de l'océan et vice versa). Un coupleur est dès lors indispensable et de plus il doit être capable d'échanger pendant la simulation des quantités discrétisées sur des maillages différents. C'est ce qui justifie l'inclusion de la librairie d'interpolation SCRIP dans le coupleur OASIS du CERFACS.

Si on rentre dans des considération d'efficacité des calculs, on s'aperçoit que ces calculs hétérogènes, car issus de modèles différents qui tournent en même temps, doivent être synchronisés pour que les plus rapides n'attendent pas trop longtemps les plus lents. C'est toute la problématique de l'équilibrage de charge que le CERFACS étudie depuis quelques temps. C'est dans cette problématique qu'intervient le placement, car des placements différents induisent des déséquilibres. C'est donc pour cela que Hippo (hwloc et Hsplit) permet de répartir finement sur les nœuds des supercalculateurs les différentes tâches des modèles couplés.

Résulats préliminaires avec SCRIP La figure 4.6 montre des résultats préliminaires obtenus pour l'application SCRIP [25] de calcul d'interpolation sphérique, intégrée dans le coupleur OASIS. Au début d'une simulation, OASIS utilise SCRIP pour calculer les poids et adresses des interpolations entre maillages du système couplé, puis les utilise dans une boucle temporelle pour faire passer les variables d'un maillage à l'autre. Il existe une version parallèle de SCRIP [160]. Les courbes bleues montrent les performances de la version initiale (sans placement dynamique) et les autres courbes montrent les peformances obtenues avec un placement dynamique (possible grâce à Hsplit et Hippo) et selon des ratios processus MPI/threads OpenMP différents. Le surcoût lié à l'utilisation de Hippo est faible, plus faible actuellement que celui de Libquo [79]. Les résultats sont donc encourageants et le travail doit être poursuivi pour offrir encore plus de souplesse. Par exemple, au lieu de paramétrer Hippo avec des

25. https://github.com/SCRIP-Project/SCRIP



(a) Interpolation Gaussienne (b) Interpolation conservative de 1^{er} ordre

FIGURE 4.6 – Temps d'exécution de l'interpolateur SCRIP pour la phase «adresse et poids» (*Courtoisie d'Éric Maisonnave et Andrea Piacentini du CERFACS*).

mots-clefs désignant des ressources physiques, il pourrait être intéressant d'indiquer le ratio processus MPI/threads OpenMP et laisser Hippo décider seul du niveau topologique compatible avec la répartition demandée à l'aide du mode *non-guidé* propre aux communicateurs topologiques.

Hippo vs. Libquo Nous avons indiqué précédemment qu'il n'existait pas vraiment de solution satisfaisante au problème de la mise en place de politiques dynamiques de placement, d'où la création de Hippo. Il existe cependant un projet qui a été créé avec des objectifs très similaires : libquo [79]. Tout comme Hippo, libquo repose à la fois sur MPI et hwloc et est disponible sous la forme d'une bibliothèque externe. Libquo propose d'«empiler» des politiques de placement à l'aide d'une opération de type *push* et de les mettre en place avec une opération de type *pop*. Il est ainsi possible de modifier le placement dynamiquement selon le noyau de calcul à exécuter. Cependant, le surcoût de libquo est assez important (de l'ordre de 8%), et de plus, libquo se révèle incapable de gérer finement la sélection et le placement des processus MPI à «désactiver» (cf. figure 4.5). Un examen rapide du code de libquo montre que la fonctionnalité manquante est justement de faire un partitionnement topologique, ce que propose Hsplit et dont Hippo fait usage. Nous sommes donc en discussions avec les créateurs de libquo afin d'examiner la possibilité d'introduire Hsplit dans leur travail. L'existence en parallèle de Hsplit et libquo n'est pas un problème en soi car libquo a été créé pour répondre à des besoins d'applications particulières tandis que Hippo vise les applications hybrides couplées de façon plus générale [26].

4.2.6.3 Autres utilisateurs potentiels de Hippo

Hippo couvre un besoin assez répandu, un peu comme hwloc en son temps. Hippo est un sérieux candidat au remplacement de solutions *ad-hoc* déjà evoquées comme PlaceMe [85], [40], [114] ou [150]. De plus, nos utilisateurs actuels du CERFACS ont commencé à contacter

26. Dans le domaine de la climatologie pour le moment, mais rien n'empêche des applications d'autres domaines d'en faire usage.

des équipes et centres de calcul qui se sont montrés intéressés par Hippo. Ainsi, nous pouvons envisager une utilisation/intégration de Hippo :

- au CERFACS : chez les développeurs et utilisateurs des coupleurs OpenPALM et OASIS. Les applications visées sont dans le domaine de chimie atmosphérique ;
- chez Météo France (partenaire du CERFACS) : pour des applications de chimie atmosphérique ;
- au *European Centre for Medium-Range Weather Forecasts* (ECMWF) pour des applications de couplage et d'assimilation océanique ;
- au MetOffice (Royaume-Uni) pour la parallélisation du code d'océanographie NEMO ;
- au CEA pour du couplage de codes de simulations climatiques et pour une implémentation hybride de la bibliothèque d'E/S et d'interpolation XIOS [27] ;
- à l'Institut Pierre Simon Laplace pour le lancement de modèles couplés hybrides. Hippo remplaçerait [114] ;
- à l'IDRIS pour le lancement d'applications hybrides (remplacement de [40]) ;
- au *Joint Center for Satellite Data Assimilation* américain pour l'intégration de codes dans le contexte OOPS [28] ;
- à l'*University Corporation for Atmospheric Research* ;
- au *National Energy Research Scientific Computing Center* du *Lawrence Berkeley National Laboratory* pour remplacer [150] ;
- à l'*Argonne National Laboratory* pour des codes de couplage ;
- au *Barcelona Supercomputing Center* pour des applications du département *Computational Earth Science*.

La liste est longue est promet de potentielles collaborations fructueuses. Notre objectif est dans un premier temps de passer du stade de prototype à un outil plus mature. Les retours et commentaires des utilisateurs seront donc très importants pour comprendre les éventuels besoins que Hippo ne couvrirait pas à l'heure actuelle.

4.3 Le groupe de travail *Hardware Topologies* du MPI Forum

Je représente Inria au sein du Forum MPI depuis 2010. Le Forum MPI est l'instance de discussion et de standardisation de l'interface MPI. Les discussions actuellement en cours concernent la mise au point de la version 4.0 de MPI, ainsi que l'ajout de fonctionnalités dans les futures révisions du standard. Le Forum MPI est composé de laboratoires de recherche, d'universités et d'industriels ayant des intérêts dans le HPC en général et le passage de messages en particulier. Par exemple, la plupart des constructeurs de supercalculateurs et de machines parallèles participent au Forum car la fourniture d'une bibliothèque MPI avec leur matériel est quasiment obligatoire pour eux. Le Forum est ouvert est n'importe quelle institution peut y participer. Des réunions physiques sont tenues tous les trois mois afin de procéder à des présentations de propositions d'extensions et de rajouts au standard. Ces réunions sont l'occasion également pour les institutions présentes de voter sur les orientations à prendre pour le standard MPI. Outre les réunions et séances plénières, le Forum est l'occasion pour différents groupes de travail de se réunir physiquement pour discuter des actions et propositions à faire plus largement devant le Forum tout entier. Tous ces groupes de travail suivent une organisation et un fonctionnement qui leur sont propres. La création d'un groupe de travail doit être soutenue par un minimum de quatre institutions et est entérinée par un vote au sein du Forum.

27. https://www.esiwace.eu/services/support/copy_of_overview
28. https://www.da.ucar.edu/sites/default/files/TremoletNextGenOOPS.pdf

4.3.1 Création d'un groupe de travail dédié aux topologie matérielles

Notre objectif étant d'introduire dans MPI de nouveaux mécanismes permettant aux applications de pouvoir comprendre et exploiter au mieux la plate-forme physique sous-jacente, nous devons nous conformer au processus de standardisation propre à MPI qui peut être long et parfois aléatoire. Nous avons donc fait une première présentation concernant les communicateurs topologiques lors de la réunion de Février 2017 à Portland. Nous avons ensuite popularisé l'idée en discutant avec des membres actifs du Forum et qui assistent très régulièrement aux réunions physiques. Nous avons alors émis l'idée de créer un nouveau groupe de travail dédié aux topologies matérielles dans MPI (*Hardware Topologies Working Group*). Les travaux menés depuis plusieurs années dans Runtime puis TADaaM consacrés au placement de processus, les développements de TREEMATCH et surtout hwloc nous ont donné la crédibilité nécessaire pour demander cette création et la légitimité pour en assurer l'animation. Je suis à l'heure actuelle le responsable et le principal animateur de ce groupe de travail international qui regroupe plus d'une trentaine de membres. Je suis amené à faire des comptes-rendus réguliers au Forum concernant l'état d'avancement des travaux et discussions menés au sein du groupe de travail. Plus globalement, ce dernier permet également un gain en visibilité pour notre équipe et ses travaux.

Alors que les modèles hybrides de programmation gagnent en importance et se sont peu à peu imposés dans le paysage du HPC, le standard MPI n'évoque que très peu les interactions avec les autres modèles qui pourraient être amenés à cohabiter avec MPI. Or, il me semble qu'il s'agit là d'un point très important. En effet, il est impératif d'anticiper les conséquences de l'hybridation sur le comportement des fonctions. De plus, si la communauté MPI se penche sur le support des topologies matérielles, cela signifie que des réflexions du même acabit sont probablement menées au sein des instances de discussion et de standardisation d'autres paradigmes et modèles de programmation. Les tentatives de contact effectuées avec la communauté OpenMP pour tenter de mener des réflexions en vue de proposer des mécanismes compatibles entre modèles se sont pour le moment soldées par un échec. Il me semble néanmoins capital d'arriver à entamer des discussions afin de ne pas proposer des mécanismes interagissant mal, voire incompatibles entre eux.

4.3.2 Organisation et objectifs du groupe de travail

L'organisation du groupe de travail repose à la fois sur les réunions physiques du Forum MPI tous les trois mois et sur des audio- ou téléconférences plus nombreuses (environ toutes les trois semaines en moyenne). Les comptes-rendus des discussions et d'autres documents pertinents sont disponibles sur le wiki du groupe de travail (`https://github.com/mpiwg-hw-topology/hw-topology-issues/wiki`). Une liste de diffusion permet également les discussions si besoin est. Trois axes de discussion ont été rapidement identifiés :

— l'approche *implicite* : cette approche se pose la question de l'utilisation de la topologie matérielle dans les applications MPI sans faire explicitement usage de détails matériels et surtout sans avoir besoin d'une description exhaustive et détaillée de la plate-forme sous-jacente. Le mode *non-guidé* des communicateurs topologiques est un bon exemple de cette approche : le développeur peut exploiter la topologie matérielle via des communicateurs, avec une relation hiérarchique entre eux et n'a pas besoin de plus de détails pour structurer son application.

— l'approche *explicite* : cette approche se pose la question des outils et fonctions qu'il faudrait mettre en place pour fournir à l'utilisateur une cartographie fidèle mais exploitable de la topologie matérielle. Quelles sont les bonnes abstractions et objets à utiliser ? Est-ce qu'il y a besoin d'en introduire de nouveaux ou bien est-ce que l'existant dans MPI

suffit ?

— la gestion du déploiement, du placement et de l'affectation des processus MPI sur les unités physiques d'exécution de la plate-forme cible. Serait-il pertinent de standardiser des arguments pour `mpiexec/mpirun` afin de garantir une certaine portabilité des options relatives aux topologies matérielles dans les gestionnaires de processus/support exécutifs des bibliothèques MPI ?

Les deux approches implicite/explicite sont complémentaires et destinées à des publics différents qui semblent d'ailleurs être disjoints. Lors des discussions que nous avons eues jusqu'à présent dans le groupe de travail mais également en dehors avec des développeurs d'applications intéressés par l'avancée de ces travaux, il apparaît que certains souhaitent des informations précises (explicites pourrait-on dire) de façon à avoir un contrôle assez fin sur l'organisation et la structure de l'application. D'autres ne souhaitent pas entrer dans les détails mais veulent néanmoins prendre en compte la topologie matérielle dans leur application. Nous retrouvons quelque peu la même différence d'utilisation que dans les modes *guidé* et *non-guidé* des communicateurs topologiques.

Le troisième point, concernant la standardisation d'options pour les commandes `mpiexec/mpirun` est plus délicat. Nous pensons que la mise en application d'une politique de déploiement et de placement devient si problématique (cf. section 3.2.2) que des discussions devraient *a minima* avoir lieu au sein du Forum. Cependant, je ne suis pas des plus optimiste quant à l'issue et il fort probable que des initiatives diverses et variées voient le jour, de façon plus ou moins coordonnées, pour répondre au problème. Il faut cependant noter l'initiative du CEA (membre fondateur du groupe de travail) qui a mis en place un groupe de travail réunissant plusieurs équipes et laboratoires intéressés par la problématique du déploiement et du placement d'applications parallèles. Le contexte envisagé est plus large que les applications MPI pures ou hybrides. Nous participons aux discussions, car hwloc mais aussi Hippo sont concernés au premier chef.

Enfin, il est important de comprendre que notre objectif n'est absolument pas d'introduire hwloc dans MPI. En effet, les objectifs de ces deux bibliothèques sont totalement différents : récolte d'informations d'un côté et communications de l'autre. De plus, le nombre très important de fonctions dans hwloc ferait littéralement exploser le nombre de fonctions dans MPI, ce qui n'est pas des plus souhaitable. Enfin, l'interface d'hwloc n'est sans doute pas compatible avec les contraintes imposées par Fortran, contraintes qui pèsent parfois lorsque des propositions sont formulées au Forum MPI.

4.3.3 Propositions du groupe de travail

Le groupe de travail dédié aux topologies matérielles existe maintenant depuis un peu plus d'une année et demie et les travaux suivent un rythme assez soutenu. Nous avons actuellement deux propositions qui sont en cours de discussion au niveau du Forum MPI :

— d'une part la proposition d'extension de `MPI_Comm_split_type` afin de créer des communicateurs basés sur la localité exprimée comme un partage de ressources matérielles entre processus MPI. Cette proposition n'est qu'une étape et par la suite nous souhaitons proposer des mécanismes complémentaires : d'une part des fonction permettant d'interagir avec le logiciel de détection de topologie présent dans la bibliothèque MPI afin de récupérer des noms de niveaux topologiques, comme évoqué en section 4.2.4. Ainsi, il serait facile de pouvoir déterminer le niveau atteint dans le mode *non-guidé* et de connaître les niveaux effectivement utilisables dans le cas du mode *guidé*. Hsplit implémente déjà de telles fonctions mais l'interface a encore besoin d'être discutée au niveau du Forum ;

— d'autre part une proposition de création de topologies virtuelles cartésiennes prenant en compte la topologie de la plate-forme cible [152]. Cette proposition vise spécifiquement des machines possédant des nœuds homogènes hiérarchiques. Le schéma de communication applicatif est bien entendu pris en compte et on cherche à préserver localité des communications (par exemple, réduire les communications inter-nœuds au profit des communications intra-nœuds). Cette seconde proposition utilise d'ailleurs les communicateurs topologiques que nous souhaitons voir standardisés.

4.4 Conclusion

Dans ce chapitre, nous avons montré comment MPI a su évolué au fil du temps pour s'adapter aux changements réguliers survenant du côtés des architectures et topologies matérielles. nous avons également montré qu'il existe une marge de manœuvre importante pour introduire de nouveaux mécanismes dont nous sommes convaincus qu'ils apporteraient d'intéressants bénéfices s'ils étaient adoptés dans le standard. Nous n'avons décrit que le principe de base de l'interface que nous souhaiterions proposer. En effet, les discussions sont encore en cours et certaines fonctionnalités connexes ne sont pas totalement définies, aussi bien en termes de services que d'interface. C'est en particulier le cas de la gestion des noms des niveaux topologiques, qui a fait l'objet de beaucoup de discussions au sein du Forum. Les nombreuses activités et discussions montrent que la communauté MPI est à l'écoute et prend en compte les changements pour améliorer le standard. Cela explique sans doute pourquoi MPI est annoncé comme «mort» depuis des années mais toujours massivement utilisé dans le calcul scientifique, car les solutions alternatives à MPI venant du HPC peinent à s'imposer.

Nous avons commencé à ouvrir des pistes pour la gestion des topologies matérielles dans MPI mais le chantier est d'ampleur. Parallèlement, nous continuons les développements autour de Hsplit afin de préparer de futures propositions concernant l'approche explicite cette fois (cf. section 4.3.2). De plus les nombreux contacts établis dans le cadre des travaux sur Hippo nous confortent dans l'idée que le déploiement et le placement d'applications parallèles hybrides est une problématique importante. Afin de ne pas avancer en ordre dispersé, nous allons examiner comment synchroniser les développements autour de Hippo avec ceux de Libquo et surtout avec les discussions menées actuellement au sein du groupe de travail dédié à cette question et animé par le CEA.

Si l'inertie du Forum permet de protéger MPI de certains effets de mode, elle entraîne son lot de difficultés dont il faut bien avoir conscience. Mais participer au Forum MPI permet à l'équipe de gagner en visibilité, de nouer des contacts, voire d'initier des collaborations. Cela permet également de promouvoir les solutions et logiciels conçus et développés au sein de l'équipe TADaaM.

Chapitre 5
Conclusion générale

Dans ce document, nous avons tenté de dresser un panorama des recherches menées dans le domaine de la prise en compte et de la gestion des topologies physiques au sein du modèle de programmation basé sur le passage de messages dans sa forme pure ou hybride (i.e. en coexistence avec un autre modèle, comme les communications implicites avec du multithreading). Nous avons replacé nos travaux dans le contexte afin de montrer leur impact, leur influence ainsi que leur utilisation dans l'univers du calcul hautes-performances. L'ensemble des travaux présentés ici se situe dans la continuation de ceux effectués durant ma thèse car déjà à l'époque, la question de la gestion des topologies matérielles était au cœur de mes préoccupations (dans un contexte toutefois différent). J'ai poursuivi dans cette direction avec cohérence et continuité afin de proposer des solutions pertinentes, instanciées sous forme de logiciels concrètement utilisables et plus aboutis que des prototypes. C'est en effet un aspect qui me semble déterminant pour la diffusion des idées, car cela permet non seulement de gagner une éventuelle base d'utilisateurs dont les retours sont très utiles à l'amélioration des travaux, mais également d'établir une certaine crédibilité nécessaire lors des discutions au sein des instances de standardisation du monde du HPC.

5.1 Bilan des travaux présentés

Lors de mes travaux, je me suis avant tout consacré aux problématiques liées au passage de messages grâce à l'expérience que j'avais acquise lors de ma thèse : en effet, implémenter MPI n'est pas une chose facile, encore moins lorsque cette dernière repose sur le multithreading [1] pour la progression des communications. Le défi posé par les architectures multicœurs hiérarchiques était d'envergure pour la nouvelle génération d'implémentations et participer à le relever a été à la fois très enrichissant sur le plan scientifique mais aussi source de grande satisfaction puisque les résultats produits ont eu un impact notable pour la communauté. Progressivement, j'ai décidé de m'intéresser non pas juste aux implémentations de MPI proprement dites, en élargissant la cible de me travaux à des éléments connexes de l'écosystème HPC (e.g. gestionnaires de ressources et de processus) tout en apportant des solutions généralisables à d'autres paradigmes et modèles de programmation (e.g. le placement de processus peut s'appliquer à d'autres entités et dans d'autres contextes). Enfin, j'ai pris conscience que c'est au niveau des standards eux-mêmes qu'il faut intervenir afin que les idées et les méthodes puissent être le mieux acceptées et surtout le plus utilisées. Ces trois phases successives [2] dans les travaux montrent une évolution cohérente des thèmes et de l'approche choisis.

Les travaux réalisés couvrent une partie importante de la pile HPC ce qui permet d'avoir une meilleure vision d'ensemble et de mieux appréhender les problèmes posés par la cohabitation de multiples éléments au sein de cette pile. De plus, ce travaux ont été mis à disposition de la communauté sous la forme de logiciels qui dépassent la simple preuve de concept.

1. Un support correct du niveau `MPI_THREAD_MULTIPLE` est toujours à l'heure actuelle une chose non triviale pour les implémentations de MPI ...
2. Avec parfois du recouvrement entre elles.

Il s'agit là encore d'un point auquel j'attache beaucoup d'importance. En effet, il me semble que la distribution d'un logiciel doit faire preuve d'attention car le logiciel joue un rôle de vitrine permettant de convaincre (ou pas) l'utilisateur. Cet aspect n'est pas toujours assez pris en considération et je pense que les doctorants devraient y être davantage sensibilisés afin de mieux valoriser leur travaux de thèse.

Enfin, et c'est peut-être le plus important selon moi, ces travaux ne sont pas étrangers à une certaine prise de conscience dans l'équipe (Runtime puis surtout TADaaM). En effet, si nous avons cherché à aborder le problème de la localité de plusieurs façons (ainsi que nous l'avons vu pour le placement par exemple) c'est qu'il a y diversité dans les approches possibles et différents emplacements dans le continuum des couches logicielles constituant l'écosystème HPC où ces travaux sont susceptibles de trouver leur place. Par ailleurs, un nombre important d'équipes travaillant sur ce sujet se limitent souvent à des solutions ciblant une partie bien spécfique de la pile logicielle. L'ensemble des travaux que nous avons menés nous a permis de mieux réfléchir à leur imbrication et leurs interactions. Nous sommes convaincus que seule une approche holistique permettra l'émergence de solutions satisfaisantes. Mais tout cela repose sur une nécessaire amélioration des échanges d'informations entre tous les éléments constitutifs de l'écosystème HPC. Ce point est même devenu central parmi les axes de recherche explorés dans/par l'EPI TADAaaM.

5.2 Visibilité et impact des travaux réalisés

Les travaux présentés dans ce document sont pour la plupart devenu des références dans le domaine et sont fréquemment cités dans des articles. Les logiciels que j'ai développés ou bien auxquels j'ai contribué sont diffusés et employés par une base relativement large d'utilisateurs, sauf pour les plus récents bien évidemment. J'ai également été régulièrement sollicité pour relire des articles de conférences ou de journaux sur le thème de la localité, du placement et de la gestion des topologies matérielles dans les applications HPC (MPI plus particulièrement). TADaaM est ainsi devenue une équipe regroupant des compétences reconnues internationalement pour tout ce qui concerne ces aspects et les problématiques ayant trait à la localité.

5.2.1 MPICH2-NEMESIS : une implémentation MPI de référence

MPICH2-NEMESIS est une réalisation dont je tire une certaine fierté : il est en effet très satisfaisant d'avoir architecturé et développé un logiciel qui aura été utilisé par une très grande base d'utilisateurs et ce pendant une période de temps relativement importante (environ une décennie). Cette longévité s'explique d'une part par les performances de tout premier plan de MPICH2-NEMESIS mais également par son évolutivité qui nous a permis d'y introduire des optimisations et améliorations sans pour autant renier les choix initiaux de conception. Également, le fait que des implémentations propriétaires dérivées de MPICH, et non des moindres, aient choisi d'utiliser NEMESIS comme leur support de communication bas-niveau est pour moi une forme de reconnaissance implicite quant à la qualité des résultats et la pertinence des choix effectués pour le développement de NEMESIS. Les articles concernant notre étude sur les communications par mémoire partagée et sur l'architecture et la mise en œuvre de NEMESIS ont été cités plusieurs centaines de fois (en cumulé) ce qui montre bien la place et l'intérêt de ce logiciel. Le développement de travaux indépendants (e.g. module réseau pour Cray Gemini/Aries, module réseau pour Infiniband, etc.) montre bien également que d'autres équipes se sont appropriées NEMESIS en tant que véhicule d'expérimentation et de recherche ce qui constitue une nouvelle fois une forme de reconnaissance quant au travail accompli. Néanmoins, je regrette un peu le déficit de notoriété des *auteurs* de NEMESIS (i.e. Darius BUNTINAS

et moi-même).

5.2.2 hwloc et sa concurrence : combien de divisions ?

Un autre objet de satisfaction est le succès incontestable du logiciel hwloc. Dès le début de mes travaux sur le placement, il était clair pour moi que la possibilité de récupérer de façon dynamique et précise des informations (quantitatives mais surtout qualitatives) sur la topologie matérielle sous-jacente était un élément fondamental pour proposer un environnement de déploiement applicatif à la fois générique et ergonomique. hwloc et son intégration dans de multiples logiciels de l'écosystème HPC montrent qu'il est possible de se retrouver dans une position dominante sans passer par des efforts de standardisation et de synchronisation avec des solutions concurrentes. Néanmoins, je pense que ce succès fulgurant ne saurait être érigé en modèle et que des circonstances particulières expliquent cette situation de quasi-monopole de fait de hwloc. Au final, la place qu'hwloc occupe actuellement montre que l'intuition de départ quant aux besoins était fondée et que son développement et sa diffusion sont arrivés juste à propos (et à temps). La notoriété d'hwloc est impressionnante [3] et permet à l'équipe d'étayer sa crédibilité lors de discussions dans les instances de standardisation pour ce qui touche à la localité et aux topologies matérielles.

5.2.3 TreeMatch : une référence pour le calcul de placement

Si hwloc est un élément-clef de l'environnement permettant la mise en place effective de politiques de placement intelligentes c'est-à-dire prenant en compte des critères sophistiqués tels que le schéma applicatif, son véritable cœur est l'algorithme TreeMatch. La notoriété de TreeMatch est certes moindre que celle d'hwloc mais son cadre d'utilisation est également plus restreint. Il n'en demeure pas moins que les concurrents de TreeMatch sont nombreux et que cet algorithme est très largement référencé dans la littérature (plusieurs centaines de citations en cumulé pour l'ensemble des articles traitant de TreeMatch et de son intégration dans des logiciels de l'écosystème HPC). TreeMatch est devenu une référence dans le domaine du placement pour grappes de nœuds multicœurs hiérarchiques complexes et les solutions concurrentes se comparent quasi-systématiquement à lui (cf. la liste en section 3.3.3.1). De plus, nous avons dès le début souhaité intégrer TreeMatch dans d'autres logiciels afin d'en montrer le potentiel et d'en assurer une diffusion plus large. C'est ainsi que nous avons pu proposer une implémentation non-triviale de `MPI_Dist_graph_create` dans Open MPI qui montre notamment le potentiel des fonctions topologies virtuelles de MPI, encore sous-utilisées dans les applications à l'heure actuelle.

5.2.4 Le pari Hsplit/Hippo

Une partie du travail effectué au sein du groupe de travail du Forum MPI et dédié aux topologies matérielles s'est concrétisé sous la forme de la bibliothèque Hsplit qui est maintenant connue de la plupart des membres du Forum MPI. L'utilisation de Hsplit et des communicateurs topologiques de façon générale peut se faire à deux niveaux : pour la structuration d'applications parallèles proprement dites (e.g. une multiplication hiérarchique de matrices) ou bien pour le déploiement de ces mêmes applications. En effet, pour un déploiement d'applications couplées hybrides, il est nécessaire de pouvoir sélectionner les processus qui vont créer des threads et les processus qui vont rester en sommeil durant l'exécution d'une telle séquence (cf. figure4.5). C'est typiquement ce que permet de faire Hsplit et comme le *binding* dynamique

3. Le premier article qui lui est consacré a été cité plusieurs centaines de fois.

de processus/threads n'est pas pris en charge par les standards actuels de programmation, il faut alors utiliser un standard *de facto* comme hwloc qui permet de faire ce travail. Cette «glue» logicielle, développée avec le CERFACS et appelée Hippo, permet de résoudre concrètement les problèmes que se posent un certain nombre d'équipes qui développent des applications couplées hybrides. Nos contacts au CERFACS travaillent plus particulièrement dans le cadre de la climatologie et ont commencé à diffuser Hippo au sein de cette communauté. Nous nous aperçevons que Hippo reçoit un accueil très favorable et nous souhaitons donc être en capacité d'offrir un outil pratique et générique, un peu comme hwloc en son temps. Bien évidemment, Hippo concerne davantage les utilisateurs finaux et non pas les développeurs de la pile HPC, ce qui fait que son adoption devrait être plus réduite. Nous allons cependant essayer de tester d'autres communautés par le biais de des concepteurs de libquo, qui est une solution «concurrente» à Hippo. Les contacts déjà pris sont encourageants et Hsplit devrait être intégré dans libquo. Il n'est pas à exclure que les deux projets fusionnent pour aboutir à la création d'une unique solution générique de placement d'applications couplées hybrides. Les besoins étant clairement établis, le développement dans l'équipe TADaaM d'un nouveau logiciel de référence dans le domaine du placement et de la localité serait un nouveau signal fort émis vers la communauté HPC. Ainsi, s'il est encore un peu trop tôt pour évaluer l'impact de Hsplit/Hippo, le potentiel est très important et les efforts doivent continuer pour le développement et la promotion de cette solution.

5.3 Perspectives

Les travaux que nous avons présentés sont pour certains achevés mais pour d'autres en cours de développement. Des perspectives peuvent donc être établies en relation avec la continuation de ces travaux. Également, des réflexions à plus long terme sont envisageables, si possible en capitalisant sur les enseignements retenus suite aux recherches menées. Ces réflexions concernent le futur des architectures et la façon de les programmer, ce qui conduit naturellement à se poser la question de la place de MPI. En effet, nous avons montré dans ce document comment ce paradigme et ses mises en œuvre ont réussi leur évolution pour demeurer une référence dans le HPC.

5.3.1 Amélioration et extension des travaux actuels

Les travaux décrits dans ce document et dans les articles associés sont susceptibles d'être poursuivis en vue de leur amélioration ou bien pour gérer de nouveaux cas ou applications. Nous donnons ci-dessous un ensemble de pistes à explorer pour ce faire.

5.3.1.1 Étudier le lien entre informations et gains

Les perspectives à court terme sont nombreuses car il subsiste toujours un certain nombre d'éléments à étudier et notamment pour comprendre plus précisement d'où proviennent les gains de performances induits par la mise en place d'une politique de placement prenant en compte l'affinité des processus applicatifs. Il y a en particulier un point qui me semble très important et qui concerne les informations de schéma applicatif. L'approche la plus répandue repose sur une instrumentation de l'application puis une éxécution préliminaire de cette dernière afin de récupérer les informations nécessaires au calcul du placement pour les exécutions suivantes. Or, une analyse statique (e.g. à la compilation) pourrait également générer des informations pertinentes. Cependant, elles seraient moins précises que celles provenant d'une exécution ou d'une simulation. La question de la relation entre la précision de l'information

fournie et du gain attendue se pose alors. Est-il nécessaire ou non de fournir une information très précise pour obtenir des gains significatifs ? À l'heure actuelle, cette question est ouverte et n'a pas – à ma connaissance – été traitée.

5.3.1.2 Placement multicritères

Dans sa forme actuelle, le placement est calculé en procédant à l'optimisation d'une fonction de coût. Cette fonction utilise le plus souvent un unique critère comme le volume des données échangées ou bien la congestion au niveau du réseau. Il est plus rare que de multiples critères soient considérés conjointement. En particulier, il serait intéressant de modéliser les transactions sur les bus mémoire afin de prendre en compte les éventuelles collisions et de les intégrer dans le calcul du placement. Également, il est courant dans les applications parallèles basées sur le passage de messages de rechercher un recouvrement calcul/communication qui soit maximal pour que le coût de ces communications soit totalement masqué par les phases de calcul de l'application. Or, le placement de processus est susceptible d'influencer sur ce recouvrement et il serait important de quantifier cette influence pour la prendre éventuellement en compte dans le placement. Enfin, les informations prises en compte ne tiennent pas compte de la réalité temporelle des échanges. Pour une quantité de données échangée, nous ne savons pas exploiter le fait qu'elle a pu être transférer avec un unique ou de multiples messages et dans ce cas avec quel espacement. La détermination de phases pour le placement nous permettrait d'élargir les types d'applications visées, avec comme par exemple des maillages adaptatifs.

5.3.1.3 Vers des schémas applicatifs génériques ?

Une autre piste à explorer repose sur la notion de classes d'applications. En effet, il serait envisageable de classifier les applications selon leurs comportements (e.g. applications *compute-bound*, *latency-bound*, etc.) et d'utiliser un schéma archétypal représentatif de cette classe, plutôt que de récolter des informations précises, spécifiques à une application en particulier. Là encore, il faut se poser la question du lien entre la précision de l'information fournie et du gain de performances escompté afin de déterminer la viabilité de l'approche. L'établissement d'une telle bibliothèque de schémas applicatifs génériques serait de plus utilisable non seulement dans le cadre du deploiement et de l'exécution des applications (placement) mais encore dans le cadre de l'allocation de ressources basées sur l'application, ainsi qu'expliqué en section 3.4.

5.3.1.4 Quid des architectures non hiérarchiques ?

Une hypothèse forte de nos travaux consiste à considérer les architectures cibles comme des entités avec une nature hiérarchique. En effet, la modélisation offerte par hwloc propose d'exploiter une structure de données arborescente et TREEMATCH effectue ses calculs dans ce contexte particulier. Cela peut être vu comme une contrainte qu'il serait pertinent d'assouplir. Dans un premier temps, il est important d'indiquer que cette vision hiérarchique correspond à une réalité matérielle, au moins au niveau des nœuds de calcul actuels. Les choses sont plus mitigées au niveau des topologies des réseaux d'interconnexion mais il est possible de s'accomoder d'une approche hiérarchique au prix d'une perte d'information. Ainsi qu'indiqué précédemment, je pense qu'il est préférable d'exploiter une information, fût-elle partielle, plutôt que de ne rien exploiter du tout. Ceci étant dit, la question des topologies non hiérarchiques mérite d'être posée et une réflexion semble nécessaire quant à l'impact sur les travaux et logiciels développés. Ainsi, TREEMATCH va devoir revoir son mode de fonctionnement pour

s'adresser à des classes de graphes plus génériques[4], ce qui pose également la question des interactions possibles avec le partitionneur Scotch, développé dans TADAaaM par François PELLEGRINI. En ce qui concerne TREEMATCH plus particulièrement, le fait d'avoir un calcul centralisé n'est pas totalement satisfaisant, et même si une version partiellement distribuée de la fonction `MPI_Dist_graph_create` existe, cela ne saurait remplacer une version de TREEMATCH instrinsèquement parallèle.

5.3.1.5 Poursuivre les efforts pour MPI 4.X

Une dernière piste concerne bien évidemment le travail de standardisation mené depuis environ deux ans. Déjà, il nous paraît fondamental de faire accepter le principe des communicateurs topologiques car cela permettrait d'abord de standardiser des comportements et fonctionnalités qui existent par ailleurs au sein des principales implémentations libres de MPI (i.e. MPICH et Open MPI). Ce premier élément nous permettra d'en introduire d'autres afin d'enrichir la proposition initiale et d'offrir une interface à la fois simple mais suffisamment riche à l'usage. Un point crucial réside dans la capacité pour l'application de déterminer au niveau applicatif quelles ressources elle utilise par le biais de noms mais ce point ne fait pas du tout consensus au sein du Forum MPI. Nous souhaitons également déterminer une interface permettant une description explicite de la topologie matérielle (à l'opposé de l'approche implicite offerte par les communicateurs topologiques) pour répondre aux besoins de certaines applications qui utilisent déjà des solutions *ad-hoc* non portables et qu'il conviendrait d'arriver à généraliser. Enfin, le point le moins consensuel réside dans la standardisation des moyens de déploiement dans MPI. Je pense que c'est un aspect qui mérite que le Forum se penche très sérieusement dessus et qu'en l'absence de réponse appropriée, nous devrons nous résoudre à proposer des solutions englobant non seulement le passage de messages mais également d'autres modèles de programmation, un peu à l'image de ce que Hippo et libquo sont capables de faire. De ce point de vue, il est fondamental de participer au groupe de travail établi par le CEA à ce sujet car il rassemble un certain nombre d'acteurs importants dans ce domaine avec lesquels il est nécessaire de dialoguer.

5.3.2 Poursuivre les efforts de standardisation, au sens large

5.3.2.1 Standardiser au maximum pour mieux diffuser les idées et solutions

De façon plus générale, je crois qu'une des clefs pour une meilleure diffusion et acceptation des travaux menés dans TADaaM passe par une phase de standardisation. Cette dernière peut se concrétiser de différentes manières : la standardisation *de facto* avec l'adoption massive et rapide d'une solution venant répondre à un besoin qui n'est pas déjà couvert par ailleurs. C'est le cas de hwloc (et peut-être de Hippo ?) mais une telle approche est hasardeuse car elle ne repose pas uniquement sur les qualités intrinsèques de la solution proposée mais également sur des circonstances qu'il est impossible de prévoir et encore moins de contrôler. Dans la majorité des cas, cela passe par la participation à des instances dédiées, comme le Forum MPI ou bien l'*Architecture Review Board* d'OpenMP (par exemple). Je crois que par le passé nous avons sans doute été trop timorés et n'avons pas assez mis en avant nos travaux notamment dans le Forum MPI. Le fait de standardiser les choses permet de confronter les points de vue, d'appréhender différemment certaines problématiques et surtout évite des propositions faites en ordre dispersé quand les objectifs sont similaires. Pour autant, cela ne signifie pas qu'il faut obligatoirement rentrer dans un cadre ou un moule car la diversité permet l'éclosion des

4. TREEMATCH a d'ailleurs changé de nom récemment, pour devenir TOPOMATCH (https://gitlab.inria.fr/ejeannot/topomatch)...

idées et peut créer une émulation positive. Par exemple, l'existence simultanée d'Open MPI et de MPICH a vraisemblablement permis d'atteindre un degré de maturité et de sophistication qui n'aurait pas été possible autrement. Selon moi, il faut continuer à participer à différents groupes de travail du Forum MPI et aux initiatives proposées, notamment en ce qui concerne le placement (groupe de travail du CEA). Si une solution émerge dans ce cadre, permettant de résoudre les problèmes posés par le placement de façon générale, alors il est possible que cela influence en retour les instances officielles de standardisation plus promptes à l'inertie. Au final, ce sera l'ensemble de la communauté qui sera gagnante car le *statu quo* est de moins en moins acceptable.

5.3.2.2 Interagir avec d'autres modèles et paradigmes de programmation parallèle

L'absence de consensus au sujet des méthodes hybrides de programmation et les résultats mitigés obtenus à l'origine, couplée avec un manque de culture du multithreading parmi les développeurs des principales implémentations de MPI a conduit à l'élaboration d'un standard MPI relativement timoré du point de vue de la gestion du multithreading au sein d'une application MPI. Cependant, si peu d'éléments sont présents, ils ont au moins le mérite d'exister. Je pense que cela n'est pas non plus suffisant et qu'un meilleur dialogue avec les communautés OpenMP et PGAS devrait être instauré, de façon à proposer dans MPI des améliorations et changements compatibles avec les mécanismes proposés par d'autres modèles en vue d'une conception hybride des applications. Il est assez étrange et frappant que le nécessaire (selon moi) dialogue inter-communautés soit si faible, pour ne pas dire inexistant. La présence dans le Forum MPI de personnes par ailleurs membres de l'ARB OpenMP n'a pas permis de faciliter les échanges pour le moment ...

5.3.3 Repenser MPI et sa place dans le paysage du HPC (et au-delà)

Pour finir, il me semble important de s'interroger sur la place que MPI occupe dans le paysage du HPC. Ainsi que nous l'avons expliqué dans ce document, MPI va rester encore pour un temps comme un modèle/environnement de programmation avec lequel il faudra compter. Les raisons sont multiples :

- L'inertie : les codes développés avec MPI sont parfois longs à mettre au point et à optimiser. Certains utilisateurs ne sont pas prêts à «faire un pari» avec des alternatives non totalement éprouvées comme peut l'être MPI ;
- Le manque de remplaçant sérieux : jusqu'à présent, aucun candidat n'a réussi à détrôner MPI, notamment parce que le standard évolue et que ses implémentations ont atteint un grand degré de maturité ce qui conduit à de très bonnes performances ;
- La richesse de l'interface : plusieurs styles de programmation sont possibles au sein d'une même bibliothèque MPI (opérations bilatérales, unilatérales, etc.) ;
- La robustesse du standard : le processus de standardisation met MPI à l'abri des effets de mode. Les membres du Forum MPI sont de plus très attachés à la pérennité des propositions de nouvelles fonctionnalités. Les utilisateurs savent qu'une communauté très active travaille à la fois sur l'interface et ses implémentations.

Pour autant, MPI devient de plus en plus complexe à utiliser et parfois ses subtilités sont difficiles à percevoir. Ainsi que nous l'avions rapidement évoqué en section 4.2.2, je crois qu'à l'avenir, MPI sera de moins en moins utilisé par les développeurs d'applications qui étaient la cible originale de cette interface, ou alors de façon très basique. Il va falloir compter sur des utilisateurs *avertis* de MPI, c'est-à-dire des gens qui participent au Forum MPI ou suivent de très près les discussions et les évolutions mais qui ne font pas forcément partie des personnes implémentant le standard. Une autre solution sera de pouvoir reposer sur des bibliothèques

spécialisées, fournissant des services très pointus (e.g. un peu comme Hsplit) mais que les développeurs d'applications (physiciens, chimistes, climatologues, etc.) seraient bien incapables de mettre au point la plupart du temps.

Il y a là un paradoxe qui ne peut que nous interroger : pourquoi MPI n'est-il pas plus populaire au moment même où des domaines connexes au HPC comme le *Big Data*, la *Data Science* ou encore l'intelligence artificielle qui sont de potentiels gros consommateurs de puissance de calcul (et donc utilisateurs de machines parallèles) connaissent un essor considérable ? En effet, les logiciels dédiés pour ces domaines fournissent à leurs utilisateurs des interfaces spécifiques complètement différentes de MPI (e.g. MapReduce, Hadoop, Spark), mais bien plus simples tandis que ce dernier est néanmoins utilisé au niveau des infrastructures de calcul (e.g. *Amazon Web Services*). Il est possible que la complexité *perçue* de MPI joue sans doute en sa défaveur alors que l'utilisation appropriée d'un sous-ensemble de ses fonctionnalités permettrait sans doute de répondre aux besoins. Là encore, une approche basée sur des bibliothèques spécialisées serait à explorer.

Liste des publications

Chapitres de livres

[Li1] Torsten HOEFLER, Emmanuel JEANNOT et Guillaume MERCIER. « Chapter 5 : An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing ». In : *High Performance Computing on Complex Environments*. Sous la dir. d'Emmanuel JEANNOT et Julius ŽILINSKAS. Wiley, 2014, p. 65-84.

Revues internationales avec comité de lecture

[RI1] Yiannis GEORGIOU et al. « Topology-Aware Job Mapping. » In : *International Journal of high-Performance Computing applications* 32.1 (2018), p. 14-27. DOI : 10.1177/1094342017727061.

[RI2] Brice GOGLIN et al. « Hardware topology management in MPI applications through hierarchical communicators ». In : *Parallel Computing* 76 (2018), p. 70-90. DOI : 10.1016/j.parco.2018.05.006. URL : https://doi.org/10.1016/j.parco.2018.05.006.

[RI3] Emmanuel JEANNOT, Guillaume MERCIER et François TESSIER. « Process Placement in Multicore Clusters : Algorithmic Issues and Practical Techniques ». In : *IEEE Transactions on Parallel and Distributed Systems* 25.4 (2014), p. 993-1002. DOI : 10.1109/TPDS.2013.104.

[RI4] Darius BUNTINAS, Guillaume MERCIER et William GROPP. « Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem ». In : *Parallel Computing* 33.9 (2007), p. 634-644.

[RI5] Olivier AUMAGE et al. « High Performance Computing on Heterogeneous Clusters with the Madeleine II Communication Library ». In : *Cluster Computing* 5.1 (2002), p. 43-54.

Conférences internationales avec comité de lecture

[CI1] Purushotham V. BANGALORE et al. « Exposition, Clarification, and Expansion of MPI Semantic Terms and Conventions : Is a nonblocking function permitted to block? » In : *Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI 2019, Zürich, Switzerland, September 11-13, 2019*. To appear. ACM, 2019.

[CI2] George BOSILCA et al. « Online Dynamic Monitoring of MPI Communications ». In : *Euro-Par 2017 : Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*. T. 10417. Lecture Notes in Computer Science. Springer, 2017, p. 49-62. DOI : 10.1007/978-3-319-64203-1_4.

[CI3] Yiannis GEORGIOU et al. « Topology-aware resource management for HPC applications ». In : *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*. ACM, 2017, p. 17. DOI : 10.1145/3007748.

[CI4] Emmanuel JEANNOT, Farouk MANSOURI et Guillaume MERCIER. « A hierarchical model to manage hardware topology in MPI applications ». In : *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*. ACM, 2017, 9 :1-9 :11. DOI : 10.1145/3127024.3127030.

[CI5] Emmanuel JEANNOT et al. « Communication and Topology-aware Load Balancing in Charm++ with TreeMatch ». In : *IEEE Cluster 2013*. Indianapolis, USA, 2013, p. 1-8. DOI : 10.1109/CLUSTER.2013.6702666.

[CI6] Emmanuel JEANNOT et Guillaume MERCIER. « Improving MPI Applications Performance on Multicore Clusters with Rank Reordering ». In : *EuroMPI*. Sous la dir. de SPRINGER. T. 6960. Santorin, Grèce, 2011, p. 39-49. DOI : 10.1007/978-3-642-24449-0.

[CI7] François BROQUEDIS et al. « Hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications ». In : *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*. Pisa, Italia : IEEE Computer Society Press, 2010.

[CI8] Emmanuel JEANNOT et Guillaume MERCIER. « Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures ». In : *Euro-Par 2010 Parallel Processing Europar*. Sous la dir. de Pasqua D'AMBRA, Mario Rosario GUARRACINO et Domenico TALIA. T. 6272. Lecture Notes on Computer Science. Ischia Italie : Springer, août 2010, p. 199-210.

[CI9] Darius BUNTINAS et al. « Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis ». In : *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*. Vienna, Austria : IEEE Computer Society Press, 2009.

[CI10] Guillaume MERCIER et Jérôme CLET-ORTEGA. « Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments ». In : *EuroPVM/MPI*. T. 5759. Lecture Notes in Computer Science. Espoo, Finland : Springer, 2009, p. 104-115.

[CI11] Guillaume MERCIER et al. « NewMadeleine : An Efficient Support for High-Performance Networks in MPICH2 ». In : *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*. Rome, Italy : IEEE Computer Society Press, mai 2009.

[CI12] Darius BUNTINAS, Guillaume MERCIER et William GROPP. « Data Transfer in a SMP System : Study and Application to MPI ». In : *Proc. 35th International Conference on Parallel Processing (ICPP 2006)*. Colombus, Ohio, août 2006.

[CI13] Darius BUNTINAS, Guillaume MERCIER et William GROPP. « Design and Evaluation of Nemesis : a Scalable, Low-Latency, Message-Passing Communication Subsystem ». In : *Proc. 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2006)*. Held in conjunction with IEEE Computer Society et ACM. Singapore, mai 2006.

[CI14] Darius BUNTINAS, Guillaume MERCIER et William GROPP. « Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface : Proceedings of the 13th European PVM/MPI Users Group Meeting (Euro PVM/MPI 2006)*. Bonn, sept. 2006.

[CI15] Olivier AUMAGE et Guillaume MERCIER. « MPICH/MadIII : a Cluster of Clusters-Enabled MPI Implementation ». In : *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*. Held in conjunction with IEEE Computer Society et ACM. Tokyo, mai 2003, p. 26-35.

[CI16] Olivier AUMAGE, Guillaume MERCIER et Raymond NAMYST. « MPICH/Madeleine : a True Multi-Protocol MPI for High-Performance Networks ». In : *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*. Held in conjunction with IEEE Computer Society et ACM. San Fransisco, avr. 2001, p. 51.

[CI17] Olivier AUMAGE et al. « A Portable and Efficient Communication Library for High-Performance Cluster Computing ». In : *IEEE International Conference on Cluster Computing (Cluster 2000)*. Chemnitz, nov. 2000, p. 78-87.

Ateliers internationaux avec comité de lecture

[AI1] Yiannis GEORGIOU, Guillaume MERCIER et Adèle VILLIERMET. « Large-Scale Experiment for Topology-Aware Resource Management ». In : *Euro-Par 2017 : Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*. 2017, p. 179-186. DOI : 10.1007/978-3-319-75178-8_15.

[AI2] Emmanuel JEANNOT, Guillaume MERCIER et Francois TESSIER. « Topology and Affinity Aware Hierarchical and Distributed Load-Balancing in Charm++ ». In : *First International Workshop on Communication Optimizations in HPC, COMHPC@SC 2016, Salt Lake City, UT, USA, November 18, 2016*. 2016, p. 63-72. DOI : 10.1109/COMHPC.2016.012.

[AI3] Olivier AUMAGE et al. « High-Performance Multi-Rail Support with the NewMadeleine Communication Library ». In : *HCW 2007 : the Sixteenth International Heterogeneity in Computing Workshop, held in conjunction with IPDPS 2007*. Long Beach, California, USA, mar. 2007, p. 9.

Conférences françaises avec comité de lecture

[CF1] François TESSIER, Emmanuel JEANNOT et Guillaume MERCIER. « TreeMatch : Un algorithme de placement de processus sur architectures multicœurs ». In : *Compass 2013*. 2013.

Divers

[Misc1] DANIEL BALKANSKI AND GUILLAUME MERCIER. « Performance Evaluation of MPICH-Madeleine against the Multi-Protocol MPI Implementations for Homogeneous and Heterogeneous SMP Clusters ». In : *Proc. 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*. Extended Abstract. Held in conjunction with IEEE Computer Society et ACM. Cardiff, mai 2005.

Rapports de recherche

[RR1] Brice GOGLIN et al. *Hardware topology management in MPI applications through hierarchical communicators*. Research Report RR-9077. Inria Bordeaux Sud-Ouest, nov. 2018.

[RR2] Emmanuel JEANNOT, Guillaume MERCIER et François TESSIER. *Process Placement in Multicore Clusters : Algorithmic Issues and Practical Techniques*. Research Report RR-8269. Inria Bordeaux Sud-Ouest, mar. 2013.

[RR3] Darius BUNTINAS, Guillaume MERCIER et William GROPP. *Data Transfer in a SMP System : Study and Application to MPI*. Research Report ANL/MCS-P1306-1105. Argonne National Laboratory, nov. 2005.

[RR4] Darius BUNTINAS, Guillaume MERCIER et William GROPP. *Design and Evaluation of Nemesis : a Scalable, Low-Latency, Message-Passing Communication Subsystem*. Research Report ANL/MCS-TM-292. Argonne National Laboratory, nov. 2005.

[RR5] Olivier AUMAGE, Guillaume MERCIER et Raymond NAMYST. *MPICH-Madeleine : a True Multi-Protocol MPI for High-Performance Networks*. Research Report RR-4016. INRIA Rhône-Alpes, Laboratoire de l'Informatique du Parallélisme, oct. 2000.

[RR6] Olivier AUMAGE et al. *A Portable and Efficient Communication Library for High-Performance Cluster Computing*. Research Report RR-3976. INRIA Rhône-Alpes, Laboratoire de l'Informatique du Parallélisme, juil. 2000.

Thèse

[Th1] Guillaume MERCIER. « High-Performance Portable Communication in Hierarchical, Heterogeneous and Dynamical Environments ». In french only. Ph.D Thesis. Bordeaux 1 University, déc. 2004.

Bibliographie

[2] Ahmed H. ABDEL-GAWAD, Mithuna THOTTETHODI et Abhinav BHATELE. « RAHTM : Routing Algorithm Aware Hierarchical Task Mapping ». In : *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE Computer Society, 2014, p. 325-335. ISBN : 978-1-4799-5500-8. DOI : 10.1109/SC.2014.32. URL : https://doi.org/10.1109/SC.2014.32.

[3] T. AGARWAL et al. « Topology-aware task mapping for reducing communication contention on large parallel machines ». In : *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI : 10.1109/IPDPS.2006.1639379. URL : https://doi.org/10.1109/IPDPS.2006.1639379.

[4] Hasan Metin AKTULGA et al. « Topology-Aware Mappings for Large-Scale Eigenvalue Problems ». In : *Euro-Par 2012 Parallel Processing - 18th International Conference*. T. 7484. Lecture Notes in Computer Science. Rhodes Island, Greece, 2012, p. 830-842.

[5] Carl ALBING et al. « Scalable Node Allocation for Improved Performance in Regular and Anisotropic 3D Torus Supercomputers ». In : *EuroMPI 2011, Santorini, Greece*. 2011, p. 61-70.

[6] Sebastian von ALFTHAN, Ilja HONKONEN et Minna PALMROTH. « Topology Aware Process Mapping ». In : *Applied Parallel and Scientific Computing*. Sous la dir. de Pekka MANNINEN et Per ÖSTER. Springer Berlin Heidelberg, 2013, p. 297-308. ISBN : 978-3-642-36803-5.

[7] B. W. KERNIGHAN AND S. LIN. « An efficient heuristic procedure for partitioning graphs ». In : *Bell System Technical Journal* 49.2 (fév. 1970), p. 291-307.

[8] Takanobu BABA, Yoshifumi IWAMOTO et Tsutomu YOSHINAGA. « A network-topology independent task allocation strategy for parallel computers ». In : *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*. IEEE Computer Society, 1990, p. 878-887. ISBN : 0-89791-412-0. DOI : 10.1109/SUPERC.1990.130114. URL : https://doi.org/10.1109/SUPERC.1990.130114.

[9] D. H. BAILEY et al. *NAS Parallel Benchmark Results*. Rapp. tech. 94-006. RNR, 1994.

[10] Pavan BALAJI et al. « Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems ». In : *Computer Science - R&D* 26.3-4 (2011), p. 247-256. DOI : 10.1007/s00450-011-0168-y. URL : https://doi.org/10.1007/s00450-011-0168-y.

[11] Pavel BAR et al. « Running Parallel Applications with Topology-Aware Grid Middleware ». In : *Fifth International Conference on e-Science, e-Science 2009, 9-11 December 2009, Oxford, UK*. IEEE Computer Society, 2009, p. 292-299. ISBN : 978-0-7695-3877-8. DOI : 10.1109/e-Science.2009.48. URL : https://doi.org/10.1109/e-Science.2009.48.

[12] Francine BERMAN et Lawrence SNYDER. « On Mapping Parallel Algorithms into Parallel Architectures ». In : *J. Parallel Distrib. Comput.* 4.5 (1987), p. 439-458. DOI : 10.1016/0743-7315(87)90018-9. URL : https://doi.org/10.1016/0743-7315(87)90018-9.

[13] David E. BERNHOLDT et al. « A survey of MPI usage in the US exascale computing project ». In : *Concurrency and Computation : Practice and Experience* 0.0 (). e4851 cpe.4851, e4851. DOI : 10.1002/cpe.4851. URL : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851.

[14] Gyan BHANOT et al. « Optimizing task layout on the Blue Gene/L supercomputer ». In : *IBM Journal of Research and Development* 49.2-3 (2005), p. 489-500. DOI : 10.1147/rd.492.0489. URL : https://doi.org/10.1147/rd.492.0489.

[15] Abhinav BHATELE, Eric J. BOHM et Laxmikant V. KALÉ. « Topology aware task mapping techniques : an api and case study ». In : *PPOPP*. 2009, p. 301-302. URL : http://doi.acm.org/10.1145/1504176.1504225.

[16] Abhinav BHATELE, Eric J. BOHM et Laxmikant V. KALÉ. « Optimizing communication for Charm++ applications by reducing network contention ». In : *Concurrency and Computation : Practice and Experience* 23.2 (2011), p. 211-222. DOI : 10.1002/cpe.1637. URL : https://doi.org/10.1002/cpe.1637.

[17] Abhinav BHATELE et Laxmikant V. KALÉ. « Benefits of Topology Aware Mapping for Mesh Interconnects ». In : *Parallel Processing Letters* 18.4 (2008), p. 549-566. DOI : 10.1142/S0129626408003569. URL : https://doi.org/10.1142/S0129626408003569.

[18] Abhinav BHATELE et al. « Mapping applications with collectives over sub-communicators on torus networks ». In : *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*. IEEE/ACM, 2012, p. 97. ISBN : 978-1-4673-0804-5. DOI : 10.1109/SC.2012.75. URL : https://doi.org/10.1109/SC.2012.75.

[19] Abhinav BHATELE et al. « Optimizing the performance of parallel applications on a 5D torus via task mapping ». In : *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*. IEEE Computer Society, 2014, p. 1-10. ISBN : 978-1-4799-5976-1. DOI : 10.1109/HiPC.2014.7116706. URL : https://doi.org/10.1109/HiPC.2014.7116706.

[20] Amanda BIENZ, William D. GROPP et Luke N. OLSON. « TAPSpMV : Topology-Aware Parallel Sparse Matrix Vector Multiplication ». In : *CoRR* abs/1612.08060 (2016). URL : http://arxiv.org/abs/1612.08060.

[21] S.H. BOKHARI. « On the Mapping Problem ». In : *IEEE Transactions on Computers* 30.3 (1981), p. 207-214. ISSN : 0018-9340. DOI : 10.1109/TC.1981.1675756. URL : http://doi.ieeecomputersociety.org/10.1109/TC.1981.1675756.

[22] S. Wayne BOLLINGER et Scott F. MIDKIFF. « Heuristic Technique for Processor and Link Assignment in Multicomputers ». In : *IEEE Trans. Comput.* 40.3 (1991), p. 325-333. ISSN : 0018-9340. DOI : 10.1109/12.76410. URL : http://dx.doi.org/10.1109/12.76410.

[23] Erik G. BOMAN et al. « The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing : Partitioning, ordering and coloring ». In : *Scientific Programming* 20.2 (2012), p. 129-150. DOI : 10.3233/SPR-2012-0342. URL : https://doi.org/10.3233/SPR-2012-0342.

[24] Cyril BORDAGE, Clément FOYER et Brice GOGLIN. « Netloc : A Tool for Topology-Aware Process Mapping ». In : *Euro-Par 2017 : Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*. T. 10659. Lecture Notes in Computer Science. Springer, 2017, p. 157-166. ISBN : 978-3-319-75177-1. DOI : 10.1007/978-3-319-75178-8_13. URL : https://doi.org/10.1007/978-3-319-75178-8%5C_13.

[25] Cyril BORDAGE et Emmanuel JEANNOT. « Process Affinity, Metrics and Impact on Performance : An Empirical Study ». In : *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. IEEE Computer Society, 2018, p. 523-532. ISBN : 978-1-5386-5815-4. DOI : 10.1109/CCGRID.2018.00079. URL : https://doi.org/10.1109/CCGRID.2018.00079.

[26] B. BRANDFASS, T. ALRUTZ et T. GERHOLD. « Rank Reordering for MPI Communication Optimization ». In : *Computer & Fluids* 80 (juil. 2013), p. 372-380. DOI : http://dx.doi.org/10.1016/j.compfluid.2012.01.019.

[27] Ron BRIGHTWELL, Trammell HUDSON et Kevin PEDRETTI. « SMARTMAP : Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor ». In : *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2008*. Austin, TX : ACM Press, nov. 2008.

[28] S. BROWNE et al. « A Portable Programming Interface for Performance Evaluation on Modern Processors ». In : *The International Journal of High Performance Computing Applications* 14.3 (2000), p. 189-204. DOI : 10.1177/109434200001400303. URL : https://doi.org/10.1177/109434200001400303.

[29] C. SMITH, B. MCMILLAN et I. LUMB. « Topology Aware Scheduling in the LSF Distributed Resource Manager ». In : *Proceedings of the Cray User Group Meeting*. 2001.

[30] Nicolas CAPIT et al. « A batch scheduler with high level components ». In : *Cluster computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom : IEEE, 2005. URL : https://hal.archives-ouvertes.fr/hal-00005106.

[31] Franck CAPPELLO et Daniel ETIEMBLE. « MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks ». In : *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Dallas, Texas, United States : IEEE Computer Society, 2000, p. 12.

[32] Ümit V. ÇATALYÜREK et Cevdet AYKANAT. « PaToH (Partitioning Tool for Hypergraphs) ». In : *Encyclopedia of Parallel Computing*. Springer, 2011, p. 1479-1487. ISBN : 978-0-387-09765-7. DOI : 10.1007/978-0-387-09766-4_93. URL : https://doi.org/10.1007/978-0-387-09766-4%5C_93.

[33] Lei CHAI et al. « Designing An Efficient Kernel-level and User-level Hybrid Approach for MPI Intra-node Communication on Multi-core Systems ». In : *Proceedings of the IEEE International Conference on Parallel Processing (ICPP-2008)*. Portland, Oregon : IEEE Computer Society Press, sept. 2008.

[34] *ChaMPIon/Pro*. URL : http://www.spscicomp.org/ScicomP8/Presentations/ChaMPIonPro.ppt/.

[35] Daniel G. CHAVARRÍA-MIRANDA, Jarek NIEPLOCHA et Vinod TIPPARAJU. « Topology-aware tile mapping for clusters of SMPs ». In : *Proceedings of the Third Conference on Computing Frontiers, 2006, Ischia, Italy, May 3-5, 2006*. ACM, 2006, p. 383-392. ISBN : 1-59593-302-6. DOI : 10.1145/1128022.1128073. URL : https://doi.org/10.1145/1128022.1128073.

[36] Hu CHEN et al. « MPIPP : an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters ». In : *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*. Sous la dir. de Gregory K. EGAN et Yoichi MURAOKA. ACM, 2006, p. 353-360. ISBN : 1-59593-282-8. URL : http://doi.acm.org/10.1145/1183401.1183451.

[37] Suresh CHITTOR et Richard J. ENBODY. « Predicting the Effect of Mapping on the Communication Performance of Large Multicomputers ». In : *Proceedings of the International Conference on Parallel Processing, ICPP '91, Austin, Texas, USA, August 1991. Volume II : Software.* CRC Press, 1991, p. 1-4.

[38] Jérôme CLET-ORTEGA. « Exploitation efficace des architectures parallèles de type grappes de NUMA à l aide de modèles hybrides de programmation ». Thèse de Doctorat. Université de Bordeaux, Avril 2012.

[39] Chris YEOH. *Cross Memory Attach*. 2010. URL : http://lwn.net/Articles/405284/.

[40] CNRS. *Ada : Executing a hybrid MPI/OpenMP job in batch under the Intel environment*. 2016. URL : http://130.84.40.142/eng/ada/ada-exec_mpi_openmp_batch_env_intel-eng.html.

[41] COMPAQ, INTEL et MICROSOFT. *Virtual Interface Architecture Specification V 1.0*. 1997. URL : http://www.cs.uml.edu/~bill/cs520/VI_spec.pdf.

[42] Adaptive COMPUTING. *Torque Resource Manager*. URL : http://docs.adaptivecomputing.com/torque/6-0-0/Content/topics/torque/2-jobs/monitoringJobs.htm.

[43] Camille COTI, Thomas HÉRAULT et Franck CAPPELLO. « MPI Applications on Grids : A Topology Aware Approach ». In : *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*. T. 5704. Lecture Notes in Computer Science. Springer, 2009, p. 466-477. ISBN : 978-3-642-03868-6. DOI : 10.1007/978-3-642-03869-3_45. URL : https://doi.org/10.1007/978-3-642-03869-3_45.

[44] CRAY. *Cray ALPS*. URL : https://slurm.schedmd.com/SUG13/Refactor_ALPS.pdf.

[45] CRAY. *Cray Performance Measurement and Analysis Tool*. 2017. URL : https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/about-the-cray-performance-measurement-and-analysis-tools-user-guide,.

[46] Eduardo H. M. CRUZ et al. « EagerMap : A Task Mapping Algorithm to Improve Communication and Load Balancing in Clusters of Multicore Systems ». In : *ACM Transactions on Parallel Computing* 5.4 (2019), 17 :1-17 :24. URL : https://dl.acm.org/citation.cfm?id=3309711.

[47] E. CUTHILL et J. MCKEE. « Reducing the bandwidth of sparse symmetric matrices ». In : *Proceedings of the 1969 24th national conference*. ACM '69. New York, NY, USA : ACM, 1969, p. 157-172.

[48] DAVID SOLT. *A profile based approach for topology aware MPI rank placement*. 2007. URL : http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt.

[49] Erik D. DEMAINE. « A Threads-Only MPI Implementation for the Development of Parallel Programs ». In : *Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97)*. Winnipeg, Manitoba, Canada, juil. 1997, p. 153-163.

[50] R. H. DENNARD et al. « Design of ion-implanted MOSFET's with very small physical dimensions ». In : *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007). Reprinted from the IEEE Journal of Solid-State Circuits, Vol. SC-9, October 1974, pp. 256-268., p. 38-50. ISSN : 1098-4232. DOI : 10.1109/N-SSC.2007.4785543.

[51] Mehmet DEVECI et al. « Exploiting Geometric Partitioning in Task Mapping for Parallel Computers ». In : *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 2014, p. 27-36. ISBN : 978-1-4799-3799-8. DOI : 10.1109/IPDPS.2014.15. URL : https://doi.org/10.1109/IPDPS.2014.15.

[52] Mehmet DEVECI et al. « Fast and High Quality Topology-Aware Task Mapping ». In : *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. IEEE Computer Society, 2015, p. 197-206. ISBN : 978-1-4799-8649-1. DOI : 10.1109/IPDPS.2015.93. URL : https://doi.org/10.1109/IPDPS.2015.93.

[53] Mehmet DEVECI et al. « Hypergraph partitioning for multiple communication cost metrics : Model and methods ». In : *J. Parallel Distrib. Comput.* 77 (2015), p. 69-83. DOI : 10.1016/j.jpdc.2014.12.002. URL : https://doi.org/10.1016/j.jpdc.2014.12.002.

[54] Karen DEVINE et al. « Zoltan Data Management Services for Parallel Dynamic Applications ». In : *Computing in Science and Engineering* 4.2 (2002), p. 90-97.

[55] Matthias DIENER et al. « Characterizing communication and page usage of parallel applications for thread and data mapping ». In : *Perform. Eval.* 88 (2015), p. 18-36. DOI : 10.1016/j.peva.2015.03.001. URL : https://doi.org/10.1016/j.peva.2015.03.001.

[56] Jack DONGARRA et al. « The International Exascale Software Project : A Call To Cooperative Action By the Global High-Performance Community ». In : *Int. J. High Perform. Comput. Appl.* 23.4 (2009), p. 309-322. ISSN : 1094-3420. URL : http://dx.doi.org/10.1177/1094342009347714.

[57] Jörg DÜMMLER, Thomas RAUBER et Gudula RÜNGER. « Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters ». In : *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*. IEEE Computer Society, 2008, p. 141-148. DOI : 10.1109/ICPP.2008.42. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4625814.

[58] Graham E.FAGG et Jack J. DONGARRA. « Heterogeneous MPI Application Interoperation and Process Management under PVMPI ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 4th European PVM/MPI Users' Group Meeting*. T. 1332. Lecture Notes in Computer Science. Springer, nov. 1997, p. 91-98.

[59] EVELYN DUESTERWALD, ROBERT W. WISNIEWSKI, PETER F. SWEENEY, GHEORGHE CASCAVAL AND STEPHEN E. SMITH. *Method and System for Optimizing Communication in MPI Programs for an Execution Environment*. 2008. URL : http://www.faqs.org/patents/app/20080288957.

[60] Charles M. FIDUCCIA et Robert M. MATTHEYSES. « A linear-time heuristic for improving network partitions ». In : *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*. ACM/IEEE, 1982, p. 175-181. DOI : 10.1145/800263.809204. URL : https://doi.org/10.1145/800263.809204.

[61] Ian FOSTER, Jonathan GEISLER et Steven TUECKE. *MPI on the I-WAY : A Wide-Area, Multimethod Implementation of the Message Passing Interface*. Rapp. tech. Argonne National Laboratory.

[62] FUJITSU. *Interconnect topology-aware resource assignment*. URL : http://www.fujitsu.com/global/Images/technical-computing-suite-bp-sc12.pdf.

[63] FUJITSU. *Hardware Topology Aware MPI extensions on Fujitsu PRIMEHPC FX10, FX100*. 2018. URL : https://github.com/mpiwg-hw-topology/hw-topology-issues/blob/master/MPI-HW-Topo-WG-fujitsu-extention.pdf.

[64] Edgar GABRIEL et al. « Distributed Computing in a Heterogeneous Computing Environment ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Sous la dir. de Vassil ALEXANDROV et Jack DONGARRA. Lecture Notes in Computer Sciences. Springer, 1998, p. 180-188.

[65] Edgar GABRIEL et al. « Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*. T. 3241. Lecture Notes in Computer Science. Springer, 2004, p. 97-104. ISBN : 3-540-23163-3. DOI : 10.1007/978-3-540-30218-6_19. URL : https://doi.org/10.1007/978-3-540-30218-6_19.

[66] Juan J. GALVEZ, Nikhil JAIN et Laxmikant V. KALÉ. « Automatic topology mapping of diverse large-scale parallel applications ». In : *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*. ACM, 2017, 17 :1-17 :10. ISBN : 978-1-4503-5020-4. DOI : 10.1145/3079079.3079104. URL : https://doi.org/10.1145/3079079.3079104.

[67] F. GARCÍA, A. CALDERÓN et J. CARRETERO. « MiMPI : A Multithread-Safe Implementation of MPI ». In : *Recent Advances in PVM and MPI. 6th PVM/MPI European User's Group Meeting*. T. 1697. Lecture Notes in Computer Science. Barcelone : Springer-Verlag, sept. 1999, p. 207-214.

[68] William L. GEORGE, John G. HAGEDORN et Judith E. DEVANEY. « IMPI : Making MPI Interoperable ». In : *Journal of research of the National Institute of Standards and Technology* 105.3 (2000), p. 343-348. DOI : 10.6028/jres.105.035. URL : https://doi.org/10.6028/jres.105.035.

[69] Balazs GEROFI, Rolf RIESEN et Yutaka ISHIKAWA. « Making the Case for Portable MPI Process Pinning ». In : Poster presented at the 25th European MPI Users' Group Meeting, EuroMPI 2018, Barcelona. 2018. URL : https://eurompi2018.bsc.es/sites/default/files/uploaded/EuroMPI2018_paper_40.pdf.

[70] Brice GOGLIN. « Towards generic Communication Mechanisms and better Affinity Management in Clusters of Hierarchical Nodes ». Habilitation à diriger des recherches. Université de Bordeaux, avr. 2014. URL : https://tel.archives-ouvertes.fr/tel-00979512.

[71] Brice GOGLIN, Joshua HURSEY et Jeffrey M. SQUYRES. « Netloc : Towards a Comprehensive View of the HPC System Topology ». In : *43rd International Conference on Parallel Processing Workshops, ICPPW 2014, Minneapolis, MN, USA, September 9-12, 2014*. IEEE Computer Society, 2014, p. 216-225. ISBN : 978-1-4799-5615-9. DOI : 10.1109/ICPPW.2014.38. URL : https://doi.org/10.1109/ICPPW.2014.38.

[72] Brice GOGLIN et Stéphanie MOREAUD. « Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters ». In : *CASS 2011 : The 1st Workshop on Communication Architecture for Scalable Systems, held in conjunction with IPDPS 2011*. Anchorage, AK : IEEE Computer Society Press, mai 2011. DOI : 10.1109/IPDPS.2011.222. URL : http://hal.inria.fr/inria-00566246.

[73] Brice GOGLIN et Stéphanie MOREAUD. « KNEM : a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework ». In : *Journal of Parallel and Distributed Computing (JPDC)* 73.2 (fév. 2013), p. 176-188. DOI : 10.1016/j.jpdc.2012.09.016. URL : http://hal.inria.fr/hal-00731714.

[74] Richard L. GRAHAM et Galen M. SHIPMAN. « MPI Support for Multi-core Architectures : Optimized Shared Memory Collectives ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting, Dublin, Ireland, September 7-10, 2008. Proceedings*. T. 5205. Lecture Notes in Computer Science. Springer, 2008, p. 130-140. ISBN : 978-3-540-87474-4. DOI : 10.1007/978-3-540-87475-1_21. URL : https://doi.org/10.1007/978-3-540-87475-1_21.

[75] William GROPP. « Learning from the Success of MPI ». In : *High Performance Computing - HiPC 2001, 8th International Conference, Hyderabad, India, December, 17-20, 2001, Proceedings*. T. 2228. Lecture Notes in Computer Science. Springer, 2001, p. 81-94. ISBN : 3-540-43009-1. DOI : 10.1007/3-540-45307-5_8. URL : https://doi.org/10.1007/3-540-45307-5_8.

[76] William GROPP. « MPICH2 : A New Start for MPI Implementations ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Linz, Austria, September 29 - October 2, 2002, Proceedings*. T. 2474. Lecture Notes in Computer Science. Springer, 2002, p. 7. ISBN : 3-540-44296-0. DOI : 10.1007/3-540-45825-5_5. URL : https://doi.org/10.1007/3-540-45825-5_5.

[77] William D. GROPP. « Using Node Information to Implement MPI Cartesian Topologies ». In : *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*. ACM, 2018, 18 :1-18 :9. DOI : 10.1145/3236367.3236377. URL : https://doi.org/10.1145/3236367.3236377.

[78] Rakhi GUPTA et Sathish S. VADHIYAR. « Application-oriented adaptive MPI_Bcast for grids ». In : *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI : 10.1109/IPDPS.2006.1639363. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10917.

[79] Samuel K. GUTIERREZ et al. « Accommodating Thread-Level Heterogeneity in Coupled Parallel Applications ». In : *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, p. 469-478. ISBN : 978-1-5386-3914-6. DOI : 10.1109/IPDPS.2017.13. URL : https://doi.org/10.1109/IPDPS.2017.13.

[80] Eduardo H. M. CRUZ, Matthias DIENER et Philippe O. A. NAVAUX. « State-of-the-Art Sharing-Aware Mapping Methods ». In : *Thread and Data Mapping for Multicore Systems : Improving Communication and Memory Accesses*. Springer International Publishing, 2018, p. 35-48. ISBN : 978-3-319-91074-1. DOI : 10.1007/978-3-319-91074-1_4. URL : https://doi.org/10.1007/978-3-319-91074-1_4.

[81] T. HATAZAKI. « Rank Reordering Strategy for MPI Topology Creation Functions ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Sous la dir. de V. ALEXANDROV et J. DONGARRA. T. 1497. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998, p. 188-195. URL : http://dx.doi.org/10.1007/BFb0056575.

[82] Yun HE et al. « Using OpenMP at NERSC ». In : *Présentation invitée à la OpenMPCon 2015*. (*cf. diapositive numéro 20.*) Sept. 2015. URL : https://www.nersc.gov/assets/Uploads/HelenHe-OpenMPCon-2015.pdf.

[83] HEIDI POXON. *MPI for Cray XE/XK7 Systems*. OLCF Workshop. Fév. 2013. URL : https://olcf.ornl.gov/wp-content/uploads/2013/02/MPI_MPT-HP.pdf.

[84] Bruce HENDRICKSON et Robert LELAND. *The Chaco User's Guide : Version 2.0*. Rapp. tech. SAND94–2692. Sandia National Laboratory, 1994.

[85] HLRN. *PlaceMe*. 2011. URL : https://www.hlrn.de/home/view/System2/PlaceMe.

[86] Torsten HOEFLER et Marc SNIR. « Generic Topology Mapping Strategies for Large-Scale Parallel Architectures ». In : *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*. Sous la dir. de David K. LOWENTHAL, Bronis R. de SUPINSKI et Sally A. MCKEE. 2011, p. 75-84. ISBN : 978-1-4503-0102-2. URL : http://doi.acm.org/10.1145/1995896.1995909.

[87] Torsten HOEFLER et Jesper Larsson TRÄFF. « Sparse collective operations for MPI ». In : *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009, p. 1-8. DOI : 10.1109/IPDPS.2009.5160935. URL : https://doi.org/10.1109/IPDPS.2009.5160935.

[88] Torsten HOEFLER et al. « The scalable process topology interface of MPI 2.2 ». In : *Concurrency and Computation : Practice and Experience* 23.4 (2011), p. 293-310. DOI : 10.1002/cpe.1643. URL : https://doi.org/10.1002/cpe.1643.

[89] Torsten HOEFLER et al. « MPI + MPI : a new hybrid approach to parallel programming with MPI plus shared memory ». In : *Computing* 95.12 (2013), p. 1121-1136. DOI : 10.1007/s00607-013-0324-2. URL : https://doi.org/10.1007/s00607-013-0324-2.

[90] Daniel J. HOLMES et al. « MPI Sessions : Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale ». In : *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*. ACM, 2016, p. 121-129. ISBN : 978-1-4503-4234-6. DOI : 10.1145/2966884.2966915. URL : https://doi.org/10.1145/2966884.2966915.

[91] Chao HUANG, Orion Sky LAWLOR et Laxmikant V. KALÉ. « Adaptive MPI ». In : *Languages and Compilers for Parallel Computing, 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003, Revised Papers*. T. 2958. Lecture Notes in Computer Science. Springer, 2003, p. 306-322. DOI : 10.1007/978-3-540-24644-2_20. URL : https://doi.org/10.1007/978-3-540-24644-2_20.

[92] Joshua HURSEY et Jeffrey M. SQUYRES. « Advancing application process affinity experimentation : open MPI's LAMA-based affinity interface ». In : *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*. ACM, 2013, p. 163-168. ISBN : 978-1-4503-1903-4. DOI : 10.1145/2488551.2488603. URL : https://doi.org/10.1145/2488551.2488603.

[93] Joshua HURSEY, Jeffrey M. SQUYRES et Terry DONTJE. « Locality-Aware Parallel Process Mapping for Multi-core HPC Systems ». In : *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE Computer Society, 2011, p. 527-531. ISBN : 978-1-4577-1355-2. DOI : 10.1109/CLUSTER.2011.59. URL : https://doi.org/10.1109/CLUSTER.2011.59.

[94] Parry HUSBANDS et James C. HOE. « MPI-StarT : delivering network performance to numerical applications ». In : *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. San Jose, Californie : IEEE Computer Society, 1998, p. 1-15.

[95] Toshiyuki IMAMURA et al. « An Architecture of Stampi : MPI Library on a Cluster of Parallel Computers ». In : *Proceedings of EuroPVM/MPI2000*. T. 1908. Lecture Notes in Computer Science. Springer-Verlag, 2000, p. 200-207.

[96] INTEL®. *Using MPI Tuner for Intel® MPI Library on Linux* OS*. 2016. URL : https://software.intel.com/en-us/node/610462.

[97] INTEL®. *Intel® MPI Libray for Linux* OS*. 2019. URL : https://software.intel.com/sites/default/files/intelmpi-2019u3-developer-reference-linux.pdf.

[98] Yutaka ISHIKAWA et al. *GridMPI : The Design of a Latency-aware MPI Communication Libary*. Rapp. tech. *(NB : cet article n'est disponible qu'en langue japonaise)*. Cluster Computing Research Center, 2003.

[99] Satoshi ITO, Kazuya GOTO et Kenji ONO. « Automatically Optimized Core Mapping to Subdomains of Domain Decomposition Method on Multicore Parallel Environments ». In : *Computer & Fluids* (avr. 2012). DOI : http://dx.doi.org/10.1016/j.compfluid.2012.04.024.

[100] Emmanuel JEANNOT et Richard SARTORI. *Improving MPI Application Communication Time with an Introspection Monitoring Library*. Research Report 9292. Inria, oct. 2019, p. 23. URL : https://hal.inria.fr/hal-02304515.

[101] Hyun-Wook JIN et al. « LiMIC : Support for High-Performance MPI Intra-Node Communication on Linux Cluster ». In : *Proceedings of the IEEE International Conference on Parallel Processing (ICPP-2005)*. Oslo, Norway : IEEE Computer Society Press, juin 2005.

[102] Hyun-Wook JIN et al. « Lightweight Kernel-Level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems ». In : *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*. Austin, TX, sept. 2007.

[103] A. KAKO et al. « Approximation Algorithms for the Weighted Independent Set Problem ». In : *LNCS*. 3787. SPRINGER-VERLAG, 2005, p. 341-350.

[104] N. KARONIS, B. TOONEN et I. FOSTER. « MPICH-G2 : A Grid-Enabled Implementation of the Message Passing Interface ». In : *Journal of Parallel and Distributed Computing*. T. 63. 5. Mai 2003, p. 551-563.

[105] Nicholas T. KARONIS et al. « Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance ». In : *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*. IEEE Computer Society, 2000, p. 377-384. DOI : 10.1109/IPDPS.2000.846009. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6818.

[106] George KARYPIS et Vipin KUMAR. *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Rapp. tech. 1995.

[107] Thilo KIELMANN et al. « MagPIe : MPI's Collective Communication Operations for Clustered Wide Area Systems ». In : *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99), Atlanta, Georgia, USA, May 4-6, 1999*. ACM, 1999, p. 131-140. ISBN : 1-58113-100-3. DOI : 10.1145/301104.301116. URL : http://dl.acm.org/citation.cfm?id=301104.

[108] Andreas KLEEN. *A NUMA API for LINUX*. Novell, Technical Linux Whitepaper. Novell, Technical Linux Whitepaper, avr. 2005. URL : http://www.novell.com/collateral/4621437/4621437.pdf.

[109] Thorsten KLEINJUNG et al. « A Heterogeneous Computing Environment to Solve the 768-bit RSA Challenge ». In : *Cluster Computing* (2010). URL : http://hal.inria.fr/inria-00535765/en.

[110] Thorsten KLEINJUNG et al. « Factorization of a 768-bit RSA modulus ». In : *CRYPTO 2010*. Sous la dir. de Tal RABIN. T. 6223. Lecture Notes in Computer Science. The original publication is available at www.springerlink.com. Santa Barbara, United States : Springer Verlag, 2010, p. 333-350. DOI : 10.1007/978-3-642-14623-7_18. URL : http://hal.inria.fr/inria-00444693/en.

[111] M. KNESER. « Aufgabe 300 ». In : *Jahresber. Deutsch. Math. -Verein 58* (1955).

[112] Bill KRAMER. « Blue Waters and Resource Management - Now and in the Future ». In : *Présentation à la MoabCon*. (*cf. diapositive numéro 14.*) 2013. URL : https://www.slideshare.net/insideHPC/kramer-post-moabcon041013v1.

[113] Ping LAI, Sayantan SUR et Dhabaleswar K. PANDA. « Designing truly one-sided MPI-2 RMA intra-node communication on multi-core systems ». In : *Proceedings of the International Supercomputing Conference (ISC'10)*. Hamburg, Germany, mai 2010.

[114] Institut Pierre Simon LAPLACE. *Intégration de la parallélisation mixte MPI-OpenMP dans les configurations de l'IPSL*. 2016. URL : https://forge.ipsl.jussieu.fr/igcmg/wiki/IntegrationOpenMP.

[115] Alexey L. LASTOVETSKY et Ravi REDDY. « HMPI : Towards a Message-Passing Library for Heterogeneous Networks of Computers ». In : *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 2003, p. 102. ISBN : 0-7695-1926-1. DOI : 10.1109/IPDPS.2003.1213210. URL : https://doi.org/10.1109/IPDPS.2003.1213210.

[116] Alexey LASTOVETSKY et Ravi REDDY. « HeteroMPI : Towards a message-passing library for heterogeneous networks of computers ». In : *Journal of Parallel and Distributed Computing* 66.2 (2006), p. 197-220. ISSN : 0743-7315. DOI : 10.1016/j.jpdc.2005.08.002. URL : http://www.sciencedirect.com/science/article/pii/S0743731505002042.

[117] Robert LATHAM, Leonardo BAUTISTA-GOMEZ et Pavan BALAJI. « Portable Topology-Aware MPI-I/O ». In : *23rd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2017, Shenzhen, China, December 15-17, 2017*. IEEE Computer Society, 2017, p. 710-719. ISBN : 978-1-5386-2129-5. DOI : 10.1109/ICPADS.2017.00096. URL : https://doi.org/10.1109/ICPADS.2017.00096.

[118] Gunho LEE et al. « Topology-aware Resource Allocation for Data-intensive Workloads ». In : *APSys '10*. New Delhi, India, 2010, p. 1-6. ISBN : 978-1-4503-0195-4. DOI : 10.1145/1851276.1851278. URL : http://doi.acm.org/10.1145/1851276.1851278.

[119] S.-Y. LEE et J. K. AGGARWAL. « A Mapping Strategy for Parallel Processing ». In : *IEEE Trans. Comput.* 36.4 (1987), p. 433-442. ISSN : 0018-9340. DOI : 10.1109/TC.1987.1676925. URL : http://dx.doi.org/10.1109/TC.1987.1676925.

[120] Edgar A. LEÓN. « mpibind : a memory-centric affinity algorithm for hybrid applications ». In : *Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017*. ACM, 2017, p. 262-264. ISBN : 978-1-4503-5335-9. DOI : 10.1145/3132402.3132415. URL : https://doi.org/10.1145/3132402.3132415.

[121] Dongyang LI, Yunlan WANG et Wei ZHU. « Topology-Aware Process Mapping on Clusters Featuring NUMA and Hierarchical Network ». In : *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013*. IEEE, 2013, p. 74-81. ISBN : 978-1-4799-2967-2. DOI : 10.1109/ISPDC.2013.19. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6598363.

[122] Kangkang LI, Maciej MALAWSKI et Jarek NABRZYSKI. « Topology-aware Job Allocation in 3D Torus-based HPC Systems with Hard Job Priority Constraints ». In : *Procedia Computer Science* 108 (2017). International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland, p. 515-524. ISSN : 1877-0509. DOI : 10.1016/j.procs.2017.05.016. URL : http://www.sciencedirect.com/science/article/pii/S1877050917305100.

[123] Kangkang LI, Maciej MALAWSKI et Jarek NABRZYSKI. « Topology-Aware Scheduling on Blue Waters with Proactive Queue Scanning and Migration-Based Job Placement ». In : *Job Scheduling Strategies for Parallel Processing*. Sous la dir. de Narayan DESAI et Walfredo CIRNE. Springer International Publishing, 2017, p. 217-231. ISBN : 978-3-319-61756-5.

[124] Shigang LI, Torsten HOEFLER et Marc SNIR. « NUMA-aware shared-memory collective communication for MPI ». In : *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*. ACM, 2013, p. 85-96. ISBN : 978-1-4503-1910-2. URL : http://dl.acm.org/citation.cfm?id=2493123.

[125] Chuang LIU et al. « Design and Evaluation of a Resource Selection Framework for Grid Applications ». In : *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*. HPDC '02. Washington, DC, USA : IEEE Computer Society, 2002, p. 63-. ISBN : 0-7695-1686-6. URL : http://dl.acm.org/citation.cfm?id=822086.823372.

[126] Jesús M. Álvarez LLORENTE, Juan Carlos Díaz MARTÍN et Juan A. RICO-GALLEGO. « Formal modeling and performance evaluation of a run-time rank remapping technique in Broadcast, Allgather and Allreduce MPI collective operations ». In : *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE, 2017, p. 963-972. ISBN : 978-1-5090-6610-0. DOI : 10.1109/CCGRID.2017.32. URL : https://doi.org/10.1109/CCGRID.2017.32.

[127] Raúl LÓPEZ et Christian PÉREZ. « Improving MPI Support for Applications on Hierarchically Distributed Resources ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*. T. 4757. Lecture Notes in Computer Science. Springer, 2007, p. 187-194. ISBN : 978-3-540-75415-2. DOI : 10.1007/978-3-540-75416-9_29. URL : https://doi.org/10.1007/978-3-540-75416-9_29.

[128] Giorgio LUCARELLI et al. « Contiguity and Locality in Backfilling Scheduling ». In : *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 2015, p. 586-595. ISBN : 978-1-4799-8006-2. DOI : 10.1109/CCGrid.2015.143. URL : https://doi.org/10.1109/CCGrid.2015.143.

[129] Miao LUO et al. « High Performance Design and Implementation of Nemesis Communication Layer for Two-Sided and One-Sided MPI Semantics in MVAPICH2 ». In : *39th International Conference on Parallel Processing, ICPP Workshops 2010, San Diego, California, USA, 13-16 September 2010*. IEEE Computer Society, 2010, p. 377-386. DOI : 10.1109/ICPPW.2010.58. URL : https://doi.org/10.1109/ICPPW.2010.58.

[130] Ewing LUSK et William GROPP. *MPICH Working Note : The Second-Generation ADI for the MPICH Implementation of MPI*. Rapp. tech. Argonne National Laboratory, 1996.

[131] Chao MA et al. « An Approach for Matching Communication Patterns in Parallels Applications ». In : *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*. Rome, Italy : IEEE Computer Society Press, mai 2009.

[132] Teng MA et al. « Locality and Topology Aware Intra-node Communication among Multicore CPUs ». In : *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*. T. 6305. Lecture Notes in Computer Science. Springer, 2010, p. 265-274. ISBN : 978-3-642-15645-8. DOI : 10.1007/978-3-642-15646-5_28. URL : https://doi.org/10.1007/978-3-642-15646-5_28.

[133] Teng MA et al. « Process Distance-Aware Adaptive MPI Collective Communications ». In : *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE Computer Society, 2011, p. 196-204. ISBN : 978-1-4577-1355-2. DOI : 10.1109/CLUSTER.2011.30. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6059523.

[134] Teng MA et al. « HierKNEM : An Adaptive Framework for Kernel-Assisted and Topology-Aware Collective Communications on Many-core Clusters ». In : *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, p. 970-982. ISBN : 978-1-4673-0975-2. DOI : 10.1109/IPDPS.2012.91. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6266782.

[135] Teng MA et al. « Kernel-assisted and topology-aware MPI collective communications on multicore/many-core platforms ». In : *J. Parallel Distrib. Comput.* 73.7 (2013), p. 1000-1010. DOI : 10.1016/j.jpdc.2013.01.015. URL : https://doi.org/10.1016/j.jpdc.2013.01.015.

[136] Amith R. MAMIDALA et al. « Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast ». In : *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006. DOI : 10.1109/IPDPS.2006.1639562. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10917.

[137] K. B. MANWADE et D. B. KULKARNI. « ClustMap : A Topology-Aware MPI Process Placement Algorithm for Multi-core Clusters ». In : *Intelligent Computing and Information and Communication*. Sous la dir. de Subhash BHALLA et al. Singapore : Springer Singapore, 2018, p. 67-76. ISBN : 978-981-10-7245-1.

[138] Motohiko MATSUDA et al. « Efficient MPI Collective Operations for Clusters in Long-and-Fast Networks ». In : *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. IEEE Computer Society, 2006. ISBN : 1-4244-0328-6. DOI : 10.1109/CLUSTR.2006.311848. URL : https://doi.org/10.1109/CLUSTR.2006.311848.

[139] Andrew J. MCPHERSON, Vijay NAGARAJAN et Marcelo CINTRA. « Static Approximation of MPI Communication Graphs for Optimized Process Placement ». In : *Languages and Compilers for Parallel Computing*. Sous la dir. de James BRODMAN et Peng TU. Springer International Publishing, 2015, p. 268-283. ISBN : 978-3-319-17473-0.

[140] J. M. MELLOR-CRUMMEY et M. L. SCOTT. « Algorithms for Scalable Synchronization on Shared-Memeory Multiprocessors ». In : *ACM Computer Architecture News* 9.1 (Février 1991), p. 21-65.

[141] George MICHELOGIANNAKIS et al. « APHiD : Hierarchical Task Placement to Enable a Tapered Fat Tree Topology for Lower Power and Cost in HPC Networks ». In : *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE, 2017, p. 228-237. ISBN : 978-1-5090-6610-0. DOI : 10.1109/CCGRID.2017.33. URL : https://doi.org/10.1109/CCGRID.2017.33.

[142] S. H. MIRSADEGHI et A. AFSAHI. « PTRAM : A Parallel Topology-and Routing-Aware Mapping Framework for Large-Scale HPC Systems ». In : *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Mai 2016, p. 386-396. DOI : 10.1109/IPDPSW.2016.146.

[143] Seyed Hessam MIRSADEGHI et Ahmad AFSAHI. « Topology-Aware Rank Reordering for MPI Collectives ». In : *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, p. 1759-1768. ISBN : 978-1-5090-3682-0. DOI : 10.1109/IPDPSW.2016.139. URL : https://doi.org/10.1109/IPDPSW.2016.139.

[144] G. E. MOORE. « Cramming more components onto integrated circuits ». In : *IEEE Solid-State Circuits Society Newsletter* 11.3 (sept. 2006). Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff, p. 33-35. ISSN : 1098-4232. DOI : 10.1109/N-SSC.2006.4785860.

[145] Stéphanie MOREAUD et al. « Optimizing MPI communication within large multicore nodes with kernel assistance ». In : *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*. IEEE, 2010, p. 1-7. DOI : 10.1109/IPDPSW.2010.5470849. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5465895.

[146] *MPI-Connect*. URL : http://icl.cs.utk.edu/projects/mpi-connect/.

[147] *MPIConnect*. URL : http://www.scali.com.

[148] Raymond NAMYST et Jean-François MÉHAUT. « PM2 : Parallel Multithreaded Machine. A Computing Environment for Distributed Architectures ». In : *Parallel Computing : State-of-the-Art and Perspectives, Proceedings of the conference ParCo 1995, Gent, Belgium, September 1995*. T. 11. Advances in Parallel Computing. Elsevier, 1995, p. 279-285. ISBN : 0-444-82490-1.

[149] Javier NAVARIDAS et al. « Reducing complexity in tree-like computer interconnection networks ». In : *Parallel Computing* 36.2-3 (2010), p. 71-85. URL : http://dx.doi.org/10.1016/j.parco.2009.12.004.

[150] NERSC. *Methods to Check Process and Thread Affinity*. 2019. URL : https://docs.nersc.gov/jobs/affinity/.

[151] Bradford NICHOLS, Dick BUTTLAR et Jacqueline Proulx FARRELL. *Pthreads Programming*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1996. ISBN : 1-56592-115-1.

[152] Christoph NIETHAMMER et Rolf RABENSEIFNER. *Topology aware Cartesian grid mapping with MPI*. Poster at EuroMPI 2018. Barcelona, Spain : High-Performance Computing Center Stuttgart, sept. 2018. URL : https://fs.hlrs.de/projects/par/mpi/EuroMPI2018-Cartesian/.

[153] ORACLE. *Grid Engine*. URL : https://blogs.oracle.com/templedf/entry/topology_aware_scheduling.

[154] Juan M. ORDUÑA, Federico SILLA et José DUATO. « On the development of a communication-aware task mapping technique ». In : *Journal of Systems Architecture* 50.4 (2004), p. 207-220. DOI : 10.1016/j.sysarc.2003.09.002. URL : https://doi.org/10.1016/j.sysarc.2003.09.002.

[155] Scott PAKIN et Avneesh PANT. *VMI 2.0 : A Dynamically Reconfigurable Messaging Layer for Availability, Usability and Management*. 2000.

[156] Jose Antonio PASCUAL, Javier NAVARIDAS et José MIGUEL-ALONSO. « Effects of Topology-Aware Allocation Policies on Scheduling Performance ». In : *Job Scheduling Strategies for Parallel Processing, 14th International Workshop, JSSPP 2009, Rome, Italy, May 29, 2009. Revised Papers*. T. 5798. Lecture Notes in Computer Science. Springer, 2009, p. 138-156. ISBN : 978-3-642-04632-2. DOI : 10.1007/978-3-642-04633-9_8. URL : https://doi.org/10.1007/978-3-642-04633-9_8.

[157] PBSWORKS. *Altaïr PBS Professional*. URL : https://www.pbsworks.com/PBSProduct.aspx?n=Altair-PBS-Professional&c=Overview-and-Capabilities#capabilities.

[158] François PELLEGRINI. « Scotch and PT-Scotch Graph Partitioning Software : An Overview ». In : *Combinatorial Scientific Computing*. Sous la dir. d'Olaf Schenk UWE NAUMANN. Chapman and Hall/CRC, 2012, p. 373-406. DOI : 10.1201/b11644-15. URL : https://hal.inria.fr/hal-00770422.

[159] Marc PÉRACHE, Patrick CARRIBAULT et Hervé JOURDREN. « MPC-MPI : An MPI Implementation Reducing the Overall Memory Consumption ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*. T. 5759. Lecture Notes in Computer Science. Springer, 2009, p. 94-103. ISBN : 978-3-642-03769-6. DOI : 10.1007/978-3-642-03770-2_16. URL : https://doi.org/10.1007/978-3-642-03770-2_16.

[160] Andréa PIACENTINI et al. *A parallel SCRIP interpolation library for OASIS*. Rapp. tech. WN/CMGC/18/34. UMR CERFACS/CNRS No5318, 2018.

[161] Simon PICKARTZ et al. « Enabling hierarchy-aware MPI collectives in dynamically changing topologies ». In : *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*. ACM, 2017, 2 :1-2 :11. ISBN : 978-1-4503-4849-2. DOI : 10.1145/3127024.3127031. URL : http://dl.acm.org/citation.cfm?id=3127024.

[162] Simon PICKARTZ et al. « Revisiting locality-awareness in view of dynamically changing topologies ». In : *Parallel Computing* 77 (2018), p. 1-18. DOI : 10.1016/j.parco.2018.05.004. URL : https://doi.org/10.1016/j.parco.2018.05.004.

[163] *Portable Linux Processor Affinity (PLPA)*. URL : http://www.open-mpi.org/projects/plpa.

[164] Martin POEPPE, Silke SCHUCH et Thomas BEMMERL. « A Message Passing Interface Library for Inhomogeneous Coupled Clusters ». In : *Proceedings of ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop for Communication Architecture in Clusters(CAC 03)*. Nice, France, avr. 2003. URL : http://www.lfbs.rwth-aachen.de/papers/papers_pdf/poeppe_metampich.pdf.

[165] Howard PRITCHARD, Igor GORODETSKY et Darius BUNTINAS. « A uGNI-Based MPICH2 Nemesis Network Module for the Cray XE ». In : *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. T. 6960. Lecture Notes in Computer Science. Springer, 2011, p. 110-119. ISBN : 978-3-642-24448-3. DOI : 10.1007/978-3-642-24449-0_14. URL : https://doi.org/10.1007/978-3-642-24449-0_14.

[166] Peixin QIAO et al. « Joint Effects of Application Communication Pattern, Job Placement and Network Routing on Fat-Tree Systems ». In : *The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 36 :1-36 :10. DOI : 10.1145/3229710.3229747. URL : http://dl.acm.org/citation.cfm?id=3229710.

[167] Jean-Noel QUINTIN, Khalid HASANOV et A. LASTOVETSKY. « Hierarchical Parallel Matrix Multiplication on Large-Scale Distributed Memory Platforms ». In : *42nd International Conference on Parallel Processing (ICPP 2013)*. IEEE. Lyon, France : IEEE, jan. 2013, p. 754-762. DOI : 10.1109/ICPP.2013.89. URL : http://hcl.ucd.ie/system/files/06687414.pdf.

[168] Rolf RABENSEIFNER. *MPI-GLUE : Interoperable high-performance MPI combining different vendor's MPI worlds*. Rapp. tech. Avr. 1998. URL : http://www.hlrs.de/people/rabenseifner.

[169] Rolf RABENSEIFNER, Georg HAGER et Gabriele JOST. « Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes ». In : *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*. 2009, p. 427-436. ISBN : 978-0-7695-3544-9. DOI : 10.1109/PDP.2009.43. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4912899.

[170] Thomas RADKE. *More Message Passing Performance with the Multithreaded MPICH Device*. Rapp. tech. University of Technology Chemnitz, 1997.

[171] Rajesh RAMAN, Miron LIVNY et Marvin SOLOMON. « Matchmaking : Distributed Resource Management for High Throughput Computing ». In : *HPDC'7*. Chicago, IL, juil. 1998, p. 28-31.

[172] Mohammad J. RASHTI et al. « Multi-core and Network Aware MPI Topology Functions ». In : *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*. T. 6960. Lecture Notes in Computer Science. Springer, 2011, p. 50-60. ISBN : 978-3-642-24448-3. DOI : 10.1007/978-3-642-24449-0_8. URL : https://doi.org/10.1007/978-3-642-24449-0_8.

[173] E. RODRIGUES et al. « Multicore aware process mapping and its impact on communication overhead of parallel applications ». In : *Proceedings of the IEEE Symposium on Computers and Communications*. Juil. 2009, p. 811-817. URL : http://dx.doi.org/10.1109/ISCC.2009.5202271.

[174] Arnold L. ROSENBERG. « Issues in the Study of Graph Embeddings ». In : *WG '80 : Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science*. London, UK, 1981, p. 150-176. ISBN : 3-540-10291-4.

[175] SAMUEL THIBAULT, RAYMOND NAMYST AND PIERRE-ANDRÉ WACRENIER. « Building Portable Thread Schedulers for Hierarchical Multiprocessors : the BubbleSched Framework ». In : *EuroPar*. ACM. Rennes,France, août 2007. URL : http://hal.inria.fr/inria-00154506.

[176] Cipriano A. SANTOS et al. « DSOM 2004, Davis, CA, USA, November 15-17 ». In : 2004. Chap. Policy-Based Resource Assignment in Utility Computing Environments, p. 100-111. ISBN : 978-3-540-30184-4. DOI : 10.1007/978-3-540-30184-4_9. URL : http://dx.doi.org/10.1007/978-3-540-30184-4_9.

[177] Kirk SCHLOEGEL, George KARYPIS et Vipin KUMAR. « Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning (Distinguished Paper) ». In : *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. Euro-Par '00. London, UK, UK : Springer-Verlag, 2000, p. 296-310. ISBN : 3-540-67956-1. URL : http://dl.acm.org/citation.cfm?id=646665.698944.

[178] Christian SCHULZ et Jesper Larsson TRÄFF. « Better Process Mapping and Sparse Quadratic Assignment ». In : *16th International Symposium on Experimental Algorithms (SEA 2017)*. Sous la dir. de Costas S. ILIOPOULOS et al. T. 75. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 4 :1-4 :15. ISBN : 978-3-95977-036-1. DOI : 10.4230/LIPIcs.SEA.2017.4. URL : http://drops.dagstuhl.de/opus/volltexte/2017/7603.

[179] Sangmin SEO et al. « Argobots : A Lightweight Low-Level Threading and Tasking Framework ». In : *IEEE Trans. Parallel Distrib. Syst.* 29.3 (2018), p. 512-526. DOI : 10.1109/TPDS.2017.2766062. URL : https://doi.org/10.1109/TPDS.2017.2766062.

[180] John SHALF, Sudip DOSANJH et John MORRISON. « Exascale Computing Technology Challenges ». In : *High Performance Computing for Computational Science – VECPAR 2010*. Sous la dir. de José M. Laginha M. PALMA et al. Springer Berlin Heidelberg, 2011, p. 1-25. ISBN : 978-3-642-19328-6.

[181] Horst D. SIMON et Shang-Hua TENG. « How Good is Recursive Bisection ? » In : *SIAM J. Sci. Comput.* 18 (5 sept. 1997), p. 1436-1445. DOI : 10.1137/S1064827593255135. URL : http://portal.acm.org/citation.cfm?id=271594.271620.

[182] Brian E. SMITH et Brett BODE. « Performance Effects of Node Mappings on the IBM BlueGene/L Machine ». In : *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005, Proceedings*. T. 3648. Lecture Notes in Computer Science. Springer, 2005, p. 1005-1013. ISBN : 3-540-28700-0. URL : http://dx.doi.org/10.1007/11549468_110.

[183] Lorna SMITH et Mark BULL. « Development of mixed mode MPI / OpenMP applications ». In : *Scientific Programming* 9.2-3 (2001), p. 83-98. DOI : 10.1155/2001/450503. URL : https://doi.org/10.1155/2001/450503.

[184] O. SONMEZ, H.H. MOHAMED et D.H.J. EPEMA. « Communication-Aware Job Placement Policies for the KOALA Grid Scheduler ». In : *Proc. of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*. IEEE Computer Science, déc. 2006, p. 79-86. ISBN : 0-7695-2734-5.

[185] Mohsen SORYANI, Morteza ANALOUI et Ghobad ZARRINCHIAN. « A Novel Process Mapping Strategy in Clustered Environments ». In : *CoRR* abs/1207.2878 (2012). arXiv : 1207.2878. URL : http://arxiv.org/abs/1207.2878.

[186] Mohsen SORYANI, Morteza ANALOUI et Ghobad ZARRINCHIAN. « Improving inter-node communications in multi-core clusters using a contention-free process mapping algorithm ». In : *The Journal of Supercomputing* 66.1 (2013), p. 488-513. DOI : 10.1007/s11227-013-0918-7. URL : https://doi.org/10.1007/s11227-013-0918-7.

[187] Jeffrey M. SQUYRES et Andrew LUMSDAINE. « A Component Architecture for LAM/MPI ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*. T. 2840. Lecture Notes in Computer Science. Springer, 2003, p. 379-387. ISBN : 3-540-20149-1. DOI : 10.1007/978-3-540-39924-7_52. URL : https://doi.org/10.1007/978-3-540-39924-7_52.

[188] Craig S. STEELE. « Placement of communicating processes on multiprocessors networks ». Master Thesis. California Institute of Technology, 1985. URL : http://resolver.caltech.edu/CaltechTHESIS:04122012-143033166.

[189] H. SUBRAMONI et al. « Design of a Scalable Infiniband Topology Service to Enable Network-Topology-Aware Placement of Processes ». In : *Proceedings of the 2012 ACM/IEEE conference on Supercomputing (CDROM)*. Salt Lake City, Utah, United States : IEEE Computer Society, 2012, p. 12.

[190] C.D. SUDHEER et A. SRINIVASAN. « Optimization of the hop-byte metric for effective topology aware mapping ». In : *High Performance Computing (HiPC), 2012 19th International Conference on*. 2012, p. 1-9. DOI : 10.1109/HiPC.2012.6507513.

[191] Hong TANG et Tao YANG. « Optimizing threaded MPI execution on SMP clusters ». In : *Proceedings of the 15th international conference on Supercomputing*. Sorrento, Italy : ACM Press, 2001, p. 381-392. ISBN : 1-58113-410-X. DOI : http://doi.acm.org/10.1145/377792.377895.

[192] Kenjiro TAURA et Andrew A. CHIEN. « A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources ». In : *9th Heterogeneous Computing Workshop, HCW 2000, Cancun, Mexico, May 1, 2000*. IEEE Computer Society, 2000, p. 102-115. ISBN : 0-7695-0556-2. DOI : 10.1109/HCW.2000.843736. URL : https://doi.org/10.1109/HCW.2000.843736.

[193] François TESSIER. « Placement d'applications parallèles en fonction de l'affinité et de la topologie ». Thèse de Doctorat. Université de Bordeaux, jan. 2015.

[194] Jesper Larsson TRÄFF. « Implementing the MPI process topology mechanism ». In : *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*. IEEE Computer Society, 2002, 40 :1-40 :14. ISBN : 0-7695-1524-X. DOI : 10.1109/SC.2002.10045. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=10618.

[195] Jesper Larsson TRÄFF. « Improved MPI All-to-all Communication on a Giganet SMP Cluster ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Linz, Austria, September 29 - October 2, 2002, Proceedings*. T. 2474. Lecture Notes in Computer Science. Springer, 2002, p. 392-400. ISBN : 3-540-44296-0. DOI : 10.1007/3-540-45825-5_57. URL : https://doi.org/10.1007/3-540-45825-5_57.

[196] Jesper Larsson TRÄFF. « SMP-Aware Message Passing Programming ». In : *Eighth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03), April 22-22, 2003, Nice, France*. IEEE Computer Society, 2003, p. 56-65. ISBN : 0-7695-1880-X. DOI : 10.1109/HIPS.2003.1196495. URL : http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8525.

[197] Jesper Larsson TRÄFF. « Direct graph *k*-partitioning with a Kernighan-Lin like heuristic ». In : *Oper. Res. Lett.* 34.6 (2006), p. 621-629. DOI : 10.1016/j.orl.2005.10.003. URL : https://doi.org/10.1016/j.orl.2005.10.003.

[198] Jesper Larsson TRÄFF et Antoine ROUGIER. « MPI Collectives and Datatypes for Hierarchical All-to-all Communication ». In : *21st European MPI Users' Group Meeting, EuroMPI/ASIA '14, Kyoto, Japan - September 09 - 12, 2014*. ACM, 2014, p. 27. ISBN : 978-1-4503-2875-3. DOI : 10.1145/2642769.2642770. URL : https://doi.org/10.1145/2642769.2642770.

[199] François TRAHAY et al. « A multithreaded communication engine for multicore architectures ». In : *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008, p. 1-7. DOI : 10.1109/IPDPS.2008.4536139. URL : https://doi.org/10.1109/IPDPS.2008.4536139.

[200] Dave TURNER et al. « Integrating New Capabilities into NetPIPE ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*. T. 2840. Lecture Notes in Computer Science. Springer, 2003, p. 37-44. DOI : 10.1007/978-3-540-39924-7_10. URL : https://doi.org/10.1007/978-3-540-39924-7%5C_10.

[201] Geoffroy VALLÉE et David BERNHOLDT. « Improving Support of MPI+OpenMP Applications ». In : Poster presented at the 25th European MPI Users' Group Meeting, EuroMPI 2018, Barcelona. 2018. URL : https://eurompi2018.bsc.es/sites/default/files/uploaded/eurompi2018-paper-vallee.pdf.

[202] Paula L. VAUGHAN et al. « Migrating from PVM to MPI, part I : the Unify System ». In : *5th Symposium on the Frontiers of Massively Parallel Computation*. Sous la dir. d'IEEE Computer Society Technical Committee on COMPUTER ARCHITECTURE. 12. McLean, Virginie : IEEE Computer Society Press, fév. 1995, p. 488-495.

[203] Vishwanath VENKATESAN et al. « Optimized process placement for collective I/O operations ». In : *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*. ACM, 2013, p. 31-36. ISBN : 978-1-4503-1903-4. DOI : 10.1145/2488551.2488567. URL : https://doi.org/10.1145/2488551.2488567.

[204] Joshua T. VOGELSTEIN et al. « Fast Approximate Quadratic Programming for Graph Matching ». In : *PLoS One* 10.4 (2015). DOI : 10.1371/journal.pone.0121002.

[205] Nagavijayalakshmi VYDYANATHAN et al. « Locality Conscious Processor Allocation and Scheduling for Mixed Parallel Applications ». In : *Proceedings of the 2006 IEEE International Conference on Cluster Computing, September 25-28, 2006, Barcelona, Spain*. IEEE Computer Society, 2006. DOI : 10.1109/CLUSTR.2006.311861. URL : https://doi.org/10.1109/CLUSTR.2006.311861.

[206] Scott WALLACE. *Minimising Communication Costs on a SMP Cluster using Process Placement*. Mémoire de Master. University of Edingburgh, 2005.

[207] C. WALSHAW et M. CROSS. « JOSTLE : Parallel Multilevel Graph-Partitioning Software – An Overview ». In : *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Sous la dir. de F. MAGOULES. Civil-Comp Ltd., 2007, p. 27-58. ISBN : 978-1-874672-29-6.

[208] Jingjin WU, Xuanxing XIONG et Zhiling LAN. « Hierarchical task mapping for parallel applications on supercomputers ». In : *The Journal of Supercomputing* 71.5 (2015), p. 1776-1802.

[209] Jingjin WU, Xuanxing XIONG et Zhiling LAN. « Hierarchical task mapping for parallel applications on supercomputers ». In : *The Journal of Supercomputing* 71.5 (2015), p. 1776-1802. ISSN : 1573-0484. DOI : 10.1007/s11227-014-1324-5. URL : https://doi.org/10.1007/s11227-014-1324-5.

[210] *YAMPII*. http://www.il.is.s.u-tokyo.ac.jp/yampii/.

[211] Xu YANG et al. « Balancing job performance with system performance via locality-aware scheduling on torus-connected systems ». In : *Cluster'2014*. 2014, p. 140-148.

[212] AndyB. YOO, MorrisA. JETTE et Mark GRONDONA. « SLURM : Simple Linux Utility for Resource Management ». In : *Job Scheduling Strategies for Parallel Processing*. 2003, p. 44-60. ISBN : 978-3-540-20405-3. DOI : 10.1007/10968987_3. URL : http://dx.doi.org/10.1007/10968987_3.

[213] H. YU, I-H. CHUNG et J. E. MOREIRA. « Blue Gene System Software - Topology Mapping for Blue Gene/L Supercomputer ». In : *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. ACM Press, 2006. ISBN : 0-7695-2700-0. URL : http://doi.acm.org/10.1145/1188455.1188576.

[214] Ghobad ZARRINCHIAN, Mohsen SORYANI et Morteza ANALOUI. « A New Process Placement Algorithm in Multi-core Clusters Aimed to Reducing Network Interface Contention ». In : *Advances in Computer Science, Engineering & Applications*. Sous la dir. de David C. WYLD, Jan ZIZKA et Dhinaharan NAGAMALAI. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 1041-1050. ISBN : 978-3-642-30111-7.

[215] Jidong ZHAI, Wenguang CHEN et Weimin ZHENG. « PHANTOM : predicting performance of parallel applications on large-scale parallel machines using a single node ». In : *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*. ACM, 2010, p. 305-314. ISBN : 978-1-60558-877-3. URL : http://doi.acm.org/10.1145/1693453.1693493.

[216] Jidong ZHAI et al. « FACT : fast communication trace collection for parallel applications through program slicing ». In : *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. ACM, 2009. ISBN : 978-1-60558-744-8. URL : http://doi.acm.org/10.1145/1654059.1654087.

[217] Jidong ZHAI et al. « CYPRESS : Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression ». In : *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE Computer Society, 2014, p. 143-153. ISBN : 978-1-4799-5500-8. DOI : 10.1109/SC.2014.17. URL : https://doi.org/10.1109/SC.2014.17.

[218] Jin ZHANG et al. « Process Mapping for MPI Collective Communications ». In : *Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings*. T. 5704. Lecture Notes in Computer Science. Springer, 2009, p. 81-92. ISBN : 978-3-642-03868-6. URL : http://dx.doi.org/10.1007/978-3-642-03869-3_11.

[219] Hao ZHU et al. « Hierarchical Collectives in MPICH2 ». In : *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*. T. 5759. Lecture Notes in Computer Science. Springer, 2009, p. 325-326. ISBN : 978-3-642-03769-6. DOI : 10.1007/978-3-642-03770-2_41. URL : https://doi.org/10.1007/978-3-642-03770-2.