Placement d'applications parallèles en fonction de l'affinité et de la topologie

François Tessier

université de BORDEAUX

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

SPÉCIALITÉ : INFORMATIQUE

par François TESSIER

Placement d'applications parallèles en fonction de l'affinité et de la topologie

Après avis de :

Franck	CAPPELLO	Senior computer scientist, Argonne National Lab.	Rapporteur
Christian	PEREZ	Directeur de recherche, Inria	Rapporteur

Présentée et soutenue publiquement le 26 Janvier 2015

Devant la commission d'examen formée de :

Emmanuel	JEANNOT	Directeur de recherche, Inria	Directeur de thèse
Guillaume	MERCIER	Maître de Conférence, IPB	Encadrant de thèse
François	PELLEGRINI	Professeur, Université de Bordeaux	Examinateur
Franck	CAPPELLO	Senior computer scientist, Argonne National Lab.	Rapporteur
Christian	PEREZ	Directeur de recherche, Inria	Rapporteur
Laxmikant V.	KALE	Professeur, Université de l'Illinois	Examinateur

Résumé

Placement d'applications parallèles en fonction de l'affinité et de la topologie.

Mots clés : Calcul haute performance, parallélisme, localité, affinité, topologie, placement, équilibrage de charge.

La simulation numérique est un des piliers des Sciences et de l'industrie. La simulation météorologique, la cosmologie ou encore la modélisation du cœur humain sont autant de domaines dont les besoins en puissance de calcul sont sans cesse croissants. Dès lors, comment passer ces applications à l'échelle ? La parallélisation et les supercalculateurs massivement parallèles sont les seuls moyens d'y parvenir. Néanmoins, il y a un prix à payer compte tenu des topologies matérielles de plus en plus complexes, tant en terme de réseau que de hiérarchie mémoire. La question de la localité des données devient ainsi centrale : comment réduire la distance entre une entité logicielle et les données auxquelles elle doit accéder ? Le placement d'applications est un des leviers permettant de traiter ce problème. Dans cette thèse, nous présentons l'algorithme de placement TreeMatch et ses applications dans le cadre du placement statique, c'est-à-dire au lancement de l'application, et du placement dynamique. Pour cette seconde approche, nous proposons la prise en compte de la localité des données dans le cadre d'un algorithme d'équilibrage de charge. Les différentes approches abordées sont validées par des expériences réalisées tant sur des codes d'évaluation de performances que sur des applications réelles.

Abstract

Placement of parallel applications according to the topology and the affinity.

Keywords : High performance computing, parallelism, locality, affinity, topology, placement, load balancing.

Computer simulation is one of the pillars of Sciences and industry. Climate simulation, cosmology, or heart modeling are all areas in which computing power needs are constantly growing. Thus, how to scale these applications ? Parallelization and massively parallel supercomputers are the only ways to do achieve. Nevertheless, there is a price to pay considering the hardware topologies incessantly complex, both in terms of network and memory hierarchy. The issue of data locality becomes central : how to reduce the distance between a processing entity and data to which it needs to access ? Application placement is one of the levers to address this problem. In this thesis, we present the TreeMatch algorithm and its application for static mapping, that is to say at the lauchtime of the application, and the dynamic placement. For this second approach, we propose the awareness of data locality within a load balancing algorithm. The different approaches discussed are validated by experiments both on benchmarking codes and on real applications.

Remerciements

Je vais bien entendu tâcher de n'oublier personne. Si par hasard vous me connaissez et que vous n'êtes pas cité, considérez que ma gratitude envers vous est sincère et éternelle.

Ah ! Une petite chose avant de débuter la cérémonie des accolades : si lire ces remerciements vous intéresse mais quand même pas autant que la Science qui constitue le socle de ce travail parce que la Science c'est la classe, une version TL ;DR [1] sous forme de nuage de mot de mauvais goût façon web 2.0 du XXIème siècle est disponible en figure 1.

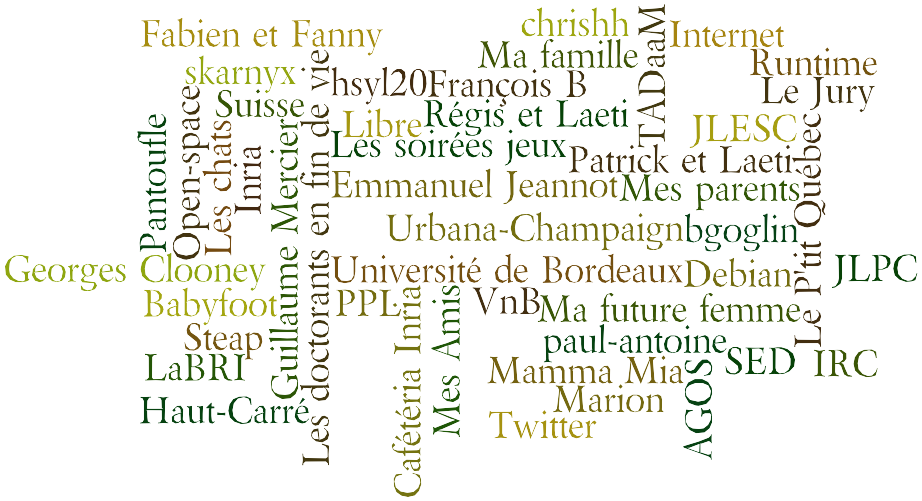


Figure 1 – La version TL ;DR des remerciements façon nuage de mot de mauvais goût, web 2.0 et XXIème siècle.

Mes premiers remerciements vont tout naturellement à mes encadrants, Emmanuel Jeannot et Guillaume Mercier. Je souhaite à tout doctorant d'avoir des encadrants comme eux. C'est sans doute le meilleur compliment que je puisse leur faire. Qui plus est, nos relations ont dépassé le simple cadre "professionnel" puisqu'ils sont devenus des amis, et compagnons de soirées jeux ! Ils ont d'ailleurs de sacrés progrès à faire dans cette discipline. On ne peut pas exceller dans tous les domaines...

1. Too Long ; Didn't Read

Je tiens à remercier également les rapporteurs de ce manuscrit, Franck Cappello et Christian Perez, et les membres venus compléter mon jury de thèse, Sanjay Kale et François Pellegrini, avec qui les échanges ont été passionnants. Nous nous recroiserons sans doute prochainement.

Je voudrais également remercier les différents laboratoires et institutions qui m'ont permis de réaliser cette thèse dans les meilleures conditions : l'Université de Bordeaux, le LaBRI et Inria Bordeaux. C'est dans ce dernier laboratoire que j'ai élu domicile pour une année d'ingénieur, puis trois ans de thèse et enfin quelques mois de postdoc. Je ne pourrai malheureusement pas citer tous les gens au sein de l'institut qui ont œuvré et qui œuvrent toujours à rendre les conditions de travail excellentes. Il y a néanmoins deux personnes que je souhaiterais nommer : Sylvie Embolla et Flavie Tregan qui se sont succédé au poste d'assistante d'équipe pour y faire un travail remarquable. Un grand merci aussi à Joint-Lab et au Parallel Programming Laboratory qui ont apporté un volet international à ce travail.

Ces quelques années de thèse ont également été grandement facilitées par les membres passés et présents de l'équipe Runtime et de ses deux équipes filles : TADaaM et STORM. En particulier, je voudrais remercier Raymond Namyst pour la gestion irréprochable de l'équipe Runtime et sa bonne humeur quotidienne, Brice "awesome" Goglin pour ses compétences techniques et ses qualités humaines bien que sérieusement entâchées par son origine Ardennaise (/yifi/), Denis Barthou pour ses avis éclairés lors de mes répétitions de soutenance et son armada de vaisseaux à X-Wing ou encore Samuel Thibault pour sa pédagogie et son engagement militant. D'une manière générale, un grand merci à toute l'équipe Runtime.

Plus nous avançons dans les remerciements et plus nous convergeons vers l'open space. C'est inévitable puisque c'est là que tout s'est joué. C'est le cœur de réseau, le noyau terrestre, l'origine du monde (ça va beaucoup trop loin...). Je vais sans doute oublier des gens, mais voilà une liste que j'espère la plus exhaustive possible. Merci donc à Bertrand, friand de tripes aux papillotes, sportif du dimanche et des autres jours de la semaine ; Sylvain le polémiste militant et base de données du Top50 ; Paul-Antoine, polémiste et militant aussi, mais pas tout à fait pareil parce qu'il fait de la CO [2] comme en CP [3] ; Cyril R., créationniste inavoué et défenseur du travail libre et non contraint, qui n'était pas physiquement dans l'open space mais toujours plus ou moins là quand même ; Cyril B. l'homme trop bien sapé et scripté ; Marc qui ne prend toujours pas son vélo pour venir et participe activement au trou de la couche d'ozone ; Chris mi-baroudeur mi-swag et compagnon de baby à la scène comme à la ville ; Nicolas qui remplit ses t-shirts ; Adèle qui reprend le flambeau de TreeMatch et qui n'aime pas bien Cyril R. ; Terry-bague-au-doigt et ses excursions en Italie ; Sam dont les progrès au baby sont inversement proportionnels à sa consommation de Whisky ; James et ses sujets de débat enflammés ; Thomas et ses conseils de soutenance avisés ; et tous les autres, ceux qui sont passés, ceux qui sont là et

2. Course d'orientation
3. Classe préparatoire

ceux qui viendront.

Merci également à mes proches, famille et amis : Mes parents, mes frères et sœurs, ma tante, ma famille de Suisse, Grand-Maman et Grand-Papa en tête, ma belle-famille, Pantoufle (Paupiette ?). Mais également à (attention, ça va aller vite) Fabien, Fanny, Patrick, Laeti, Régis, Laeti encore, François B., Maxime[2], Charly, Bulle, et tous les autres ! Vous êtes géniaux.

Merci au babyfoot, à IRC, à Debian, à Aquilenet et au VnB. Ils se reconnaîtront.

Remercier, c'est bien. Ne pas remercier, ça a son charme aussi. Je tiens donc à ne surtout pas remercier les gens qui utilisent la frontale de PlaFRIM pour lancer des expériences ; ceux qui utilisent la numérotation physique des cœurs et, ça va avec, ceux qui n'utilisent pas hwloc ; et enfin ceux, et c'est plus politique, qui portent atteinte chaque jour à nos libertés fondamentales en votant des lois dangereuses légalisant la surveillance généralisée (Loi Renseignement, Loi Terrorisme, etc.).

Enfin, le meilleur pour la fin. Ou devrais-je dire "la meilleure pour la fin". Un immense merci à ma fiancée, Marion, sans qui je ne pourrais réussir ni professionnellement, ni personnellement. Son aide est précieuse et inestimable.

Table des matières

Table des figures

Liste des tableaux

Introduction

Il y a douze ans, J.M. Tendler et al. publiaient « POWER4 system microarchitecture » [75], un article de journal détaillant pour la première fois l'architecture d'un processeur multi-cœur. Depuis, la multiplication des unités de calcul est devenue la norme et avec elle la complexité des architectures s'est accrue. Cette évolution dans la conception des microprocesseurs a répondu à des contraintes de dissipation thermique empêchant la montée en fréquence des processeurs. Qui plus est, la demande en performance des applications parallèles est restée constamment croissante incitant les constructeurs à trouver des solutions. Il n'en demeure pas moins que la conception et l'exécution de ces applications parallèles sur ces nouvelles architectures n'est pas une tâche triviale et un fossé s'est creusé entre les performances matérielles et les performances des applications. Pour réduire cet écart, trois pistes principales peuvent être dégagées. Premièrement, une solution est à trouver du côté de l'écriture des applications. Toutefois, il n'existe pas de consensus sur la façon adéquate d'exprimer le parallélisme au sein du code et le corpus d'applications existantes est trop volumineux pour imaginer sa réécriture. Deuxièmement, un objectif concerne le support d'exécution. Optimiser les bibliothèques utilisées et définir dans le support d'exécution les moyens permettant de coller au plus près de la machine sont des pistes qui ne font là encore pas l'unanimité quant à leur mise en œuvre. En effet, le rôle du support d'exécution dans la gestion de la topologie est un sujet très débattu. Qui plus est, de nombreuses difficultés apparaissent lorsqu'il s'agit de les implémenter (architectures diverses, paradigmes peu ou pas interopérables, etc). Le dernier levier qui peut être actionné repose sur la façon dont les applications sont exécutées. Bien entendu, agir à ce niveau est complémentaire des autres méthodes, mais il y a là une réelle marge d'intervention. Ce travail d'amélioration des interactions entre les applications et l'*écosystème* au sein duquel elles vont être amenées à s'exécuter est une problématique qui a gagné en importance au cours des dernières années.

L'amélioration de ces interactions repose sur plusieurs constats : d'une part, les applications possèdent un comportement qui les caractérise, que ce soit en termes d'accès aux données, à la mémoire ou encore en terme de schéma de communication. D'autre part, une partie essentielle de l'écosystème, à savoir l'architecture matérielle cible, possède également des caractéristiques propres, telles que le nombre de CPU, la topologie du réseau d'inter-

connexion, la disposition des bancs mémoire, etc. Qui plus est, ces caractéristiques ne sont pas nécessairement connues à la conception de l'application. Une idée intuitive est alors de mettre en correspondance ces caractéristiques des applications et du matériel sous-jacent, de façon à exploiter ce dernier au mieux pour une application donnée. Plusieurs aspects peuvent être pris en considération : l'ordonnancement des tâches, l'allocation des ressources matérielles, la gestion des entrées/sorties, l'équilibrage de charge ou encore la localité des données. C'est sur ce dernier point que nous avons concentré nos efforts, avec le but de rapprocher physiquement les données des entités logicielles les utilisant afin d'améliorer les temps d'accès et de communication des applications. Dans le contexte des architectures multi-cœurs, cela revient notamment à minimiser les effets induits par la disposition des bancs mémoire (effets NUMA) au sein de la machine. Cette minimisation peut s'effectuer en répartissant les différents processus de l'application parallèle sur les différents coeurs disponibles. On parle alors de *placement de processus*. Si le principe est simple, sa réalisation l'est moins. Tenir compte des caractéristiques de l'architecture et de l'affinité des entités logicielles pour proposer un placement satisfaisant soulève des problèmes importants, tant d'un point de vue technique qu'algorithmique. Il s'agit alors de trouver des méthodes adaptées permettant de déterminer un placement efficace et de l'appliquer à plusieurs paradigmes. Appliquer un placement statique, c'est-à-dire au lancement de l'application sans qu'il n'évolue durant l'exécution, ou combiner les bénéfices du placement avec de l'équilibrage de charge en cours d'exécution, voici les objectifs que nous nous sommes fixés dans cette thèse.

Nous présenterons tout d'abord dans la suite de ce manuscrit une analyse du contexte de façon à dégager une problématique générale. Dans une seconde partie, nous exposerons nos travaux sur le placement statique appuyés par une importante série d'expériences. Une troisième partie présentera deux algorithmes d'équilibrage de charge prenant en compte les contraintes de localité que nous avons là encore évalués, notamment sur des exécutions à grande échelle. Enfin, un bilan de ces travaux et des perspectives qu'ils offrent concluront ce document.

Sommaire

Chapitre 1

Contexte et problématique

La simulation numérique est un domaine en pleine expansion. Qu'elle touche la science ou encore l'économie, elle apporte une solution en permettant de corroborer des résultats théoriques grâce aux outils mathématiques et informatiques. En effet, les expériences réelles sont très souvent impossibles à réaliser, pour des raisons aussi variées que le temps, le coût ou les possibilités matérielles. La simulation numérique propose de passer outre ces limitations. Pour ce faire, les besoins en puissance de calcul sont très importants et ne cessent de croître. Par exemple, l'organisme français de météorologie Météo France utilise à ce jour une infrastructure pouvant fournir jusqu'à 1 Pétaflops afin d'effectuer ses prévisions. Depuis 1992, la puissance de calcul théorique de Météo France a dû être multipliée par 500 000. Ces besoins entraînent d'ailleurs d'importantes questions de passage à l'échelle d'une part et d'écologie d'autre part, l'augmentation des performances allant de pair avec un accroissement de la consommation énergétique.

Toujours est-il que les constructeurs ont répondu à cette demande en proposant des plate-formes de plus en plus puissantes et des architectures de plus en plus complexes. Les méthodes ont cependant évolué. La stratégie ne consiste plus à augmenter continuellement la fréquence des processeurs, comme ce fut le cas il y a une dizaine d'années. La dissipation thermique ayant atteint ses limites, c'est la multiplication des cœurs de calcul qui prévaut. La figure 1.1 présente la moyenne annuelle du nombre de cœurs de calcul et leur fréquence sur les supercalculateurs du Top500 depuis 2000. Alors que la fréquence moyenne des processeurs s'est stabilisée aux alentours de 2,5 GHz depuis dix ans, le nombre de cœurs présents sur les plate-formes suit une courbe exponentielle. Pour ne citer qu'un exemple

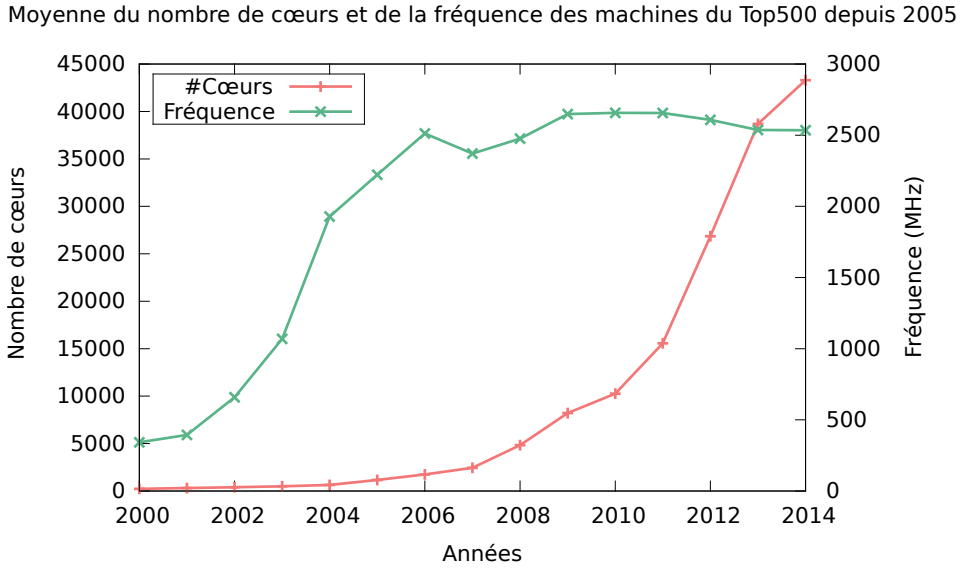


Figure 1.1 – Évolution annuelle du nombre moyen de cœurs et de leur fréquence sur les supercalculateurs du Top500 depuis 2000 (source : top500.org)

parmi d'autres, le supercalculateur Blue Waters [53] contient près de 400 000 cœurs de calcul, répartis sur quelques 27 000 nœuds, pour une performance crête atteignant les 13,34 Petaflops.

Cette évolution s'accompagne de topologies complexes, en particulier en ce qui concerne la mémoire : les différents types de mémoires (RAM, cache) sont hiérarchisés et dépendent de bus de communication différents, ce qui conduit à des effets NUMA (*Non Uniform Memory Access*). Tirer pleinement parti de ces architectures est un réel défi et demande d'adapter les briques logicielles existantes. Une piste concerne la notion de localité des données. En effet, cette structuration de la mémoire et la multiplication des nœuds et des cœurs de calcul augmentent la distance entre les entités logicielles d'une application parallèle et les données auxquelles elles doivent accéder. Sur la longue route vers l'Exascale, cet aspect est déterminant.

1.1 La localité des données

La localité des données se définit comme la distance entre une entité de traitement logicielle (processus lourd, fil d'exécution, tâche, etc) et les données auxquelles elle doit accéder. Plus ces données sont distantes, plus le coût pour l'entité logicielle qui souhaite les atteindre sera important. Ce coût peut s'évaluer selon plusieurs critères bien qu'au final, c'est sur le temps d'exécution de l'application qu'il sera répercuté. Notons que la notion

de localité ne sous-entend pas uniquement l'accès direct à une donnée en mémoire. Elle implique également les échanges entre entités logicielles : l'accès à des données distantes peut se voir comme un échange entre l'entité locale et une entité distante.

Afin d'illustrer ce propos, la figure 1.2 présente des mesures de bande passante sur différents niveaux de mémoire obtenues grâce à l'outil MBENCH [66]. Ces mesures ont été réalisées sur une architecture Intel Xeon X5550 comptant 2 processeurs de 4 cœurs. Chaque groupe de 4 cœurs partage 8 Mo de cache L3 et 12 Go de RAM. Sur cette figure, la courbe intitulée *cache miss* montre la bande passante obtenue lors de l'accès à une donnée après qu'elle a été lue par un voisin distant, générant ainsi un défaut de cache. La courbe intitulée *cache hit*, quant à elle, montre les performances d'accès sur une donnée dans le cache ou la mémoire (en fonction de la taille de cette donnée). Ce graphique illustre bien l'importance de la localité des données puisque la différence de bande passante pour l'accès à une information est très importante selon que la donnée est directement accessible ou non. Les différents paliers de la courbe verte correspondent aux différents niveaux de cache. Une donnée proche, c'est-à-dire à une distance courte dans la hiérarchie mémoire, sera accédée beaucoup plus rapidement. Sur un nœud Intel Xeon X5550 à deux processeurs de quatre cœurs chacun, la différence de bande passante entre l'accès au cache L1 et l'accès à la mémoire est de l'ordre de 150%. Dans la course à la performance, la multiplication des unités de calcul et des hiérarchies mémoire fait de la localité des données un critère clé pour tirer profit au maximum des performances du matériel. Placer les entités logicielles qui échangent ou partagent beaucoup de données proches les unes des autres s'avère bénéfique pour les coûts de communication et d'accès.

Prendre en compte la localité des données implique trois conditions. Premièrement, il faut arriver à fournir une mesure de la localité. La localité peut-elle être considérée en fonction des coûts d'accès mémoire ? En fonction du volume des communications entre deux processus ? D'une manière générale, nous parlerons d'affinité pour exprimer l'ensemble des *métriques* permettant de préciser la dépendance entre deux entités de traitement logicielles. Deuxièmement, comprendre l'architecture matérielle qui accueille l'application est là aussi une condition nécessaire pour prendre en compte la localité. Enfin, il est important que le paradigme de programmation utilisé pour écrire l'application parallèle permette d'exprimer cette localité afin de l'exploiter. Nous abordons dans les sections suivantes ces différents points.

1.2 Comprendre l'architecture matérielle

Au delà du comportement de l'application, il est nécessaire, pour prendre en compte la localité des données, d'avoir des informations sur l'architecture matérielle qui sert de support à l'exécution. En particulier, ce sont les informations de structure et, éventuellement, des données quantitatives de type latence ou bande passante qu'il faut être capable

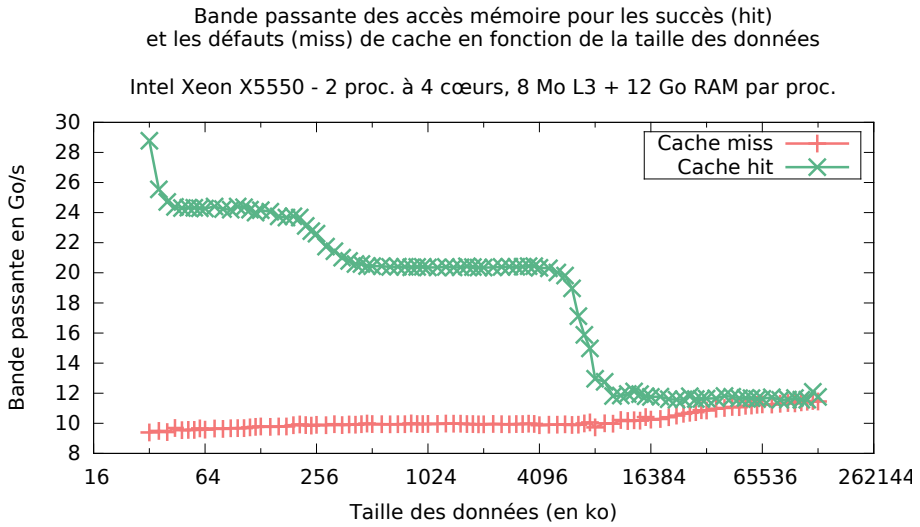


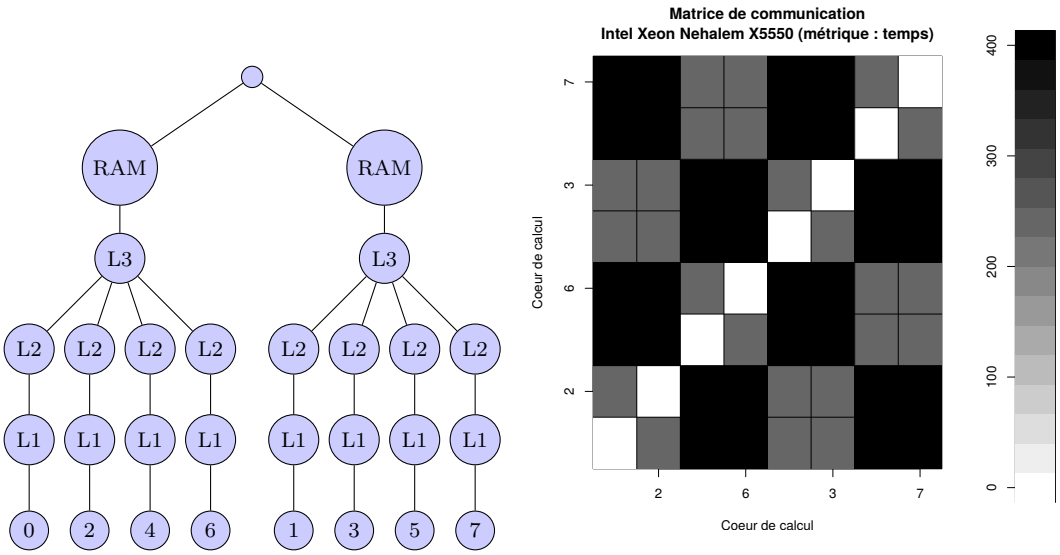
Figure 1.2 – Bande passante en fonction de la localité des données, déterminée suivant la taille des messages et d'un accès préliminaire à la donnée. Mesures réalisées avec l'outil MBENCH.

de récupérer. Or, ce n'est pas une tâche triviale compte tenu de la multiplicité et de la complexité des architectures d'une part et de la faible documentation proposée par les fabricants d'autre part. En effet, la diversité des topologies réseaux s'accompagne désormais d'une architecture complexe à l'intérieur même des nœuds. La démocratisation des technologies dites multi-cœur et *many-core* favorise une organisation très hiérarchique de la mémoire, créant ainsi de nombreux effets d'accès hétérogènes (NUMA). Ainsi, ce n'est plus seulement la topologie du réseau qu'il convient de prendre en considération pour gérer la localité des données mais également la topologie *intra-nœud*.

1.2.1 Constats architecturaux

Il est important de faire ici un préambule sur ce que nous sommes en droit d'attendre des architectures matérielles. Plus concrètement, nous nous posons la question de la corrélation entre les informations théoriques que nous avons à propos d'une architecture (qu'elles soient fournies par le constructeur ou par un outil dédié) et ce que la réalité nous montre. En effet, diverses expériences nous ont révélé, sur certaines architectures, des différences notables entre théorie et pratique.

Afin de mettre en lumière cette problématique, nous avons écrit une application MPI dans laquelle chaque processus envoie de manière bloquante un message de taille 1 Mo à tous les autres. Ces envois sont réalisés séquentiellement de façon à éviter les problèmes



(a) Topologie d'une architecture Intel Xeon X5550

(b) Matrice d'affinité par cœur de calcul (Métrique : temps de communication - unité : ms)

Figure 1.3 – Topologie matérielle d'une architecture Intel Xeon X5550 et modèle d'affinité (métrique de temps) d'une application de type *all to all*

de contention sur les bus de communication et ainsi garantir davantage la précision des mesures. L'outil EZTrace [76] nous permet d'extraire une trace de l'exécution de cette application. En particulier, il nous fournit le temps de communication (en millisecondes) pour chaque message envoyé. Nous présentons ensuite ces temps sous forme de matrice de taille $p \times p$ avec $p = n$ où p est le nombre de processus de l'application et n le nombre de cœurs de l'architecture cible. Compte tenu des spécifications des architectures que nous avons évaluées, nous nous attendons à retrouver dans la matrice de temps de communication les caractéristiques en terme de bande passante des niveaux de mémoire.

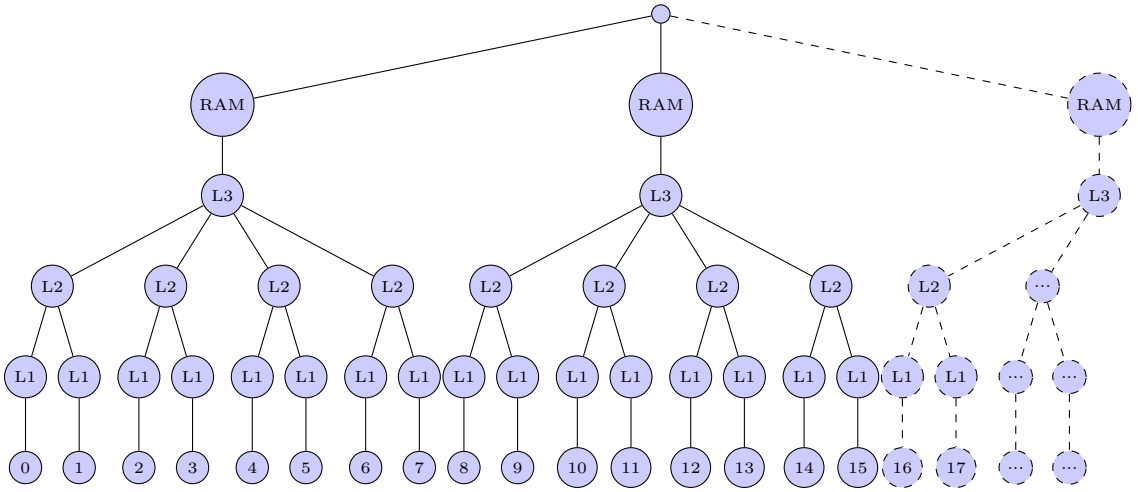
La première matrice présentée en figure 1.3(b) montre l'affinité de huit processus répartis sur les huit cœurs d'une architecture Intel Xeon X5550. La matrice est volontairement représentée par rapport aux unités de calcul et non aux processus. L'architecture concernée possède un seul niveau de hiérarchie mémoire : un cache L3 et 12 Go de mémoire RAM partagés par les quatre cœurs d'un même socket, comme décrit en figure 1.3(a). Sur la matrice, nous pouvons clairement distinguer cette hiérarchie mémoire puisque les temps de communication entre cœurs d'un même socket sont plus faibles que les temps de communication entre sockets (et qui utilisent donc la mémoire RAM, plus lente). Le tout est très homogène et corrobore les informations théoriques en notre possession.

La seconde matrice de la figure 1.4(b) présente quant à elle l'affinité de 64 processus assignés aux 64 cœurs d'une architecture AMD Opteron 6272. Cette architecture propose un cache L2 partagé entre des paires de cœurs. Chaque groupe de quatre paires partage un cache L3 et 16 Go de mémoire RAM. Une représentation de cette topologie est montrée en figure 1.4(a). La métrique déterminant l'affinité est une fois encore basée sur le temps de communication entre processus. Nous pouvons voir sur cette matrice que les mesures de temps obtenues ne reflètent absolument pas la hiérarchie mémoire annoncée. En effet, nous devrions distinguer sur cette matrice des groupes de deux processus (L2 partagé) regroupés par quatre (L3 partagé) autour de la diagonale. Le reste devrait faire apparaître des temps de communication plus longs puisqu'aucun banc mémoire n'est partagé. Or, nous observons à l'inverse une répartition plutôt chaotique des temps de communication. Cet exemple démontre les difficultés que nous pouvons rencontrer pour obtenir une bonne représentation des différentes architectures et pour comprendre leurs caractéristiques peu ou pas documentées.

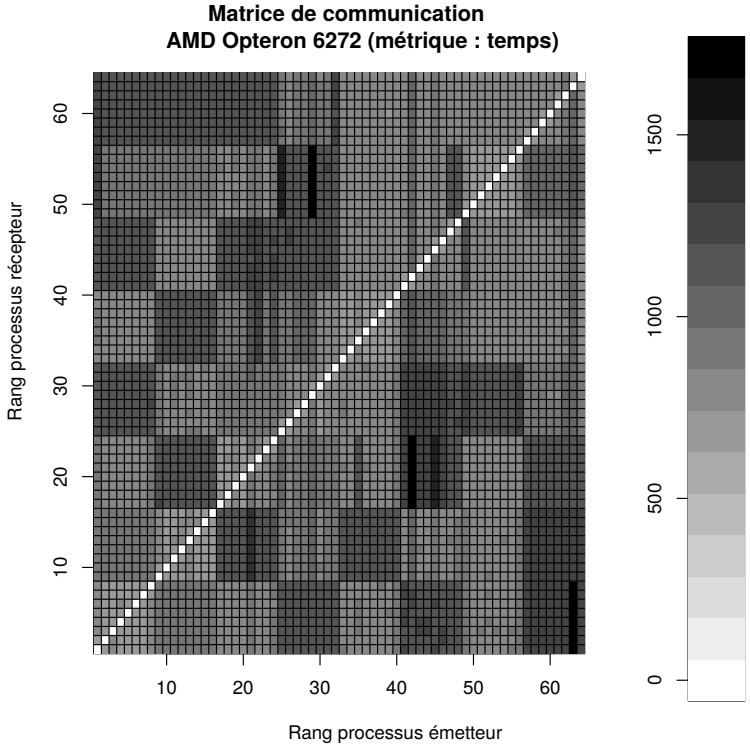
1.2.2 Numérotations physiques et logiques

Les différents cœurs d'une architecture matérielle sont numérotés par le constructeur. Cette numérotation, dite physique, n'est pas toujours cohérente d'une génération à l'autre et peut poser d'importants problèmes lors de l'affectation des entités logicielles. En effet, il est d'une part compliqué pour l'utilisateur de connaître avec précision la numérotation physique définie par le constructeur et, d'autre part, manipuler cette numérotation peut s'avérer relativement délicat et sujet à erreurs.

La numérotation physique des cœurs permet d'identifier les unités de calcul de façon à y affecter des entités logicielles. Cette numérotation est définie par le constructeur et gérée depuis le système élémentaire d'entrée/sortie (Basic Input Output System, BIOS). Ainsi, il peut arriver qu'une simple mise à jour du BIOS fasse varier la numérotation. Le tableau 1.1 énumère quelques exemples de numérotations présentes sur des architectures que nous avons utilisées. Nous avons été largement confrontés à ces contraintes de numérotation. Il peut arriver qu'au sein d'une même plate-forme de calcul plusieurs nœuds d'architecture identique ne fournissent pas la même numérotation physique. Une solution consiste à ne manipuler qu'une numérotation dite logique, basée sur le parcours en profondeur des feuilles de l'arbre de la topologie. Cette alternative se propose de déléguer à une bibliothèque la correspondance entre les indices physiques et logiques afin d'éviter les erreurs de manipulation au sein du code. Il s'agit d'une solution satisfaisante même si son adoption n'est pas encore généralisée. Si une telle solution n'est pas adoptée, il incombe alors à l'utilisateur de jongler avec les indices physiques au risque réel de mal prendre en compte la localité des données.



(a) Partie de la topologie d'une architecture AMD Opteron 6272



(b) Matrice d'affinité (Métrique : temps de communication - unité : ms)

Figure 1.4 – Topologie matérielle d'une architecture AMD Opteron 6272 et modèle d'affinité (métrique de temps) d'une application de type *all to all*.

Processeur	Numérotation physique	Ordre de numérotation
Intel Xeon X7460 96 cœurs	0,4,8,12,16,20,1,5,9,13,17,21,2,6,10,14, 18,22,3,7,11,15,19,23...	Socket, cœur, nœud NUMA
AMD Opteron 6272 64 cœurs BIOS v2.3.0	0,4,8,12,16,20,24,28,1,5,9,13,17,21,25, 29,32,36,40,44,48,52,56,60...	Socket, cœur
AMD Opteron 6272 64 cœurs BIOS v3.2.1	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15, 16,17,18,19,20,21,22,23,24...	Cœur
Intel Xeon X5550 8 cœurs (sans *hyperthread*)	0,1,2,3,4,5,6,7 (parfois 0,2,4,6,1,3,5,7)	Cœur (parfois Socket, cœur)
Intel Xeon X5550 8 cœurs (avec *hyperthread*)	0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15	Cœur, *hyperthread*

Table 1.1 – Exemple de numérotation physique des cœurs sur différentes architectures. La numérotation est celle obtenue en effectuant un parcours en profondeur des feuilles de l'arbre de la topologie *c-à-d.* le $i^{\text{ème}}$ élément de la liste correspond au cœur logique i.

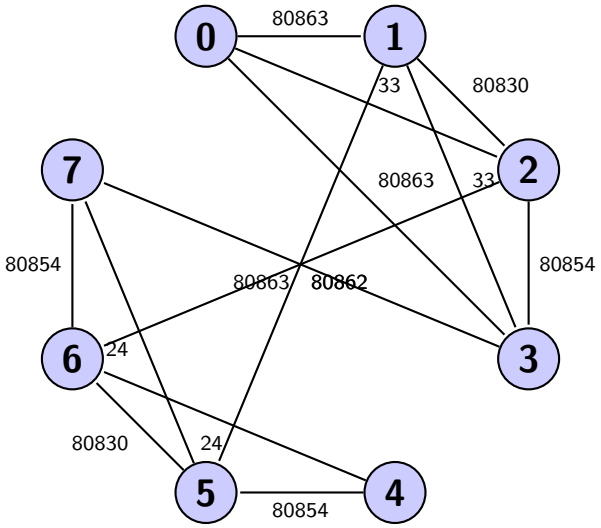
1.3 L'affinité

L'affinité peut se définir comme le degré de dépendance entre des entités logicielles en fonction de certains critères. Ainsi, l'affinité peut s'exprimer par exemple en termes de volume de communication entre entités, d'entrées/sorties ou encore d'accès mémoire. Une représentation possible de l'affinité se fait sous forme de graphe dans lequel les sommets représentent les entités logicielles et les arêtes représentent les dépendances. Ces arêtes peuvent être pondérées en fonction de la métrique utilisée.

La figure 1.5(a) montre un exemple de graphe d'affinité déterminé à partir de huit entités logicielles d'une application et basé sur une métrique quelconque (communication, accès mémoire, etc). Une autre représentation, sous forme de matrice, est présentée en figure 1.5(b). On peut voir par exemple qu'une dépendance existe entre le sommet 0 et le sommet 1 et que l'arête correspondante est pondérée par la valeur 80863. Concrètement, cela signifie que les entités de rang 0 et 1 présentent une affinité de valeur 80863 durant la totalité de l'exécution de l'application. Cette valeur est déterminée suivant la métrique. L'affinité permettra de déterminer si deux entités ont intérêt à être placées plus proches physiquement l'une de l'autre sur l'architecture matérielle. Pour cela, encore faut-il avoir une représentation précise de cette architecture afin de prendre en compte ses spécificités.

1.4 Formalisation du problème

Le problème du placement d'entités logicielles en fonction de la topologie matérielle peut être vu comme un problème de minimisation de certaines métriques. Nous allons ici formaliser ce problème.



(a) Exemple de graphe des affinités entre entités logicielles basé sur une métrique quelconque.

Entité	0	1	2	3	4	5	6	7
0	0	80863	33	0	80863	0	0	0
1	80863	0	80830	33	0	80862	0	0
2	33	80830	0	80854	0	0	80862	0
3	0	33	80854	0	0	0	0	80863
4	80863	0	0	0	0	80854	24	0
5	0	80862	0	0	80854	0	80830	24
6	0	0	80862	0	24	80830	0	80854
7	0	0	0	80863	0	24	80854	0

(b) Matrice présentant la valeur de l'affinité entre des entités logicielles.

Figure 1.5

Posons :
- A le graphe pondéré des entités logicielles tel que $A = (V_A, \omega_A)$ avec :
  - V_A : l'ensemble des sommets représentant les entités logicielles ;
  - $\omega_A(u, v)$: une métrique d'affinité (communication, accès mémoire, etc) déterminée entre deux entités $u, v \in V_A$.
- H le graphe pondéré de l'architecture tel que $H = (V_H, \omega_H)$ avec :
  - V_H : l'ensemble des nœuds de la topologie ;
  - $\omega_H(u, v)$: le poids de l'arête incidente des sommets u et v, $u, v \in V_H$. Certaines solutions limitent $\omega_H(u, v)$ à l'ensemble binaire $0, 1$ de telle façon qu'une arête soit uniquement présente ou absente.
- $D(a, b)$ le débit mesuré entre les unités de calcul a et b.
- $d_H(a, b)$ la distance du plus court chemin entre les unités de calcul a et b

Le placement en fonction de la topologie peut alors se formaliser par la fonction $\sigma : V_A \to V_H$. Cette fonction σ affecte chaque sommet s de l'ensemble V_A des entités logicielles à un sommet cible t de l'ensemble V_H des ressources de l'architecture. Certains travaux considèrent la fonction σ comme injective ou surjective. Parler d'injectivité de la fonction σ revient à dire que, sur chaque unité de calcul, une et une seule entité logicielle peut être affectée. Ceci implique également qu'une unité de calcul peut ne pas être utilisée. Dans ce cas précis, on parlera de *permutation* pour évoquer la fonction σ puisqu'elle consiste à permuter les entités d'une application sur les unités de calcul. La surjectivité de la fonction σ sous-entend que toutes les unités de calcul doivent héberger au moins une entité logicielle. Ces deux caractéristiques sont cependant complémentaires. En effet, plusieurs entités du graphe A peuvent être affectées à la même ressource du graphe H, de même qu'une ressource du graphe H peut ne pas être utilisée.

1.4.1 Fonctions objectifs

À chaque placement σ est associé une fonction objectif qu'il s'agit le plus souvent de minimiser. Ces fonctions objectifs dépendent là encore de la métrique adoptée. Nous en avons défini quatre, qui se basent sur une approche visant la minimisation des coûts de communication.

La première fonction objectif consiste à minimiser le temps total passé dans les communications, *TempsComm*. Elle se base sur le rapport entre volume de données (ici, $\omega_A(u, v)$ est uniquement un volume de communication) et débit entre unités de calcul de la topologie pour estimer ce temps. Plus formellement l'objectif peut s'énoncer :

$$TempsComm(\sigma) = \sum_{(u,v) \in V_A} \frac{\omega_A(u, v)}{D(\sigma(u), \sigma(v))}.$$

Une autre approche propose de minimiser le temps maximum passé à communiquer, que nous appelons *MakespanComm*, en se basant sur des mesures de débit entre deux unités de calcul. Ainsi, soit $T(u, v)$ le temps passé à échanger le volume de données $\omega_A(u, v)$ entre les entités u et v, la fonction objectif peut s'écrire :

$$MakespanComm(\sigma) = \max_{(u,v)\in V_A} T(u, v) \mid T(u, v) = \frac{\omega_A(u, v)}{D(\sigma(u), \sigma(v))}.$$

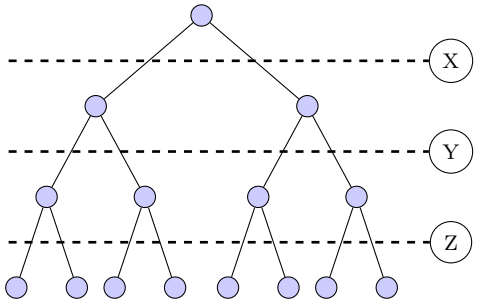
Une troisième méthode a pour objectif de minimiser la somme des communications pondérée par le nombre de liens parcourus dans la topologie, *DistComm*. Cette méthode part de l'hypothèse que la topologie est un arbre équilibré et symétrique. Soit n le nombre de niveaux dans l'arbre de la topologie et E_n l'ensemble des couples (u, v) avec $u, v \in V_A$ tel que la distance du plus court chemin entre l'unité de calcul dédiée à u et l'unité de calcul dédiée à v soit $d_H(\sigma(u), \sigma(v)) = n \times 2$. Alors la fonction objectif peut s'écrire :

$$DistComm(\sigma) = \sum_{i=1}^{n} \left(\sum_{(u,v)\in E_i} \omega_A(u, v) \times d_H(\sigma(u), \sigma(v)) \right)$$

$$\text{où } E_i = \{(u, v)\} \mid d_H(\sigma(u), \sigma(v)) = i \times 2$$

Un exemple illustrant cette fonction objectif *DistComm* est présenté en figure 1.6. L'arbre (figure 1.6(a)) représente une topologie matérielle sur laquelle chaque feuille correspond à une unité de calcul $\sigma(u)$ accueillant l'entité logicielle $u \in V_A$. Sur cet représentation, X est la somme des communications passant par les arêtes adjacentes au niveau 0. Y est la somme des communications passant par les arêtes adjacentes au niveau 1 mais qui ne passent pas par le niveau 0. Enfin, Z est la somme des communications des arêtes adjacentes au niveau 2 et qui ne passent pas par les niveaux supérieurs. Un exemple de modèle de communication est présenté en figure 1.6(b). D'après la fonction objectif, le but est donc de minimiser $Z \times 2 + Y \times 4 + X \times 6$. Les valeurs $2, 4, 6$ correspondent au nombre de liens à parcourir pour deux entités communicantes à travers ce niveau de l'arborescence. En figure 1.6(c), trois exemples de placements, c'est-à-dire de solution à la fonction objectif, sont présentés. Cette notation donne pour chaque $\sigma(j)$, où j est une entité logicielle telle que $j \in V_A$, l'unité de calcul sur laquelle l'entité j sera affectée. Le tableau associé en figure 1.6(d) montre le résultat de ces placements en fonction du modèle de communication et de la topologie. La dernière solution est de loin la plus profitable puisque la majorité des communications sont routées sur le niveau le plus bas de l'arbre de la topologie.

Cette fonction, à la différence des deux précédentes, ne se base sur aucune valeur quantitative expérimentale. Seule la distance dans l'arbre de la topologie est nécessaire. De notre point de vue, il s'agit d'un avantage important, tant les mesures expérimentales dépendent de contraintes fortes (congestion, concurrence, etc).



(a) Calcul de la fonction objectif *DistComm*.

Entité	0	1	2	3	4	5	6	7
0	0	**20**	0	0	**10**	0	**10**	**40**
1	**20**	0	0	0	0	0	0	**10**
2	0	0	0	**20**	0	0	0	0
3	0	0	**20**	0	**20**	0	0	**10**
4	**10**	0	0	**20**	0	**20**	0	0
5	0	0	0	0	**20**	0	0	0
6	**10**	0	0	0	0	0	0	**20**
7	**40**	**10**	0	**10**	0	0	**20**	0

(b) Matrice d'affinité.

$\sigma_1 = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$\sigma_2 = \{0, 2, 4, 6, 1, 3, 5, 7\}$
$\sigma_3 = \{0, 1, 4, 5, 6, 7, 2, 3\}$

(c) Fonctions σ (permutations).

Fonction	σ_1	σ_2	σ_3
X	100	80	20
Y	0	80	80
Z	80	20	80
MinDistComm	**760**	**840**	**600**

(d) $DistComm(\sigma) = Z \times 2 + Y \times 4 + X \times 6$.

Figure 1.6 – Exemple de calcul de la fonction objectif *DistComm*.

Enfin, certains travaux [28, 21] proposent de réduire les communications sur le plus haut niveau de la hiérarchie puis d'appliquer récursivement cet objectif sur les niveaux inférieurs. Cette fonction objectif *CoupeMin* peut s'énoncer comme suit. Soient V_1 et V_2 des sous-graphes de V_A tels que $V_1 \subset V_A$, $V_2 \subset V_A$ et $V_1 \cap V_2 = \emptyset$. Posons $Edges(V_1, V_2)$ la fonction déterminant le nombre d'arêtes incidentes entre les deux sous-graphes V_1 et V_2. Ainsi pour chaque niveau de la topologie on cherche :

$$CoupeMin = \min Edges(V_1, V_2) \mid |V_1| = |V_2| \; et \; V_1 \cup V_2 = V_A$$

1.4.2 Approche algorithmique

Le problème du placement sur une topologie en fonction d'une métrique quelconque peut être réduit à plusieurs problèmes de la théorie des graphes. Deux découlent des fonctions objectif précédemment détaillées. La minimisation des deux premières fonctions (*TempsComm*, *MakespanComm*) implique un problème d'affectation quadratique. En effet, pour ces deux méthodes, il s'agit d'affecter les sommets du graphe V_A sur les sommets du graphe V_H tout en définissant un facteur de coût (débit, distance, etc). Ce problème est NP-Difficile [23]. La fonction *CoupeMin* correspond à un problème de bi-partitionnement à coupe minimum. Ce problème est NP-Complet [23]. Enfin, nous allons nous attarder sur la dernière fonction *DistComm*, afin de prouver la NP-Difficulté de sa minimisation.

1.4.2.1 Preuve que la minimisation de *DistComm* est NP-Difficile

Afin de prouver que le problème de minimiser la fonction objectif *DistComm* est NP-Difficile, nous allons nous baser sur une réduction vers le problème de bi-partitionnement à coupe minimum.

Soit $G = (V, E)$ un graphe tel que son nombre de sommets est $|V| = n$ et E est l'ensemble des arêtes pondérées. Pour les besoins de la démonstration, on suppose n pair. Dans le cas où n est impair, il suffit d'ajouter au graphe un sommet isolé. Dans le cas du placement d'entités logicielles, il s'agit d'un graphe d'affinité.
Posons la fonction $\tau : E \to \mathbb{N}$ définissant pour chaque arête la valeur de son poids.

Soit T un arbre de structure 2:$n/2$. C'est-à-dire un arbre dont l'arité du niveau 0 est $k_0 = 2$ et l'arité des sommets de niveau 1 est $k_1 = n/2$. Une illustration de cet arbre est exposée en figure 1.7. La construction de cet arbre à partir du graphe G à n sommets se fait en temps polynomial.

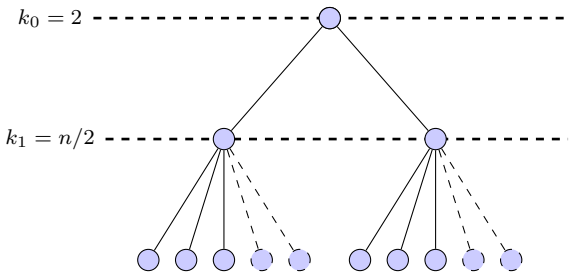


Figure 1.7 – Arbre T à n feuilles. L'arité k_0 du niveau 0 est de 2, l'arité k_1 de niveau 1 est de $n/2$.

Soit σ une affectation des sommets de G sur les feuilles de T. On remarque que σ définit un bi-partitionnement (V_1, V_2) de V où V_1 est affecté aux $n/2$ feuilles de gauche et V_2 est affecté aux $n/2$ feuilles de droite. Pour chaque arête $(a, b) \in E$, deux cas sont possibles :

1. a et b sont dans le même sous-arbre de T. On note C_1 l'ensemble de telles arêtes.
2. a et b sont dans deux sous-arbres différents de T. On note C_2 l'ensemble de telles arêtes.

Posons :
— $c = \sum_{e \in C_2} \tau(e)$ la somme du poids des arêtes du cas 2;
— $M = \sum_{e \in E} \tau(e)$ la somme du poids de toutes les arêtes du graphe G.

Ainsi, nous avons $DistComm(G, \sigma) = 4c + 2(M - c) = 2c + 2M$. Cette évaluation se fait en temps polynomial.

Soit $\sigma^\star$ tel que $DistComm(G, \sigma^\star)$ soit minimal. On appelle $c^\star$ la somme du poids des arêtes du cas 2 pour $\sigma^\star$. Alors $c^\star$ est aussi la coupe optimale pour le bi-partitionnement de G.

En effet, supposons qu'il existe $\hat{\sigma}$ un bi-partitionnement (V_1, V_2) de V tel que la coupe $\hat{c}$ soit inférieure à $c^\star$. Alors, en affectant les sommets de V_1 au sous-arbre de droite de T et ceux de V_2 à son sous-arbre de gauche, on obtient que la somme des poids de toutes les arêtes entre V_1 et V_2 entre dans le cas 2 et sa valeur est $\hat{c}$. Ceci implique que $DistComm(G, \hat{\sigma}) = 2\hat{c} + 2m < 2c^\star + 2m = DistComm(G, \sigma^\star)$, ce qui est impossible puisqu'en contradiction avec la minimalité de $DistComm(G, \sigma^\star)$.

Ainsi, pour tout graphe G, si on possède un algorithme pour minimiser la fonction objectif *DistComm*, alors on a un algorithme pour minimiser le bi-partitionnement à coupe minimum de G. Ce dernier problème étant NP-Difficile, minimiser *DistComm* l'est également.

Dans la majorité des cas, calculer un placement efficace sur une topologie en prenant en compte une fonction de coût se réduira à un problème au moins aussi difficile qu'un problème NP-Difficile.

1.5 Gestion de la localité et du placement dans les supports d'exécution parallèle

L'expression de la localité des données et la possibilité de choisir le placement des entités logicielles d'un paradigme de programmation sont des éléments déterminants pour prendre en compte les dépendances entre entités et l'architecture matérielle dans l'exécution d'une application. Cependant, force est de constater que bon nombre de paradigmes très utilisés sont inadaptés et la quantité importante d'applications qu'ils ont permis de développer ne favorise naturellement pas l'émergence de concepts plus appropriés (comme Charm++, ou les versions plus récentes de MPI proposant des options de placement gloutons).

Toutefois, la notion de localité a été adoptée dans plusieurs standards et implémentations comme OpenMP ou MPI dans le but d'améliorer le passage à l'échelle des applications parallèles. Ces ajouts apparaissent cependant assez minimalistes dans le contexte récent du multi-cœur. Le fossé est parfois grand entre la représentation proposée par ces paradigmes et l'architecture réelle. Concernant MPI, cette différence se justifie par le fait que le standard a été conçu pour ne pas se soucier des caractéristiques de la topologie matérielle. Néanmoins la question se pose d'introduire un niveau de détail supérieur dans la

spécification au vu des évolutions des plate-formes de ces dernières années. Ce sujet est régulièrement abordé lors des réunions de définition du standard, bien qu'aucun consensus n'existe à ce jour.

Pour pallier ce manque de précision dans l'expression de la localité des données, la possibilité pour l'utilisateur de préciser le placement de ses entités logicielles est nécessaire. Plusieurs implémentations de paradigmes proposent cette fonctionnalité comme OpenMP, MPI ou encore Charm++. L'utilisateur peut alors définir un placement des objets en fonction de critères d'affinité, d'une représentation plus fidèle de l'architecture et des algorithmes que nous venons de lister. Nous allons nous concentrer ici sur les techniques permettant de prendre en compte la politique de placement calculée. Ce placement est fourni au support d'exécution qui se chargera d'attacher les différentes entités aux unités de calcul physique. Notons que plusieurs approches sont possible. D'un côté, nous allons évoquer le placement statique des entités afin d'exprimer le fait que le support d'exécution assigne les objets à des unités de calcul physique dédiées au début de l'exécution. D'un autre côté, nous allons parler de réordonnancement dynamique des entités logicielles pour signifier que leur rang au sein du paradigme de programmation évolue en cours d'exécution.

1.5.1 Placement statique

La première solution de placement que nous pouvons évoquer concerne le placement statique, c'est-à-dire le fait d'affecter chaque entité logicielle de l'application à une unité de calcul dédiée pour toute la durée de l'exécution. Par exemple, sur les systèmes d'exploitation libres de type GNU/Linux, les commandes comme `numactl` ou `taskset` remplissent ce rôle. Si aucune contrainte de placement n'est spécifiée, l'ordonnanceur du système d'exploitation gère lui-même cette tâche et peut à tout moment déplacer des entités logicielles d'une unité de calcul à une autre, créant de fait des défauts de cache impactant les performances. Qui plus est, la reproductibilité des exécutions se trouve fortement altérée si aucune garantie sur la stabilité de la localité n'est fournie. Les résultats en tableau 1.2 montrent les différences de temps d'exécution de l'application de mécanique des fluides ZeusMP/2 dans le cas où les processus sont attachés et dans le cas où rien n'est spécifié. Nous pouvons constater que dans tous les cas, le coefficient de variation (ratio de l'écart type par la médiane) est plus faible dans le cas avec placement que dans le cas sans placement. Cette donnée statistique indique que le placement des entités réduit le bruit système et favorise la stabilité des exécutions.

Les principaux paradigmes de programmation parallèle s'accompagnent d'un gestionnaire capable de gérer le placement fourni par l'utilisateur. Par exemple, le gestionnaire de processus ORTE de l'implémentation Open MPI du standard MPI prend en entrée un fichier de type *rankfile* précisant les informations de placement des processus. Un exemple de fichier est présenté en figure 1.8. Cette solution a l'avantage de ne pas nécessiter de

Table 1.2 – Statistiques pour 10 exécutions de l'application ZeusMP/2 sur 64 processus (cas étudiant la magnétohydrodynamique) comparant le cas avec placement et le cas sans placement de processus. L'unité de mesure du temps est la seconde.

Nombre d'Iterations	Sans placement			Avec placement		
	Moyenne	Écart type	Coef. de Variation	Moyenne	Écart type	Coef. de Variation
2000	2.8627	0.127	0.044	2.6807	0.062	0.023
3000	4.1691	0.112	0.027	4.0023	0.097	0.024
4000	5.4724	0.069	0.013	5.2588	0.052	0.010
5000	7.3187	0.289	0.039	6.8539	0.121	0.018
10000	13.9583	0.487	0.035	13.3502	0.194	0.015
15000	20.4699	0.240	0.012	19.8752	0.154	0.008
20000	27.0855	0.374	0.014	26.3821	0.133	0.005
25000	33.7065	0.597	0.018	32.8058	0.247	0.008
30000	40.6487	0.744	0.018	39.295	0.259	0.007
35000	46.7287	0.780	0.017	45.7408	0.299	0.007
40000	53.3307	0.687	0.013	52.2164	0.227	0.004
45000	59.9491	0.776	0.013	58.8243	0.632	0.011
50000	66.6387	1.095	0.016	65.3615	0.463	0.007

modification du code de l'application. Cependant, elle permet uniquement d'établir le placement de chaque entité au lancement de l'application. Gérer les mouvements d'entités en cours d'exécution soulève d'autres difficultés.

1.5.2 Réordonnancement dynamique

Le réordonnancement dynamique est un autre moyen d'appliquer une politique de placement. Cette technique se base sur les indices de chaque entité logicielle. En effet, dans les différents paradigmes de programmation parallèle, chaque entité possède un numéro unique permettant de l'identifier afin de réaliser les échanges de données entre elles ou encore de les synchroniser. Le réordonnancement dynamique consiste à réattribuer ces indices en cours d'exécution. Cette méthode ne nécessite pas d'outil externe mais est directement implémentée dans certains paradigmes comme MPI ou Charm++. Il s'agit d'un avantage important, favorisant de fait la portabilité, la transparence et la dynamicité. Cependant, elle a besoin d'être prise en compte dans le code de l'application, contrairement au placement statique. Concrètement, comment utiliser cette technique dans les paradigmes sur lesquels nous nous sommes penchés (MPI et Charm++) ? Concernant MPI, la solution se trouve du côté de la fonction `MPI_Dist_graph_create`, capable de créer un nouveau communicateur dont les rangs des processus MPI ont été réordonnés. Charm++, en revanche, propose un mécanisme de placement dynamique interne un peu différent. En effet, chaque entité logicielle peut se voir attribuer une unité de calcul en cours d'exécution et est migrée en fonction. La migration suit le principe du *pack-unpack* : lorsqu'une entité doit être migrée, sont état est stocké dans une portion de mémoire contiguë (message, fichier,

```
rank 0=fourmi017 slot=p0:0
rank 1=fourmi017 slot=p0:1
rank 2=fourmi017 slot=p1:0
rank 3=fourmi017 slot=p1:1
rank 4=fourmi022 slot=p0:0
rank 5=fourmi022 slot=p0:1
rank 6=fourmi022 slot=p1:0
rank 7=fourmi022 slot=p1:1
rank 8=fourmi017 slot=p0:2
rank 9=fourmi017 slot=p0:3
rank 10=fourmi017 slot=p1:2
rank 11=fourmi017 slot=p1:3
rank 12=fourmi022 slot=p0:2
rank 13=fourmi022 slot=p0:3
rank 14=fourmi022 slot=p1:2
rank 15=fourmi022 slot=p1:3
```

Figure 1.8 – Fichier de type *rankfile* pour Open MPI décrivant le placement des processus à l'exécution. Le format est de type `rank <rang MPI>=<hôte> slot=p<socket>:<cœur physique>`. À l'exécution : `mpiexec -np <nombre de processus> -rankfile <fichier> <exécutable>`

etc) puis est envoyé vers la nouvelle unité de calcul. Sur cette unité, une nouvelle entité logicielle est créée avec l'état réceptionné.

Notons enfin que cette solution n'est réellement efficace que lorsque les entités sont attachés au lancement de l'application (sans politique de placement particulière). Dans le cas contraire, les mouvements d'entités provoquées par l'ordonnanceur du système d'exploitation créent là encore un bruit impliquant des pertes de performance.

Sommaire

Chapitre 2

Placement statique

2.1 Problématiques et motivations

Nous l'avons vu un peu plus tôt : le placement d'entités logicielles a une influence sur les coûts de communication. Les niveaux de hiérarchie (cache, mémoire, réseau, etc) ont des caractéristiques très différentes, en particulier en termes de latence et de bande passante. Ainsi, deux entités qui communiquent beaucoup verront leur coût en temps d'échange de données se réduire si elles partagent le même cache L3. À l'inverse, si ces entités sont placées sur des nœuds différents et doivent communiquer via le réseau, le temps de communication sera fortement pénalisé. Ce chapitre va s'articuler autour de notre technique de placement statique, dont le but est d'affecter chaque entité logicielle à une unité de calcul dédiée. Plus concrètement, nos travaux sur le placement statique se résument à placer les entités à l'intérieur des nœuds de façon à tenir compte des effets NUMA. D'une manière générale, nous considérons le réseau plat même si notre solution peut s'adapter à des réseaux arborescents sous certaines conditions (arbre équilibré et symétrique).

Calculer un bon placement d'entités nécessite de connaître d'une part les détails de l'architecture cible et, d'autre part, d'avoir des informations quant à l'affinité des entités de l'application. L'affinité, définie en détail dans le chapitre précédent, peut être vue comme la tendance qu'ont deux éléments à être inter-dépendants. Afin de déterminer cette affinité,

nos travaux se sont focalisés sur les volumes de communications entre éléments. Cependant, d'autres métriques peuvent s'avérer pertinente en fonction des applications ciblées (entrées/sorties, accès mémoire, métrique multi-critère, etc).

Notons que nous avons choisi comme support de notre méthode de placement le paradigme d'Interface à Passage de Message (MPI) qui est une solution répandue dans le domaine du calcul intensif. Cependant, les travaux que nous présenterons dans les sections suivantes ne sont pas dépendants de ce paradigme. Notre algorithme de placement peut tout à fait s'adapter à d'autres modèles de programmation basés sur des éléments communicants (que les communications soient implicites ou explicites) comme Charm++ ou les langages de type PGAS, par exemple. Néanmoins, dans la suite de ce chapitre, nous parlerons de placement de processus afin d'évoquer le fait que nous manipulons des processus MPI.

2.2 État de l'art

Le problème du placement de processus sur des unités de calcul en tenant compte du modèle d'affinité de l'application et de l'architecture cible est un problème ouvert qui a déjà été étudié. Ce problème de placement est couramment réduit à un problème de plongement de graphes [70]. Plus précisément, le problème a été introduit par Hatazaki [25] et un algorithme basé sur l'heuristique de Kernighan-Lin [3] a été proposé et validé par une série d'expériences. Cependant, ces travaux sont limités à un matériel très spécifique et ne s'appliquent pas à des architectures plus génériques. De plus, les auteurs ont optimisé leurs algorithmes pour les graphes d'affinité suivant une topologie cartésienne (grille à N dimensions) mais ont laissé de côté les graphes quelconques. Ainsi, seules des applications très spécifiques comme les stencils sont visées. Les expériences présentées montrent des gains de performance substantiels mais sont réduites à des applications n'effectuant que des communications et aucun calcul.

D'autres travaux prennent en compte les graphes d'affinité génériques mais ne considèrent que la topologie réseau pour l'aspect matériel du placement. Les différentes machines Blue Gene ont particulièrement été visées [72], [79], [5]. Les réseaux InfiniBand ont également fait l'objet d'études, et en particulier d'une évaluation des performances afin de produire une représentation de l'architecture basée sur ce réseau d'interconnexion [67] [33]. Une méthode de *Neighbor Joining* a également été proposée [74] afin de détecter la topologie physique des réseaux InfiniBand. LibTopoMap [28] est capable de traiter des topologies réseaux génériques et se sert de ParMETIS [47] pour résoudre le problème de placement. Ces approches ne prennent cependant pas en compte l'architecture interne des nœuds multi-cœurs et leurs effets NUMA.

Les problèmes d'implémentation des mécanismes de topologie de MPI ont été abordés [36]. Les topologies cartésiennes et les graphes génériques sont traités dans ces travaux, et l'algorithme proposé se base lui aussi sur l'heuristique de Kernighan-Lin. Le critère d'optimisation considéré peut être soit le coût de communication, soit la répartition de charge. Une fois encore, ce travail se concentre sur du matériel précis (architecture NEC SX). L'approche présentée est peu générique et elle ne s'applique pas aux grappes de nœuds multi-cœurs.

MPIPP [13] est une suite d'outils dont le but est d'optimiser l'exécution des applications MPI sur l'architecture sous-jacente. MPIPP se base sur un outil externe pour récupérer de façon statique les détails de l'architecture là où nous effectuons cette tâche dynamiquement durant l'exécution. De plus, MPIPP permet uniquement de répartir les processus MPI sur les différents nœuds mais n'effectue aucun placement précis sur les unités de calcul à l'intérieur de ces nœuds. Les machines multi-cœurs ne sont ainsi pas totalement exploitées étant donné que la hiérarchie mémoire n'est pas prise en compte. Les mêmes inconvénients apparaissent dans d'autres recherches [9]. Ces travaux réduisent bien les coûts de communication inter-nœuds pour les applications MPI en appliquant une opération de renumérotation des rangs. La renumérotation telle qu'elle est présentée [9] ne réordonne que le fichier contenant les noms des nœuds (le *hostfile*) et modifie donc la façon dont les processus seront répartis sur les nœuds. Une fois encore, les processus MPI ne sont pas attachés à des unités de calcul dédiées et l'application n'appelle pas réellement les fonctions de réordonnancement de MPI.

Des travaux récents ont proposé des solutions pour attacher les processus MPI à des unités de calcul dédiées afin de réduire les coûts de communication entre processus [69] [56]. Ces deux articles ont été publiés à la même période et évoquent des approches plutôt similaires : ils n'utilisent pas les fonctions du standard MPI pour réordonner les rangs des processus, ils se basent sur le partitionneur de graphe Scotch [21] et ils ne sont pas capables de récupérer les informations sur l'architecture de manière dynamique à l'exécution. La première étude [69] utilise une approche purement quantitative. La seconde en revanche [56] s'est orientée vers une méthode qualitative puisqu'elle considère les informations de structure hiérarchique de la mémoire d'un nœud et non des valeurs mesurées.

Dans certains cas, une connaissance précise de l'application peut s'avérer très utile. En particulier, une étude très poussée d'une application de calcul de valeur propre [1] a permis de dégager un modèle d'affinité sans exécution préliminaire. Le placement basé sur ce modèle a entraîné des gains très intéressants.

Au delà de l'heuristique de Kernighan-Lin mentionnée précédemment, il existe des algorithmes capables de résoudre le problème de plongement de graphes grâce au partitionnement. Chaco [26] et METIS [47] (ou ParMETIS dans sa version parallèle [71]) sont des exemples de tels partitionneurs de graphes. Le premier se base sur un algorithme

de partitionnement spectral [27] tandis que le second implémente un k-partitionnement multi-niveaux [48, 49]. Zoltan [18], successeur de Chaco, offre plusieurs algorithmes de partitionnement et en particulier une solution de partitionnement géométrique [17]. Scotch [21] est un cadriciel de partitionnement de graphe capable de gérer les structures arborescentes (format *tleaf*) en entrée pour calculer un placement. Scotch se base sur une approche *divisier pour résoudre* en appliquant un double bi-partitionnement récursif [63]. Une différence importante entre les techniques utilisant le plongement/partitionnement de graphe et notre travail est que nous avons besoin uniquement de la structure de l'architecture cible là où les travaux connexes nécessitent des informations quantitatives à propos du matériel. Acquérir ces informations est une étape complexe tant les facteurs qui rentrent en jeu sont importants (effets NUMA, congestion, etc).

Les principales implémentations MPI libres comme MPICH [59] et Open MPI [22] fournissent des options permettant d'attacher les processus à des unités de calcul précises dès le lancement de l'application, et ce grâce à leur gestionnaire de processus (respectivement, Hydra et ORTE). L'utilisateur peut choisir d'utiliser des placements prédéfinis plus ou moins efficaces [32]. Cependant, ces stratégies de placement sont génériques et ne tiennent aucun compte du modèle de communication de l'application. Il existe d'autres supports d'exécutions proposant ce type d'options, comme les implémentations MPI propres aux constructeurs. Citons en exemple Cray ([60], [37]), HP [16] ou IBM [19]. Notons qu'il est également possible d'attacher d'autre types d'entités logicielles. Ce genre de cas peut intervenir par exemple dans le cadre d'applications hybrides, c'est-à-dire lorsque plusieurs paradigmes sont utilisés (passage de message associés à plusieurs processus légers, etc). L'association MPI et OpenMP fait partie des cas traités [35].

Le standard MPI offre plusieurs fonctions permettant de manipuler les graphes d'affinité. Des topologies cartésiennes et des graphes quelconques de processus peuvent être créés au niveau des applications. Plusieurs de ces fonctions possèdent un paramètre de réordonnancement. Cependant, l'implémentation actuelle de ces fonctions est très triviale et le réordonnancement n'est souvent effectif que dans les implémentations des constructeurs [36]. La version 2.2 du standard MPI a néanmoins apporté quelques améliorations entre autres à la manipulation des graphes d'affinité notamment avec la fonction `MPI_Dist_graph_create` [30]. Une implémentation d'une partie des fonctions standards dans le but d'optimiser les entrée/sorties a également été publiée [78] sous le nom de SetMatch.

Les communications collectives sont une fonctionnalité importante du standard MPI et plusieurs travaux ont pour but d'en améliorer les performances en tenant compte des spécificités du support matériel. Une méthode [86] utilise un modèle hiérarchique à deux niveaux afin d'utiliser plus efficacement les nœuds multi-cœurs. D'autres [82], [51] proposent des stratégies de placement de processus dans les collectives pour trouver l'algorithme le plus performant pour l'opération collective considérée. Nos travaux retiennent toutes les communications et ne se restreignent pas aux opérations collectives.

Les paradigmes de programmation autres que MPI peuvent être utilisés pour répondre au problème du placement de processus. Citons par exemple Charm++ [42], que nous présenterons plus en détail en Chapitre 3, qui permet d'effectuer de la répartition de charge dynamique tout en tenant compte de l'affinité et de la topologie. Les langages de type PGAS [77] proposent un modèle simple à deux niveaux (local et distant) pour l'affinité mémoire qui peut être utilisé pour le placement de processus en fonction de l'affinité et de la topologie matérielle.

2.2.1 Tableau récapitulatif des méthodes de placement statique

Le tableau 2.1 récapitule les différentes solutions de placement statique évoquées dans cet état de l'art. Pour ce faire, nous avons déterminé plusieurs critères qui nous ont semblés pertinents :

- Indépendant du matériel : ce critère défini si la solution propose une approche générique ou si elle est adaptée à une architecture précise ;
- Indépendant du paradigme : une méthode est indépendante d'un paradigme précis (MPI, OpenMP, etc) si elle peut une fois encore s'adapter à tout type de modèle ;
- Effet NUMA : les effets NUMA des architectures multi-cœurs sont-ils pris en compte ?
- Topologie Réseau : la structure du réseau est-elle considérée pour calculer un placement efficace ?
- Approche qualitative : une approche qualitative ne se base que sur les informations de structure de la topologie. Aucune autre donnée (bande passante, latence, etc) n'est nécessaire ;
- Récupération dynamique de la topologie : ce critère est rempli si la solution est capable de récupérer les informations de topologie automatiquement ;
- Restrictions du placement : est-il possible de forcer certaines ressources (nœuds, cœurs, etc) à ne pas être utilisées ?

Notons que deux solutions, Chaco et ParMETIS, se trouvent séparées en fin de tableau. En effet, il s'agit de deux partitionneurs de graphes qui, à l'inverse de Scotch et son outil *gmap* par exemple, ne fournissent pas d'algorithme de placement tel quel. Il est donc nécessaire de considérer ces deux entrées du tableau sous l'angle de leur utilisation dans une méthode de placement basée sur des bi-partitionnements successifs que nous avons implémentés pour les besoins de nos expériences. Enfin, certains travaux cités précédemment ne sont pas à proprement parler des méthodes de placement et ne figurent donc pas dans ce tableau.

Méthodes	Indépendant du matériel	Indépendant du paradigme	Effets NUMA	Topologie réseau	Approche qualitative	Récupération dynamique de la topologie	Restrictions du placement
Hatazaki [25]			√				
Blue Gene [72, 79, 5]				tore 3D	√		√
Infiniband [67, 33, 74]			√	√		√	
LibTopoMap [28]	√			√			
Träff [36]		√		√			
MPIPP [13]	√	√		√			
Alrutz [9]	√	√					
Mercier [56], Rodrigues [69]	√	√	√				
Aktulga [1]	√			√			
Scotch [21]	√	√	√				
Dümmler [35]	√		√		√		
SetMatch [78]	√	√					
Chaco [26]	√	√	√				
ParMETIS [71]	√	√	√			√	

Table 2.1 – Comparaison des solutions de placement citée en fonction de différents critères. Chaco, ParMETIS sont présentés selon leur utilisation dans une implémentation de méthode de placement

2.3 Solution

Afin de répondre aux maximum aux critères énoncés dans le tableau 2.1, l'algorithme TreeMatch a vu le jour. Déterminer un bon placement de processus sur une architecture matérielle en fonction de l'affinité nécessite de connaître un certain nombre d'informations. Notre approche se base sur des informations qualitatives, c'est-à-dire que nous nous focalisons sur la structure de l'architecture et non sur des mesures théoriques ou expérimentales qui déterminerait la qualité du lien (par exemple des mesures de latence ou de bande passante). Ce choix se justifie dans la mesure où d'une part, la pratique nous montre que les mesures théoriques fournies par les constructeurs diffèrent souvent de la réalité, et d'autre part, des mesures expérimentales dépendent de bien trop de facteurs pour leur accorder une fiabilité suffisante. Par exemple, de la congestion sur un bus au moment de la prise de mesure peut aisément fausser les résultats. De plus, des mesures expérimentales nécessitent l'exécution préliminaire de programmes d'évaluation de performances sur chaque plate-forme hôte. Notre algorithme TreeMatch a donc besoin en entrée d'une représentation fidèle de l'architecture et également du modèle d'affinité de l'application. Nous allons détailler dans cette section ces deux éléments d'informations ainsi que les moyens utilisés pour les récupérer. Enfin, nous évoquerons le calcul du placement et les techniques mises en œuvre pour l'appliquer.

2.3.1 Représentation de l'architecture

Récupérer une représentation fidèle de l'architecture d'une machine n'est pas une tâche triviale. La diversité du matériel, les spécifications propres à chacun, la faible documentation fournie par les constructeurs ou encore la version du BIOS ou du système d'exploitation sont autant de facteurs qui rendent difficile la récupération des informations matérielles. Depuis quelques années, un outil peut cependant se charger de cette étape. Il s'agit d'hwloc [10] (ou Hardware Locality), développé essentiellement au sein de notre équipe mais aussi conjointement avec l'équipe en charge d'OpenMPI à UTK (University of Tennessee, Knoxville). hwloc permet d'extraire la topologie d'une machine (nœuds NUMA, sockets, hiérarchie mémoire, cœurs) de façon portable, quel que soit le système d'exploitation ou l'architecture. Un nombre important d'informations est collecté, qu'il s'agisse de la taille des caches ou de la localité des périphériques d'entrée/sortie (disque, carte réseau, GPU, etc) par exemple. La figure 2.1 montre la sortie graphique de hwloc exécuté sur un nœud de la plate-forme PlaFRIM, tandis que la sortie au format XML d'une sous-partie de ce nœud (deux cœurs) est présentée en figure 2.2. Un autre intérêt d'hwloc est qu'il fournit une couche d'abstraction au dessus du matériel, de telle façon que l'utilisateur (ou le développeur) n'a pas à se soucier de certaines spécificités. En particulier, la numérotation physique des cœurs décidée par le constructeur varie fortement d'une machine à l'autre. Il devient alors très difficile d'appliquer un quelconque placement tant cette numérotation peut s'avérer compliquée (voir chapitre 1).

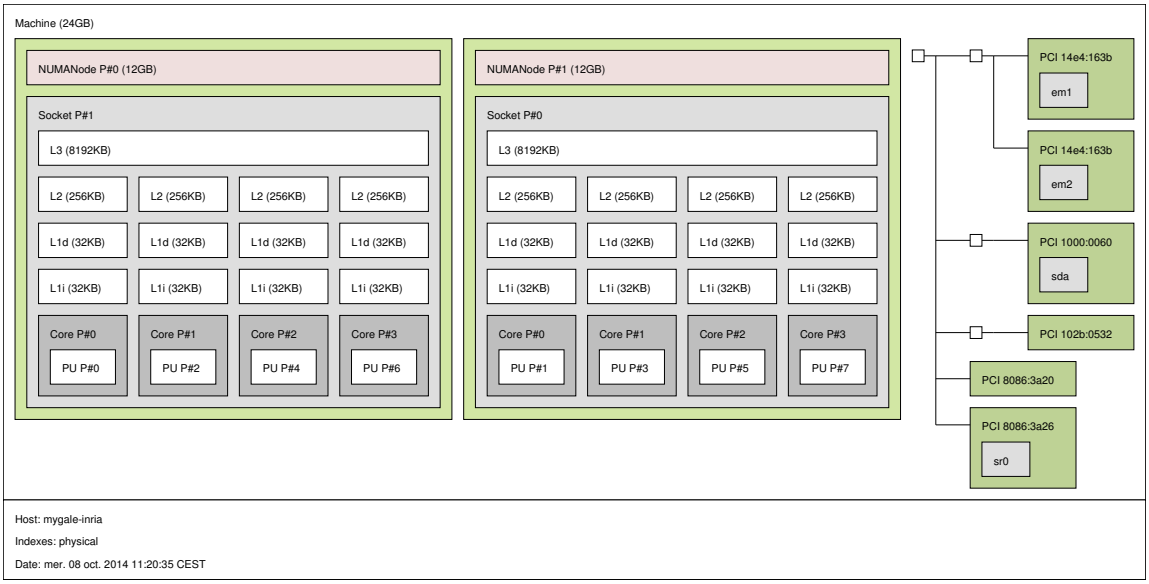


Figure 2.1 – Exemple de topologie fournie par Hwloc d'un nœud PlaFRIM pourvu d'un processeur à 8 cœurs Intel Xeon 5550.

Pour déterminer un bon placement de processus, l'algorithme TreeMatch se limite aux architectures arborescentes. La majorité des architectures modernes présentent ce type de topologie. De récents modèles, comme l'Intel Xeon Phi, se sont davantage orientés vers des structures en anneau. Cependant, cette évolution reste à ce jour très marginale. Dans la représentation utilisée par TreeMatch, chaque niveau de l'arbre correspond à un niveau dans la hiérarchie matérielle (nœuds NUMA, sockets, caches, cœurs). Les feuilles de l'arbre représentent les unités de calcul. Il est nécessaire que l'arbre soit équilibré et symétrique ce qui est finalement assez fidèle à la réalité.

L'intérêt de cette représentation est de pouvoir conserver les informations de structure. De précédents travaux [56] représentaient cette information sous forme d'une matrice de vitesse, n'exploitant pas la notion de parenté des différents niveaux de hiérarchie.

Prendre en compte la topologie du réseau est également une tâche très complexe. Là encore, la diversité du matériel et les topologies très variées que l'on peut rencontrer (tore 3d, *fat-tree*, grille, etc) rendent ardue la détection précise. netloc [24] est l'équivalent au niveau du réseau d'hwloc. Bien qu'encore en cours de développement, il permet déjà de fournir quelques représentations abstraites du réseau ciblé. Ne pas considérer les caractéristiques du réseau est une des limitations de TreeMatch. En effet, comme nous allons le voir plus en détail, notre algorithme est conçu pour traiter les structures arborescentes. Si le réseau s'organise sous forme d'arbre (avec les contraintes citées précédemment d'équilibre et de symétrie), TreeMatch pourrait tout à fait le prendre en compte pour calculer

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topology SYSTEM "hwloc.dtd">
<topology>
  <object type="Machine" os_index="0" >
    <info name="DMIProductName" value="PowerEdge R410"/>
    <info name="DMIBoardVendor" value="Dell Inc."/>
    <info name="DMIBIOSVersion" value="1.0.5"/>
    <info name="OSName" value="Linux"/>
    <info name="OSRelease" value="2.6.32-358.23.2.el6.x86_64"/>
     <info name="HostName" value="mygale-inria"/>
    <info name="Architecture" value="x86_64"/>
    <object type="NUMANode" os_index="0" local_memory="12884901888">
      <object type="Socket" os_index="1" >
        <info name="CPUModel" value="Intel(R) Xeon(R) CPU X5560  @ 2.80GHz"/>
        <object type="Cache" cache_size="8388608" depth="3">
          <object type="Cache" cache_size="262144" depth="2">
            <object type="Cache" cache_size="32768" depth="1" cache_type="1">
              <object type="Cache" cache_size="32768" depth="1" cache_type="2">
                <object type="Core" os_index="0">
                  <object type="PU" os_index="0"/>
                </object>
              </object>
            </object>
          </object>
          <object type="Cache" cache_size="262144" depth="2">
            <object type="Cache" cache_size="32768" depth="1" cache_type="1">
              <object type="Cache" cache_size="32768" depth="1" cache_type="2">
                <object type="Core" os_index="1">
                  <object type="PU" os_index="2"/>
                </object>
              </object>
            </object>
          </object>
        </object>
      </object>
    </object>
  </object>
</topology>
```

Figure 2.2 – Exemple allégé d'une sortie en XML de HWLOC pour une moitié de socket d'un nœud de la plate-forme PlaFRIM.

un placement de processus efficace. Dans le cas contraire, soit d'autres algorithmes comme celui proposé par LibTopoMap peuvent s'en charger, soit une évolution de l'algorithme TreeMatch pourrait être envisagée. En particulier, des techniques de partitionnement de graphes pourraient servir de base à cette amélioration.

2.3.2 Modèle d'affinité

Le modèle d'affinité d'une application représente la tendance qu'ont les processus à interagir (au sens large). Cette affinité peut être déterminée en fonction de plusieurs métriques : entrées/sorties, accès mémoire, etc. Nous avons choisi de nous baser sur l'échange de données entre processus. En effet, la multiplication des unités de calcul au sein d'une même machine et la diminution de la mémoire par cœur encourage les applications parallèles à communiquer de plus en plus pendant l'exécution. Cette métrique nous a semblé pertinente pour ces raisons. Il est cependant à noter que tout comme l'algorithme TreeMatch ne dépend pas de MPI, il ne dépend pas non plus de la seule métrique d'échange de données. Il est tout à fait envisageable d'utiliser d'autres types d'informations en entrée de l'algorithme.

Quoi qu'il en soit, TreeMatch a donc besoin d'un modèle d'affinité que nous appellerons *modèle de communication* dans la suite de ce document. Il existe plusieurs moyens, plus ou moins efficaces, permettant de récupérer ce modèle de communication. Nous allons présenter tout d'abord la solution que nous avons retenue puis les autres possibilités connues ou que nous avons envisagées.

2.3.2.1 Exécution préliminaire

Une solution pour déterminer le modèle de communication d'une application parallèle consiste en une exécution préliminaire de cette application au moyen d'une version instrumentée du support d'exécution. Cette technique, bien que difficile à mettre en œuvre, a plusieurs avantages. Tout d'abord, elle permet d'instrumenter les communications dans les couches très bas niveau du support d'exécution. On peut de cette manière récupérer l'intégralité des communications entre processus depuis une implémentation de MPI, ce qu'un outil de trace, par exemple, ne permet pas. Deuxièmement, elle n'implique aucune modification du code de l'application. Le principal inconvénient, en revanche, est bien entendu le coût en temps de calcul d'une exécution préliminaire. Qui plus est, cette exécution doit être effectuée à nouveau dès lors qu'il y a un changement dans les conditions d'expérimentation : nouvelles données en entrée, changement de paramètres. En effet, dans la grande majorité des cas, à contexte équivalent (données en entrées, nombre de processus, paramètres), le schéma de communication ne change pas d'une exécution à l'autre. Notons que dans le cadre de nos expériences, le modèle de communication que nous avons extrait

des applications testées était identique d'une architecture matérielle à une autre. Cependant, un taux important d'appels collectifs peut influencer le modèle de communication en fonction de l'architecture sous-jacente. En effet, la façon dont sera géré le graphe de communication de certaines opérations collectives (`MPI_Gather par exemple`) peut dépendre de la topologie matérielle ou du nombre de processus par unité de calcul.

Open MPI et MPICH étant les deux implémentations libres du standard MPI les plus populaires, nous avons travaillé sur chacune d'elle à un moyen d'extraire le modèle de communication. En moyenne, c'est une centaine de lignes de code dans les couches bas niveau de chacune de ces implémentations (le module Nemesis [15] pour MPICH et le module MCA [1] pour Open MPI) qui ont permis d'ajouter une couche d'instrumentation des communications. Cet ajout nous permet de tracer tant les communications point à point que les communications collectives, ce qui est un avantage face à des solutions de trace standards. Notons enfin que cette instrumentation affecte peu le temps d'exécution de l'application. En effet, comme le montre la figure 2.3, sur trois applications testées avec et sans instrumentation, on peut voir que l'impact de la capture des volumes de communications est estimé à moins de 20%. Pour les noyaux de calcul CG et LU, le déficit de performance avoisine 6%. Pour le noyau FT, cette différence s'établit à près de 16%.

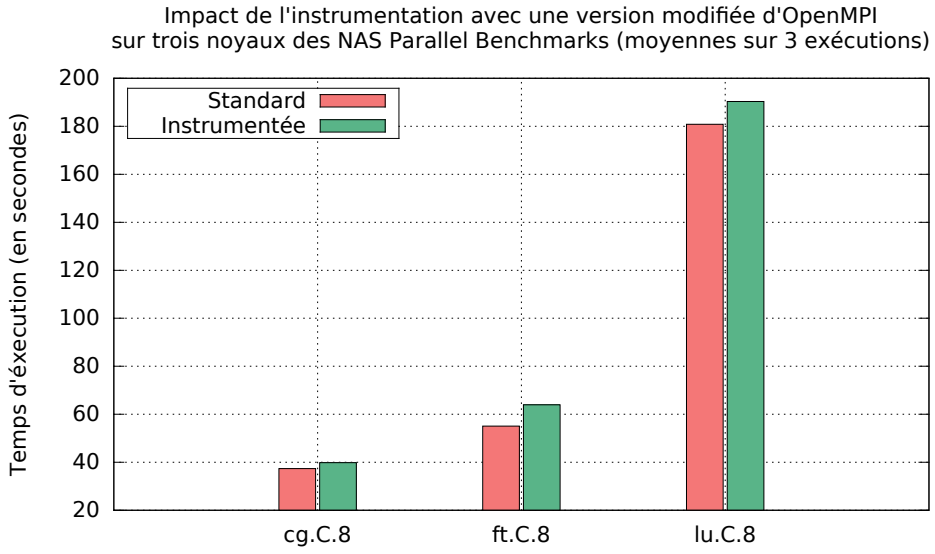


Figure 2.3 – Évaluation de l'impact de l'instrumentation des communications avec une version modifiée de Open MPI sur des noyaux des NAS en classe C sur 8 processus. Les expériences ont été réalisées sur un nœud PlaFRIM embarquant un processeur Intel Xeon X5550.

1. MCA : Modular Component Architecture.

2.3.2.2 Autres techniques

D'autres solutions existent afin d'extraire les échanges de données entre processus. Les outils de trace sont une approche intéressante que nous avons évaluée. EZTrace [76], développé à Inria Bordeaux, est un de ces outils. Il nécessite également une exécution préliminaire de l'application mais n'implique aucune modification du support d'exécution ni même du code de l'application. En revanche, EZTrace n'est capable d'extraire que les communications point à point explicitées dans le code. Les communications point à point résultant du graphe de communication d'une fonction collective ne peuvent pas être interceptées. Les matrices 2.4(a) et 2.4(b) illustrent les différences que l'on peut observer entre un modèle de communication obtenu avec EZTrace ou avec une version instrumentée de Open MPI.

P	0	1	2	3	4	5	6	7
0	1976	6004	6004	0	0	0	0	0
1	6004	1976	0	6004	0	0	0	0
2	6004	0	0	6004	1976	0	0	0
3	0	6004	6004	0	0	1976	0	0
4	0	0	1976	0	0	6004	6004	0
5	0	0	0	1976	6004	0	0	6004
6	0	0	0	0	6004	0	1976	6004
7	0	0	0	0	0	6004	6004	1976

(a) Matrice de communication extraite par EZTrace.

P	0	1	2	3	4	5	6	7
0	1976	**6033**	**6033**	0	**23**	0	0	0
1	**6072**	1976	0	**6033**	**6**	**23**	0	0
2	**6072**	0	0	**6023**	1976	**6**	**29**	0
3	0	**6026**	**6072**	0	0	1976	0	**29**
4	**68**	0	1976	0	0	**6026**	**6026**	0
5	0	**22**	0	1976	**6072**	0	0	**6026**
6	0	0	**22**	0	**6072**	0	1976	**6026**
7	0	0	0	**22**	0	**6026**	**6072**	1976

(b) Matrice de communication extraite par la version instrumentée de Open MPI.

Figure 2.4 – Modèles de communication d'un gradient conjugué (NAS Parallel Benchmarks) exécuté en classe C sur 8 processus sur un nœud PlaFRIM Intel Xeon X5550 (8 cœurs, 8 Mo de cache L3 et 12 Go de RAM par 4 cœurs). Métrique : nombre de messages.

Réduire le temps de l'exécution préliminaire est un objectif pertinent. Des travaux en ce sens ont été effectués [81]. Ils consistent à exclure du code de l'application les portions de calcul de manière à n'exécuter finalement que la partie générant des communications. Cette approche est cependant très complexe à mettre en œuvre.

Certains modèles de programmation fournissent nativement les moyens d'obtenir ces informations. Charm++ en est un exemple puisque chaque entité logicielle communicante est un objet C++ possédant des attributs de charge et de volume de communication. Cette approche sera traitée dans le Chapitre 3.

Enfin, un outil comme Zoltan [18], grâce à un algorithme de partitionnement de maillage est capable d'identifier les frontières entre les différentes sous-parties d'un jeu de données afin d'en déduire les échanges au cours de l'exécution. Associé à une très bonne connaissance de l'application, les valeurs obtenues peuvent être très précises.

2.3.2.3 Représentation du modèle de communication et métriques

L'exécution préliminaire de l'application grâce à un support d'exécution instrumenté nous permet de déterminer un graphe à n sommets, n étant égal au nombre de processus de l'application et m arêtes pondérées en fonction de l'affinité. Dans le cadre de l'utilisation de TreeMatch, nous choisissons de stocker ce graphe sous forme d'une matrice de communication d'ordre n, telle que présentée en figure 2.5.

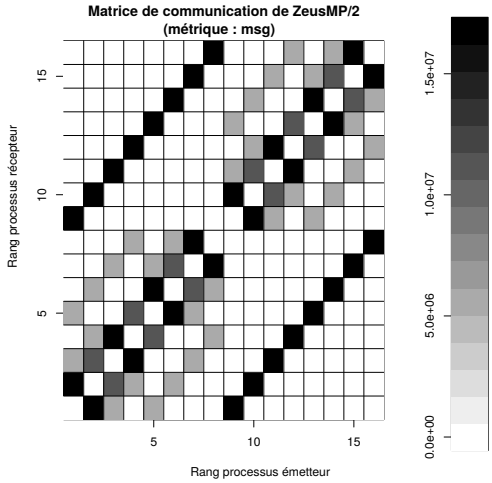


Figure 2.5 – Modèle d'affinité des processus de l'application de mécanique des fluides ZeusMP/2. Le niveau de gris donne le nombre de message échangés par paire de processus.

Nous avons vu que, pour déterminer l'affinité entre processus, nous nous basions sur leurs communications. Cependant, plusieurs métriques permettent d'estimer ces communications. Nous en avons dégagé trois :

— msg : le nombre de messages échangés (au sens MPI) ;
— size : la quantité de données échangées durant l'exécution ;
— avg : la taille moyenne d'un message obtenue à partir des deux mesures précédentes.

Nous avons également travaillé sur une autre métrique basée sur le temps passé dans les communications. Les valeurs ont été récupérées grâce au logiciel EZTrace, qui permet de mesurer avec précision les temps de communication point à point. Cette métrique s'est

avérée peu efficace, même si elle a permis de mettre en lumière des spécificités de l'architecture cible (AMD Opteron 6272, Bulldozer) qui ne sont pas documentées par le constructeur (voir chapitre 1).

2.4 Implémentation

Dans cette section, nous allons présenter notre algorithme de placement de processus TreeMatch. Dans un premier temps, nous détaillerons l'algorithme général puis nous proposerons quelques optimisations.

2.4.1 L'algorithme TreeMatch

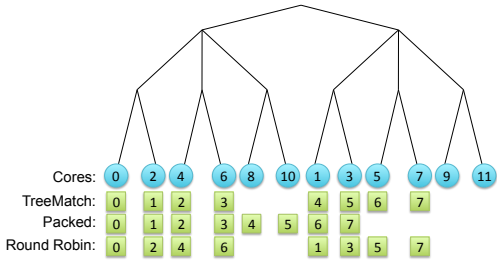
L'algorithme TreeMatch a pour but d'affecter chaque entité logicielle à une unité de calcul afin de prendre en compte l'affinité. Dans le cas de nos métriques, il s'agit de réduire les coûts de communication entre entités. Cet algorithme prend en compte le modèle d'affinité de l'application, mais aussi la complexité de la plate-forme qui accueille l'application.

L'algorithme général est présenté par l'Algorithme 1. Nous allons expliquer par l'exemple son fonctionnement. Pour ce faire, nous nous basons sur la topologie présentée en figure 2.6(a) et sur le modèle d'affinité (matrice de communication) en figure 2.6(b). L'arbre de la topologie est équilibré (toutes les feuilles sont à la même profondeur) et symétrique (tous les nœuds de même profondeur ont la même arité). Il contient quatre niveaux de profondeur et douze feuilles qui correspondent aux unités de calcul. La matrice de communication, de taille 8×8, représente les communications effectuées entre chaque paire d'entités logicielles parmi les huit entités de l'application. Dans cette version de l'algorithme, le nombre de processus doit être inférieur ou égal au nombre d'unités de calcul. Une évolution de cette contrainte sera discutée en chapitre 3.

Pour commencer, l'algorithme parcourt l'arbre vers le haut en commençant par les feuilles, soit au niveau de profondeur 3. À ce niveau, nous appellerons k l'arité du niveau supérieur. Dans notre cas, $k = 2$ et divise l'ordre $p = 8$ de la matrice m. L'algorithme se poursuit donc ligne 6 avec l'appel de la fonction `GroupProcesses`. Cette fonction génère tout d'abord la liste de tous les groupes possibles d'entités logicielles. 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, nous pouvons regrouper l'entité 0 avec l'entité 1 ou 2, jusqu'à 7 et l'entité 1 avec l'entité 2 jusqu'à 7, etc. Plus généralement, nous auront $\binom{8}{2} = 28$ groupes possibles d'entités. Comme nous avons $p = 8$ entités et que nous les groupons par paires ($k = 2$), nous devons déterminer $p/k = 4$ groupes qui n'ont pas d'entités en commun. Pour trouver ces groupes, nous construisons un graphe des incompatibilités entre les groupes (ligne 2 de la fonction `GroupProcesses`).

Algorithme 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]=∅ // Comment les nœuds sont groupés à chaque niveau
2  foreach depth← D − 1..1 do // On commence à partir des feuilles
3      p ← order of m
       // On étend 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)// On groupe les entités par affinité
7      m ←AggregateComMatrix(m,groups[depth]) // On aggrège les communications des groupes
       d'entités
8  MapGroups(T,groups) // Construction de la permutation
```



(a) Arbre de la topologie (les carrés représentent les entités assignées en fonction des différents algorithmes).

Entités	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) Matrice de communication.

Figure 2.6 – Données en entrée de l'algorithme TreeMatch : une représentation de l'architecture et un modèle d'affinité (matrice de communication).

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 // La matrice de communication
   Input: g // Liste des groupes d'entités
          (virtuels)
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 entité. Par exemple le groupe {2,5} est incompatible avec le groupe {5,7} puisque l'entité 5 est commune aux deux groupes. Sous-entendu, l'entité 5 ne peut pas être attachée à deux unités de calcul à la

fois. Dans ce graphe des incompatibilités, les sommets correspondent aux groupes et une arête est présente entre deux sommets si les groupes correspondants sont incompatibles. Dans la littérature, un tel graphe est un complémentaire du graphe de Kneser [50] et se note *Complémentaire de* $KG_{n,k}$. L'ensemble des groupes que nous recherchons est dans un ensemble indépendant de ce graphe, c'est-à-dire un ensemble de groupes qui n'ont aucune arête incidente entre eux. Il s'agit d'une condition indispensable pour ne pas avoir plusieurs entités par unité de calcul. Une particularité intéressante du complément du graphe de Kneser est que si k divise p, les ensembles indépendants maximaux sont de taille 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. Ils dépendent des valeurs contenues dans la matrice. Dans notre exemple, grouper l'entité 0 avec l'entité 5 n'est pas efficace puisqu'elles n'échangent qu'une unité de données. Les grouper aura pour conséquence de reporter un volume important de communications sur le niveau supérieur de l'arborescence, niveau que nous considérons plus coûteux. Pour prendre ceci en compte, nous pondérons le graphe des incompatibilités avec le volume de communication total du groupe réduit du volume de communication interne au groupe. Par exemple, en se basant sur la matrice m, la somme des communications de l'entité 0 est de 1114 et celle de l'entité 1 est de 2104 pour un total de 3218. Si nous groupons ces deux entités, nous devons réduire ce total de 2000, soit le volume de données que l'entité 0 envoie à l'entité 1 et vice-versa. Le groupe se verra ainsi attribuer un poids de $3218 - 2000 = 1218$. Plus ce poids est faible, plus le groupe est intéressant. Malheureusement, trouver un tel ensemble indépendant de poids minimal est un problème NP-Difficile [39]. Par conséquent, nous utilisons des heuristiques afin de trouver un ensemble indépendant adéquat :

— plus petite valeur d'abord : on trie les sommets par ordre croissant de poids et on crée de façon gloutonne un ensemble indépendant maximal ;
— plus petite valeur maximale : on trie les sommets par ordre croissant de poids et on détermine un ensemble indépendant maximal de façon à ce que le sommet de poids maximal soit minimisé ;
— plus grand poids des voisins d'abord : on trie les sommets de façon décroissante en fonction du poids moyen des sommets voisins. Puis, on détermine un ensemble indépendant maximal en commençant par les sommets dont cette valeur est la plus importante [39].

Dans notre implémentation de TreeMatch, nous commençons par la première méthode puis nous tentons d'améliorer la solution obtenue grâce aux deux autres méthodes. Nous pouvons définir un seuil dans le nombre de groupes au-delà duquel la technique du "plus grand poids des voisins d'abord" ne sera pas exécutée. Dans notre exemple, sans tenir compte de l'heuristique utilisée, nous trouvons l'ensemble indépendant de poids minimal {(0,1),(2,3),(4,5),(6,7)}. La figure 2.7 illustre cette sélection. La liste des groupes membres de l'ensemble indépendant est affectée au tableau *group*[3], comme le montre la ligne 6

de l'algorithme TreeMatch. Cela signifie, par exemple, que l'entité 0 et l'entité 1 seront affectées à des feuilles partageant le même nœud parent direct.

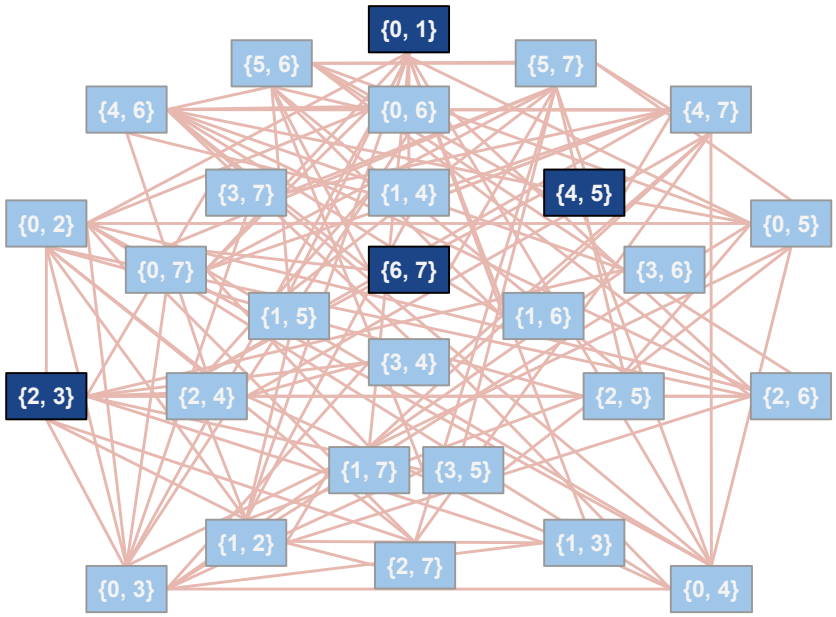


Figure 2.7 – Ensemble indépendant de poids minimal du graphe des incompatibilités. *Complémentaire de* $KG_{8,2}$.

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`. Le but est de calculer les communications restantes entre chaque groupe d'entités. Par exemple, entre le premier groupe (0,1) et le second groupe (2,3), le volume de communication est de 1012 et se voit assigné à $r[0,1]$ (voir figure 2.8(a)). La matrice r de taille 4×4 (nous avons 4 groupes) est affectée à m (ligne 7 de l'algorithme TreeMatch). La matrice m correspond dorénavant au modèle de communication entre les groupes d'entités, que nous appellerons "entités virtuelles". Les étapes suivantes de l'algorithme consistent à grouper ces entités virtuelles 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)

Virt. Ent.	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).

Virt. Ent.	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.

Virt. Ent.	0	1
0	0	412
1	412	

(c) Matrice agrégée (profondeur 1).

Figure 2.8 – Évolution de la matrice de communication aux différentes étapes de l'algorithme TreeMatch.

L'algorithme boucle donc et décrémente la profondeur à 2. Ici, l'arité à la profondeur 1 est $k = 3$ et ne divise pas l'ordre $p = 4$ de la matrice m. Nous ajoutons donc deux groupes artificiels qui ne communiquent avec aucun autre groupe grâce à la fonction `ExtendCommMatrix`. Plus concrètement, nous ajoutons deux lignes et deux colonnes remplies de 0 à la matrice m, incrémentant de ce fait son ordre p à 6. Cette étape illustre le cas où le nombre de ressources est supérieur au nombre d'entités logicielles. La nouvelle matrice est présentée en figure 2.8(b). Une fois cette étape effectuée, nous pouvons grouper les entités virtuelles. Là encore, il nous faut trouver un ensemble indépendant de poids minimal de taille $p/k = 3$. Les groupes créés sont les suivants : {(0,1,4),(2,3,5)} comme le montre la figure 2.9.

Nous agrégeons donc une fois encore les communications restantes dans une matrice de taille 2×2 (voir figure 2.8(c)). L'itération suivante, c'est-à-dire à la profondeur 1, n'offre qu'une seule possibilité pour regrouper les entités virtuelles : (0,1), ensemble qui est affecté à *group*[1]. L'algorithme se poursuit ligne 8. L'objectif maintenant est d'attacher les entités aux ressources. La première phase consistait à remonter dans l'arbre afin de grouper les entités et les entités virtuelles. Ici, nous allons redescendre de la racine vers les feuilles en parcourant le tableau de groupes qui décrit la hiérarchie des groupes d'entités. C'est cette hiérarchie qui va nous donner le placement final. Par exemple, l'entité virtuelle 0 (resp. 1) contenue dans *group*[1] est affectée à la partie gauche (resp. droite) de l'arbre. Si un groupe correspond à un groupe artificiel, aucune entité ne sera affectée au sous-arbre correspondant. À la fin, les entités 0 à 7 sont affectées aux feuilles de l'arbre (sous-entendu,

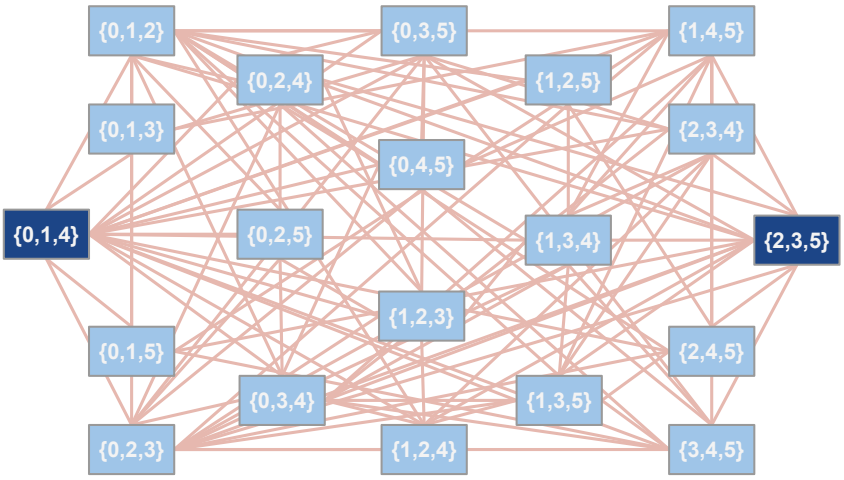


Figure 2.9 – Ensemble indépendant de poids minimal du graphe des incompatibilités entre entités virtuels (profondeur 2 de l'arbre de la topologie).

aux unités de calcul) 0,2,4,6,1,3,5,7 respectivement (voir l'arbre en figure 2.6(a)). Notons que l'algorithme est capable de s'appliquer si le nombre de ressources de calcul est supérieur ou égal au nombre de processus. En revanche, si le nombre d'unités de calcul est plus petit que le nombre d'entités à assigner, TreeMatch ne trouvera pas de solution. Une solution à cette limitation est néanmoins discutée dans le prochain chapitre.

Nous pouvons voir deux autres placements sur la figure 2.6(a). Le premier, que nous appelons *Packed*, consiste à affecter les entités linéairement sur les feuilles de l'arbre. Nous appelons également ce placement glouton "identité logique", dans la mesure où chaque entité i est affectée au cœur logique $\sigma(i)$. Le second, appelé *Round Robin*, distribue les entités une par une sur chaque sous-arbre. Sur certaines architectures, ce placement correspond à une identité physique. Dans notre exemple, TreeMatch offre un bien meilleur placement que ces deux techniques gloutonnes. En effet, "Packed" sépare les entités 4 et 5 des entités 6 et 7. Or, ces deux groupes échangent un volume important de données. Il convient donc de les affecter dans le même sous-arbre. Pour le même type de raisons, "Round Robin" propose une solution nettement sous-optimale.

2.4.2 Optimisations

2.4.2.1 Décomposition des arités de l'arbre

La partie la plus coûteuse de l'algorithme TreeMatch que nous venons de détailler concerne la fonction `GroupProcesses`. Si k est l'arité du niveau supérieur et p l'ordre de la

matrice de communication courante, la complexité de cette portion est proportionnelle au nombre de groupes de taille k parmi un ensemble de p éléments. Ce nombre est $\binom{p}{k} = O(p^k)$. Une optimisation intéressante de TreeMatch est donc de réduire le nombre de groupes possibles dans la fonction `GroupProcesses`.

Pour cela, nous décomposons un niveau de l'arbre ayant une arité importante en un ou plusieurs niveaux d'arité plus faible, la contrainte principale étant que le produit des arités de ces nouveaux niveaux doit être égal à l'arité du niveau original. Par exemple, si un arbre a un niveau d'arité 4, nous décomposons ce niveau en deux niveaux d'arité 2. Ceci multiplie le nombre de fois où la fonction `GroupProcesses` est appelée. Mais, $2 * \binom{p}{2}$ est plus petit que $\binom{p}{4}$ pour $p > 8$. Ainsi, si nous souhaitons placer 8 processus ou plus, la solution de décomposer les niveaux s'avère payante. Plus généralement, nous pouvons décomposer k en un produit de facteurs premiers et procéder à la décomposition d'un nœud de l'arbre d'arité k en différents niveaux ayant pour arité chaque facteur. Les architectures modernes ayant souvent un nombre d'unités de calcul multiple de 2 ou 3, une telle technique aide souvent à réduire k à une valeur raisonnable aux niveaux les plus bas de l'arbre (quand p est important).

2.4.2.2 Accélérer la génération des groupes

Réduire l'arité d'un nœud peut être très avantageux en considérant la complexité de la création des groupes $\binom{p}{k} = O(p^k)$. La solution proposée précédemment convient à bon nombre d'architectures modernes dont le nombre d'unités de calcul peut très souvent être décomposé en produits de 2 et de 3. Ceci réduit alors la complexité de chaque étape de TreeMatch à une complexité quadratique ou cubique. Cependant, même dans ce cas, le coût de ces étapes de calcul peut s'avérer important si TreeMatch a vocation à être utilisé en cours d'exécution (dans un répartiteur de charge par exemple). Qui plus est, nous ne pouvons pas considérer cette spécificité répandue des architectures comme acquise. Par exemple, l'utilisation d'un nombre premier élevé de nœud connectés à un commutateur ne pourrait pas bénéficier de cette amélioration à l'étape concernée. Afin de prendre en compte ces cas, nous avons introduit une méthode plus rapide de création des groupes de processus. La fonction `FastGroupProcesses` décrit cette solution. Elle est exécutée en lieu et place de `GroupProcesses` dès lors que $\binom{p}{k} \geq T_h$, où T_h est un seuil déterminé (30000 par défaut).

Function FastGroupProcesses(T,m,depth)

```
   Input: b //Nombre de conteneurs
   Input: T //Arbre de la topologie
   Input: m //Matrice de communication
1  bucket_list←PartialSort(m, b)
2  return GreedyGrouping(bucket_list,T)
```

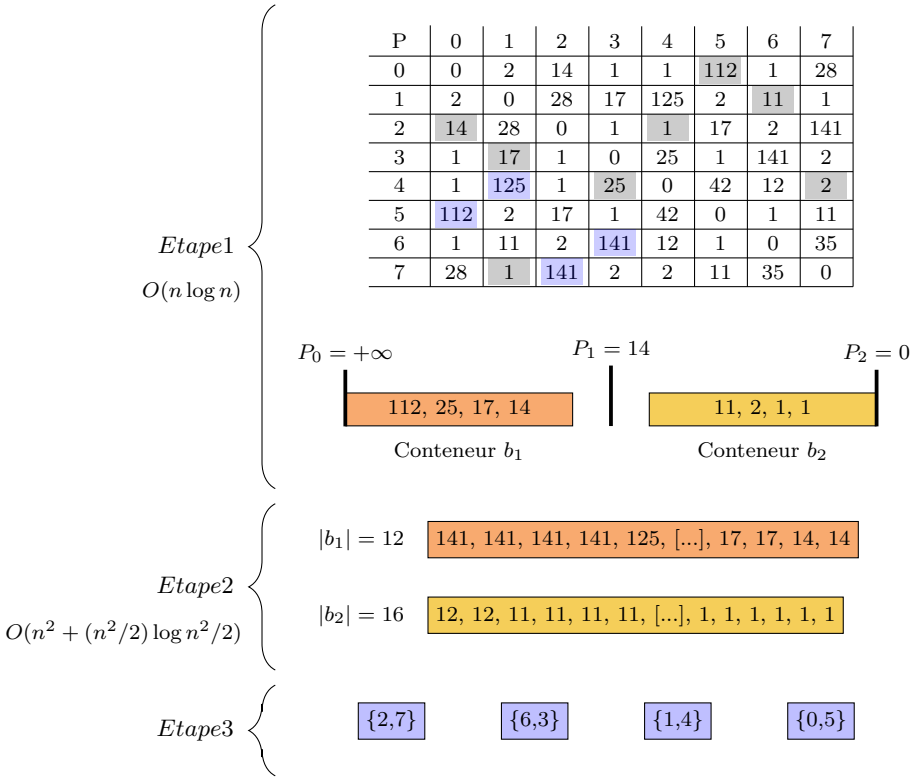


Figure 2.10 – Exemple du fonctionnement de l'algorithme de génération rapide des groupes sur une matrice d'ordre $n = 8$ et en utilisant deux conteneurs. Les valeurs grisées dans la matrice sont les valeurs choisies aléatoirement en première étape. Les valeurs bleues sont celles déterminées par l'algorithme en étape 3.

Son fonctionnement est le suivant : dans un premier temps, un échantillon des éléments de la matrice ($\binom{n^2}{n}$ éléments choisis aléatoirement) est trié en fonction de la valeur de ces éléments. Ces éléments sont ensuite équi-répartis dans b conteneurs (8 conteneurs par défaut). Les plus grands éléments de la matrice sont mis dans le premier conteneur, les plus petits dans le dernier. Cette première étape, de complexité $O(n \log n)$ nous permet de déterminer les pivots p entre chaque conteneur. Nous posons $p_0 = +\infty$ et $p_b = 0$. Cette première étape est illustrée sur un cas de petite taille en figure 2.10. Sur cet exemple, huit éléments sont choisis aléatoirement dans une matrice de communication quelconque d'ordre 8. Ces éléments sont ajoutés dans les deux conteneurs correspondants et le pivot est déterminé comme étant le plus petit élément du conteneur b_1. Un parcours de la matrice, en $O(n^2)$, va déterminer pour chaque élément dans quel conteneur il va être affecté en fonction de sa valeur. Plus formellement, chaque élément v de la matrice est ajouté au conteneur j tel que $p_{j-1} > v \geq p_j$. Il s'agit de la seconde étape présentée en figure 2.10.

Une fois que tous les éléments de la matrice ont été affectés à des conteneurs, nous traitons les éléments conteneur par conteneur en débutant par celui incluant les éléments les plus grands. À cette étape, chaque conteneur contient environ n^2/b éléments. Nous trions ce conteneur (avec une complexité de $O(n^2/b \log(n^2/b))$) et nous en groupons les éléments en suivant l'ordre de tri descendant avec pour seul contrainte qu'un élément ne peut pas être dans deux groupes différents. Ainsi, si un élément a déjà été affecté à un groupe, il est écarté. Cette opération se poursuit en traitant les conteneurs un par un jusqu'à ce que le nombre de groupes attendu soit atteint. Dans notre exemple, les groupes choisis sont exposés en troisième étape de la figure 2.10.

2.5 Expériences

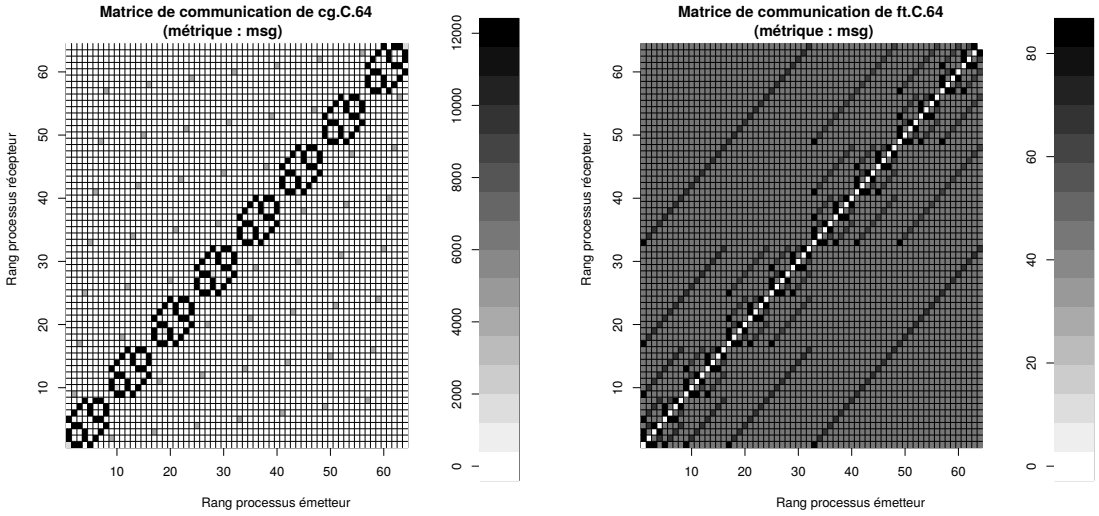
Nous présentons dans cette section les expériences réalisées autour de l'algorithme TreeMatch. Dans un premier temps, nous introduisons les conditions d'expérimentation tant du point de vue matériel que logiciel. Nous exposons dans un second temps les résultats obtenus : les performances de calcul des permutations sont évaluées puis nous montrons l'impact que peut avoir le placement produit par TreeMatch sur plusieurs applications. Nous comparons nos résultats à d'autres méthodes de placement.

2.5.1 Conditions d'expérimentation

Toutes les expériences ont été effectuées sur la plate-forme PlaFRIM. Ce calculateur est composé de 64 nœuds connectés par un réseau InfiniBand (Contrôleur Mellanox Technologies, MT26428 ConnectX IB QDR). Chaque nœud embarque deux processeurs Intel Xeon Nehalem X5550 (2,66 GHz) à quatre cœurs. 8 Mo de cache L3 et 12 Go de mémoire RAM DDR3 cadencée à 1,33 GHz sont partagés entre les quatre cœurs d'un processeur. Le système d'exploitation est une distribution SuSE proposant un noyau Linux en version 2.6.27. Nous avons réservé deux commutateurs InfiniBand QDR et l'intégralité des 16 nœuds par commutateur pour nos expériences. Concernant la partie logicielle, Open MPI en version 1.5.4 et hwloc en version 1.4.1 ont été utilisés.

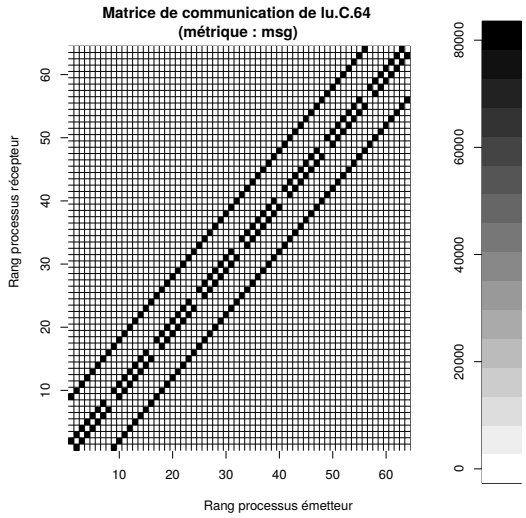
Les premières expériences ont été effectuées sur la suite NAS Parallel Benchmarks [4]. Nous nous sommes attardés en particulier sur trois noyaux de calcul :

— un gradient conjugué (CG) compte tenu de ses accès mémoire et ses communications irrégulières ;
— une transformée de Fourier (FT) étant donné ses communications intensives ;
— une méthode de Gauss-Seidel (LU) dont les accès mémoire sont irréguliers.



(a)

(b)



(c)

Figure 2.11 – Modèles de communication des noyaux CG, FT et LU, exécutés en classe C sur 64 processus.

Pour ces trois noyaux de calcul, nous avons utilisé deux classes (C et D) qui représentent des problèmes de taille moyenne et importante. Les modèles de communication de ces trois noyaux sont présentés en figures 2.11(a), 2.11(b) et 2.11(c). Ces matrices de communication sont issues des exécutions instrumentées de ces noyaux en classe C, sur 64 processus.

Nous avons également travaillé sur le placement de processus dans le cas d'une application réelle, ZeusMP/2 [34]. ZeusMP/2 est une application de simulation dans le domaine de la dynamique des fluides. Plus concrètement, cette application se concentre sur des problèmes comme la dynamique des gaz ou la magnétohydrodynamique (fluides conducteurs de courant électrique). Pour la majorité de ces problèmes, le modèle de communication de l'application est plutôt irrégulier, comme le montre la figure 2.12.

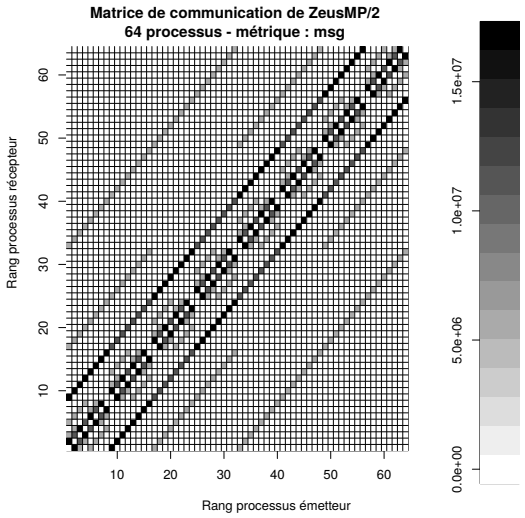


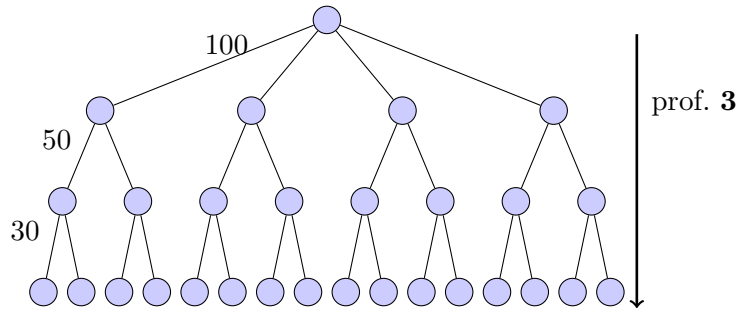
Figure 2.12 – Modèle de communication de l'application ZeusMP/2, exécutée sur 64 processus. Métrique *msg*.

Nous avons comparé TreeMatch à Scotch [21], ParMETIS [47], Chaco [26] et MPIPP [13]. MPIPP a été testé en deux versions : MPIPP 1 et MPIPP 5, version qui consiste à appliquer cinq fois MPIPP 1 afin d'affiner la solution. Nous avons également testé deux placements de processus gloutons : *Packed* (identité logique : le processus i est affecté à l'unité de calcul logique i) et *Round Robin* (identité physique sur les nœuds PlaFRIM : le processus i est assigné à l'unité de calcul physique i). Quelques partitionneurs de graphes sont capables de trouver une solution au problème du placement de processus comme nous l'avons défini (Scotch ou MPIPP par exemple). Pour ParMETIS et Chaco, nous avons implémenté un algorithme de plongement de graphe pour résoudre le problème du placement en tirant avantage de leur algorithme de k-partitionnement. Il est à noter qu'en raison d'un algo-

rithme de grossissement (*coarsening*) Scotch et ParMETIS nécessitent parfois une matrice normalisée. En cas de valeurs trop importantes dans la matrice, des erreurs d'exécution ou de mauvais placements peuvent survenir.

Scotch soulève un autre problème : l'architecture fournie en entrée doit suivre la structure du type de fichier *tleaf*. Dans cette représentation, les arêtes de l'architecture doivent être pondérées. Ces poids sont utilisés dans le calcul du placement. Un exemple est donné en figure 2.13. Le mot clé "tleaf" décrit le type de structure utilisée. La première valeur donne la profondeur de l'arbre de la topologie (3 dans ce cas). La paire de valeur qui suit définit l'arité du niveau 0 (racine) et le poids accordé aux arêtes qui partent de ce niveau. Chaque paire suivante détermine l'arité du niveau inférieur et le poids des arrêtes qui relient chaque sommets aux sommets de niveau inférieur.

tleaf **3** 4 100 2 50 2 30



(a) Fichier au format *tleaf*.

(b) Arbre correspondant.

Figure 2.13 – Exemple du contenu d'un fichier *tleaf* représentant une architecture à 64 cœurs.

Dans nos résultats, nous présentons deux versions de Scotch :

— *Scotch*, qui utilise des poids faibles sur la topologie ($tleaf$ 4 8 4 2 3 2 2 2 1) ;
— *Scotch_w* utilise des valeurs plus importantes ($tleaf$ 4 8 100 2 50 2 20 2 1).

La figure 2.14 montre à ce sujet deux matrices de communication de l'application ZeusMP/2 après application du placement de processus produit par Scotch. Sur la première matrice, les gros volumes de communication sont agrégés sur la diagonale, ce qui signifie que les processus placés sur des unités de calcul physiquement proches échangent beaucoup de données. À l'inverse, la seconde matrice montre des communications extrêmement irrégulières. Les processus de l'application communiquent beaucoup avec des processus physiquement très éloignés. Ce résultat contre-productif est dû à un calcul de placement effectué sur une représentation de l'architecture dont les poids les plus hauts avoisinent 200. La première matrice quant à elle est issue d'une représentation pour laquelle les poids ont été normalisé, c'est-à-dire réduits à un intervalle $[0; 10]$. Ce problème de poids est

une contrainte non négligeable puisqu'il devient alors difficile de se baser sur des mesures expérimentales (bande passante, latence) sans avoir à normaliser les valeurs.

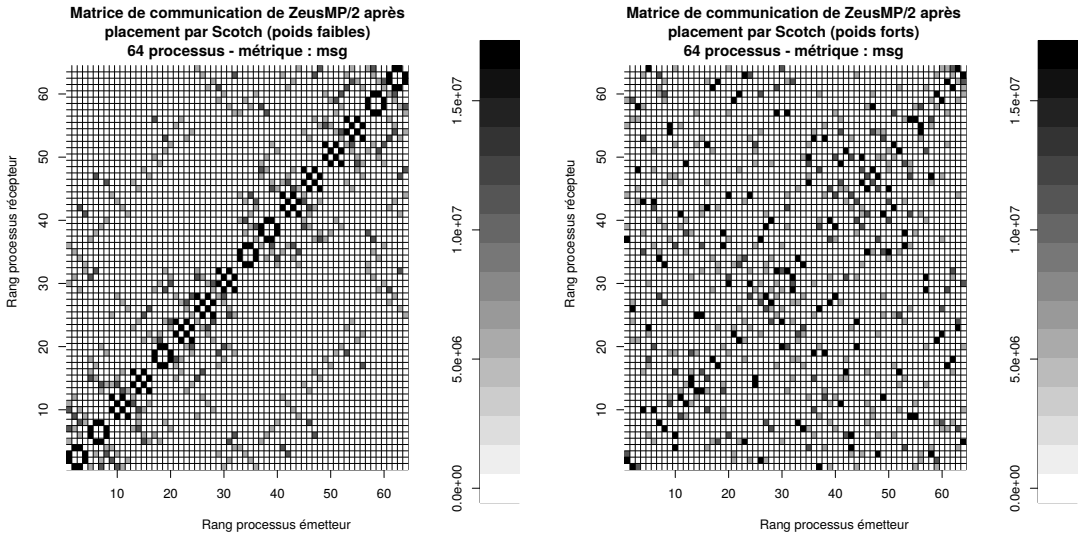


Figure 2.14 – Permutations calculées par Scotch avec des poids faibles (< 10) et des poids plus importants (< 200).

2.5.2 Résultats

2.5.2.1 Temps de calcul de la permutation

TreeMatch a vocation à être utilisé pour du placement statique de processus mais aussi en cours d'exécution. Aussi, il apparaît nécessaire de fournir un bon résultat en un temps qui soit le plus court possible. Nous avons donc mesuré le temps de calcul d'une permutation pour chaque méthode placement évoquée précédemment et TreeMatch. Nous nous sommes basés sur un graphe d'affinité comprenant entre 64 et 16384 sommets (correspondant au même nombre de processus) et sur une topologie représentant 128 commutateurs, connectant chacun 16 nœuds hébergeant deux processeurs à quatre cœurs (soit 16384 unités de calcul). Nous avons dissocié les modèles d'affinité denses, qui correspondent à des applications pour lesquelles tous les processus communiquent au moins une fois avec tous les autres, des modèles d'affinité creux qui correspondent à des applications pour lesquelles les échanges sont ciblés et non collectifs.

Les résultats sont présentés en figures 2.15 et 2.16. Chaque point est une moyenne sur 10 exécutions du temps de calcul du placement en fonction de la taille du graphe. Seuls les temps de calcul sont pris en compte. Les entrées/sorties (charger le graphe, écrire la

solution) sont exclus. Les calculs ont été effectués sur un processeur Intel Nehalem cadencé à 2,66 GHz. Les stratégies de MPIPP n'ont pas été testées pour des graphes dont le nombre de sommets est plus grand que 64. En effet, pour cette valeur, les temps de calcul sont déjà très importants (près de 28 secondes pour MPIPP 5 sur un graphe dense). Notons également que les implémentations utilisant Chaco ou ParMETIS ont provoqué des erreurs au-delà d'un certain nombre de processus.

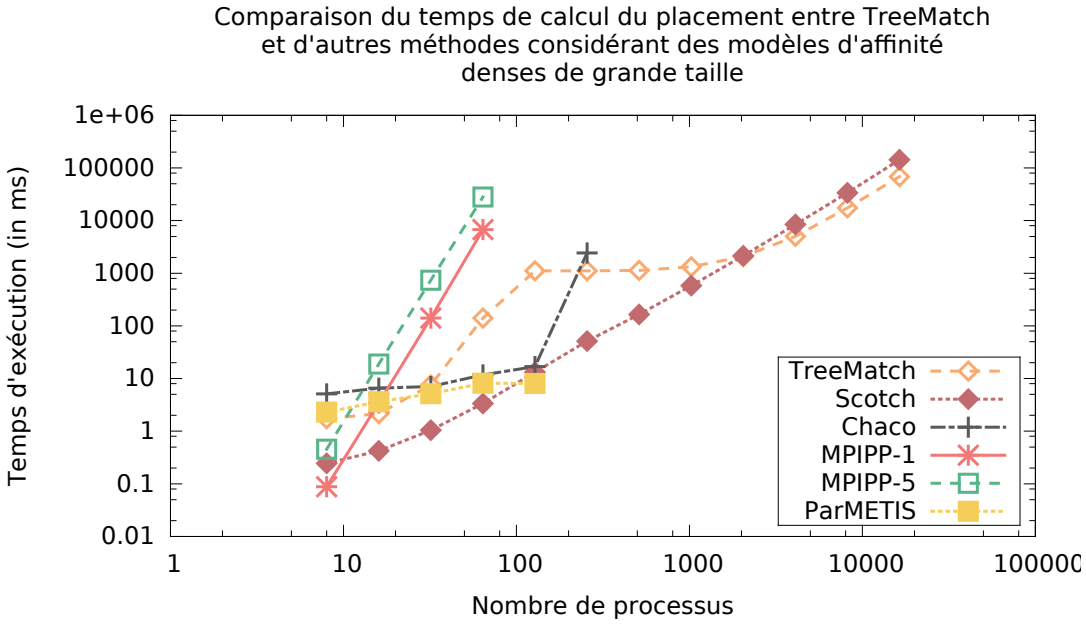


Figure 2.15 – Temps moyen de calcul d'une permutation pour plusieurs méthodes de placement en fonction de la taille du problème considérant un graphe dense.

Sur les deux figures, nous pouvons voir qu'en moyenne, Scotch est la solution la plus rapide. Les échelles des figures étant logarithmiques, nous pouvons également constater que les méthodes MPIPP suivent une progression exponentielle. Une autre remarque commune aux deux expériences concerne TreeMatch. On peut voir sur les deux courbes une inflexion à partir de 128 sommets. Cette inflexion s'explique par le changement de stratégie de TreeMatch pour la création des groupes. Pour les graphes de 128 sommets et au-delà, c'est la version optimisée de création de groupes comme détaillée en section 2.4.2.2 qui est utilisée. Enfin, les deux figures montrent un bon passage à l'échelle de Scotch et TreeMatch sur des graphes importants.

Si nous comparons maintenant les temps de permutation en fonction de la densité du graphe d'affinité, nous pouvons faire plusieurs remarques. Premièrement, il est intéressant de noter que TreeMatch dépend assez peu de cette caractéristique. Notre algorithme est

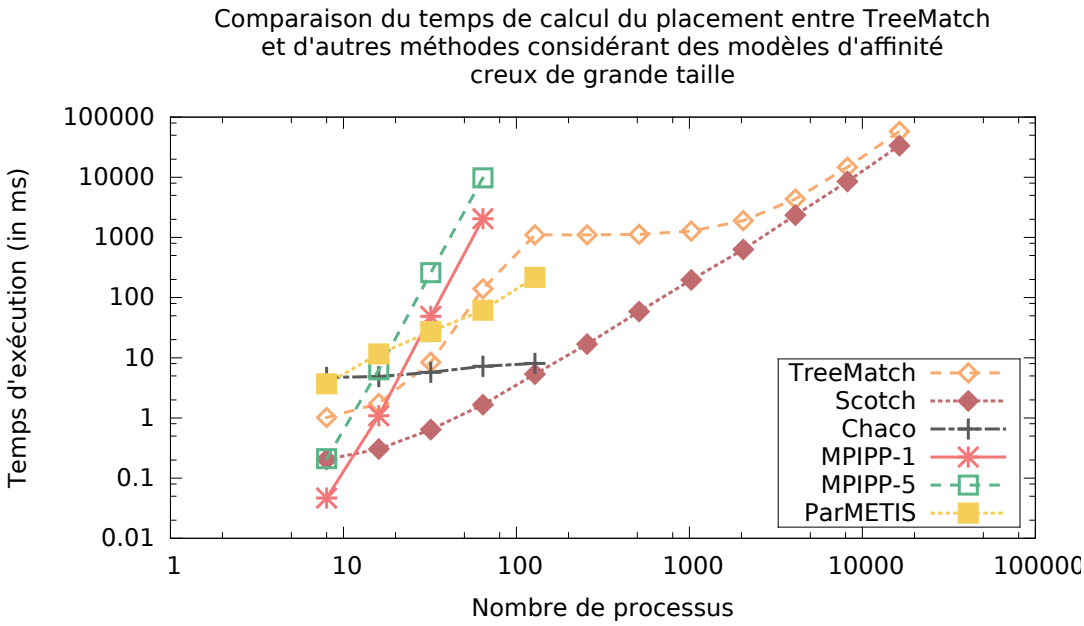


Figure 2.16 – Temps moyen de calcul d'une permutation pour plusieurs méthodes de placement en fonction de la taille du problème considérant un graphe creux.

un peu plus rapide lorsque les graphes de grande taille sont creux même si ce gain n'est pas aussi marqué que d'autres méthodes. Par exemple, pour un graphe à 8192 sommets, le temps de calcul si le graphe est dense est environ 18% plus long que pour un graphe creux. À l'inverse, Scotch dépend fortement de ce critère. Là encore pour un graphe à 8192 sommets, Scotch met 4 fois plus de temps à trouver une solution si le graphe est dense comparé à un graphe creux. MPIPP dépend de ce critère dans les mêmes proportions. Chaco est également plus efficace sur des graphe creux. Sur un cas à 64 sommets, le temps de calcul passe du simple au double. Enfin, ParMETIS montre un comportement différent en étant beaucoup plus efficace sur des graphes denses. Certains résultats atteignent presque un facteur dix.

TreeMatch n'est pas la solution la plus rapide pour les cas de petite taille. Cependant, la qualité de la permutation calculée est souvent supérieure, comme nous allons le voir dans les expériences suivantes. Sur des cas de grande taille, TreeMatch se montre particulièrement efficace et démontre, avec Scotch, une bonne capacité de passage à l'échelle.

2.5.2.2 NAS Parallel Benchmarks

Les expériences conduites autour des noyaux de la suite NAS Parallel Benchmarks nous ont amené à faire le produit cartésien de tous les paramètres (méthodes de placement, métriques, noyaux, classes et nombre de processus), soit $8 \times 3 \times 3 \times 2 \times 4 = 576$ cas différents. Chaque cas a été exécuté dix fois. Il est à noter que pour réduire notre champ de test à trois noyaux, nous avons dans un premier temps étudié les résultats sur tous les noyaux disponibles dans la suite logicielle. Ceci a abouti à une première série de plus de 1700 expériences.

Les figures 2.17, 2.18, 2.19, 2.20 présentent des projections de ces résultats en fonction de différents paramètres et montrent le ratio par rapport à TreeMatch pour chaque méthode de placement. Ainsi, plus le ratio est important, meilleur est TreeMatch. La figure 2.18(b) quant à elle montre les différences du temps d'exécution. Chaque "boîte à moustache" inclut cinq statistiques : la médiane (ligne en gras), le premier et le troisième quartile (boîtes colorées), les moustaches inférieures (resp. supérieures) qui représentent la plus petite (resp. grande) donnée dans 1,5 fois l'écart interquartile du plus petit (resp. grand) quartile et les valeurs aberrantes (ronds gras). Dans ces expériences, nous nous sommes uniquement concentrés sur la version de Scotch utilisant des poids faibles sur les arêtes de la topologie. Enfin, notons que certaines projections n'incluent pas la métrique *avg*. En effet, celle-ci fournit dans beaucoup de cas des résultats très décevants. Il sera fait mention de cette absence en fonction des graphiques (voir figure 2.19).

Sur la figure 2.17, nous pouvons voir que pour chaque noyau de calcul, le placement calculé par TreeMatch donne en moyenne de meilleurs résultats que les placements gloutons *Packed* et *RR* (*Round Robin*). Ceci s'explique par le fait que *Packed* et *RR* donnent de bon résultats essentiellement lorsque les communications importantes sont nativement agrégées sur la diagonale de la matrice de communication. TreeMatch montre de meilleures performances que les autres méthodes de placement. Le meilleur gain est obtenu sur le noyau CG. Ceci est dû au fait que le modèle d'affinité de CG est très irrégulier et montre d'importants volumes de communication loin de la diagonale. Ainsi, le placement calculé par TreeMatch réduit efficacement les coûts associés à ces communications. Concernant le noyau FT, les gains sont plus faibles : la matrice de communication de FT est assez homogène (particulièrement pour la métrique *size*). Le placement a alors un effet relativement faible sur le temps d'exécution. Enfin, le noyau LU se place entre CG et FT en termes de régularité et du positionnement natif des communications sur la diagonale. Les gains de performance suite au placement des processus de LU sont minimes.

Les figures 2.18(a) et 2.18(b) montrent une projection des résultats en fonction de la classe, c'est-à-dire de la taille des problèmes. La classe C génère des problèmes de taille moyenne tandis que la classe D correspond à des problèmes de grande taille. La

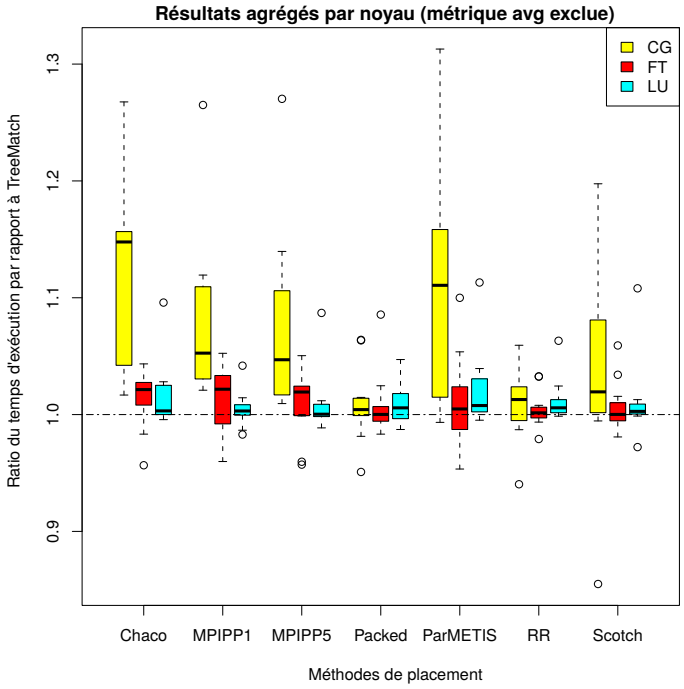
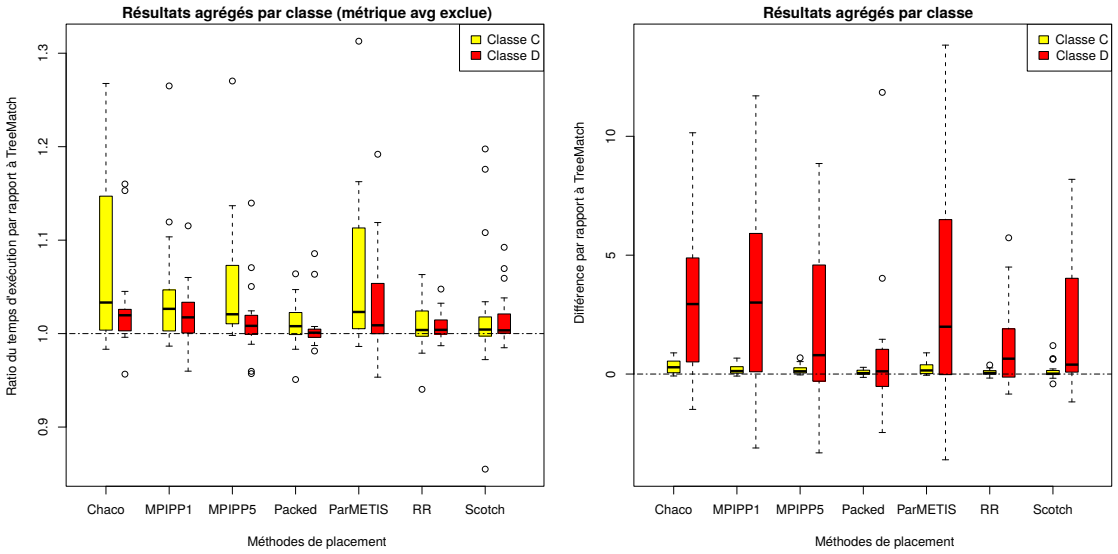


Figure 2.17 – Ratio du temps d'exécution moyen entre TreeMatch et les autres méthodes de placement sur trois noyaux de calcul (NAS Parallel Benchmarks). Résultats projetés par noyaux (LU, CG, FT). Métrique *avg* exclue.

figure 2.18(a) expose le ratio entre TreeMatch et les autres méthodes de placement tandis que la figure 2.18(b) montre la différence de temps entre les exécutions des expériences avec le placement de TreeMatch et avec les autres méthodes. Ces deux graphiques ne prennent pas en compte la métrique *avg*.

En figure 2.18(a), nous pouvons remarquer que le ratio a tendance à diminuer lorsque la taille du problème augmente. En effet, les noyaux que nous testons effectuent des phases de calcul intensif qui sont d'autant plus importantes que le problème en entrée est grand. Sur des problèmes de classe D, les phases de calcul deviennent nettement prédominantes. Le placement de processus n'intervenant que sur la réduction des coûts de communication, l'impact sur la proportion de gain dans le temps total d'exécution devient moins intéressante à mesure que le problème grandit. Cependant, si nous comparons la différence brute de temps entre TreeMatch et les autres méthodes de placement, nous pouvons constater que le gain augmente avec la taille du problème comme le montre la figure 2.18(b).

La figure 2.19 présente les résultats de nos expériences sous l'angle des métriques. Pour rappel, les métriques que nous considérons sont les suivantes :



(a) (b)

Figure 2.18 – Ratio du temps d'exécution moyen et différence de temps entre TreeMatch et les autres méthodes de placement sur trois noyaux de calcul (NAS Parallel Benchmarks). Résultats projetés par classes. Métrique *avg* exclue.

— *msg* : nombre de messages échangés ;
— *size* : volume de données échangées ;
— *avg* : taille moyenne d'un message (*size*/*msg*).

La première remarque que nous pouvons faire est que, excepté pour la métrique *avg* face aux placements gloutons *Packed* et *RR*, toutes les médianes sont supérieures à 1. Concernant les métriques *size* et *msg*, près de 75% des ratios sont au-dessus de 1. Les gains de plus de 10% sur le temps d'exécution sont assez courants, tandis que les dégradations de plus de 10% sont très marginales.

Les plus mauvais résultats sont obtenus quand *Packed* et *RR* sont comparés à TreeMatch utilisé avec la métrique *avg*. Les temps d'exécution sont en général plus importants avec cette métrique qu'avec *size* et *msg*. *Packed* et *RR* ne dépendent pas des métriques utilisées, ni même du modèle d'affinité mais ne se basent que sur les unités de calcul disponibles. Ainsi, il est très souvent préférable de ne pas utiliser cette métrique de la taille moyenne par message. C'est pour cette raison que nous l'avons exclue d'une partie des résultats présentés ici.

Enfin, la figure 2.20 propose ces résultats projetés en fonction du nombre de processus. Le principal élément que nous pouvons tirer de cette représentation est que la différence

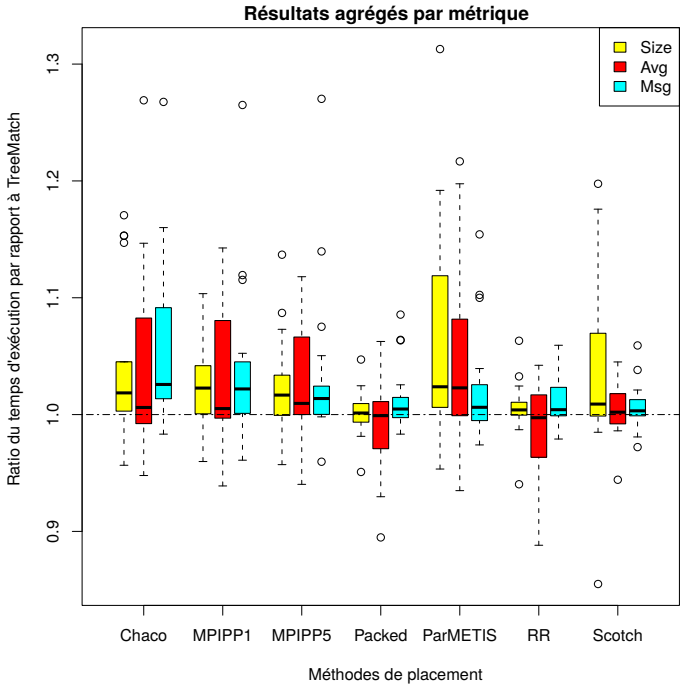


Figure 2.19 – Ratio du temps d'exécution moyen entre TreeMatch et les autres méthodes de placement sur trois noyaux de calcul (NAS Parallel Benchmarks). Résultats projetés par métriques.

de ratio tend à augmenter avec le nombre de processus. La plupart des cas présentés avec un ratio supérieur à 1 ont tendance à conserver un ratio supérieur à 1 lorsqu'on fait varier le nombre de processus. Nous pouvons également noter que plus le nombre de processus augmente, plus l'écart type augmente. Ceci peut s'expliquer par le fait que plus le nombre de processus est grand, plus la part de communication par rapport aux calcul est importante. Ainsi, les gains sont plus visibles et ont davantage d'impact sur le temps total.

Nous remarquons aussi que MPIPP 1 est moins efficace que MPIPP 5, ce qui valide l'approche qui consiste pour MPIPP 5 à affiner la solution fournie par MPIPP 1. Chaco s'avère être la méthode de placement la moins efficace. Enfin, ParMETIS ne garantit pas que les partitions qu'il calcule sont de tailles égales (ce partitionneur n'a pas été conçu pour déterminer ce genre de partitions). Ceci pourrait expliquer les mauvais résultats otbtenus avec cette méthode.

2.5.2.3 ZeusMP/2

Nous allons présenter ici les résultats des expériences de placement de processus réalisées sur l'application ZeusMP/2. Ces résultats montrent les tests utilisant la métrique *msg*, celle-

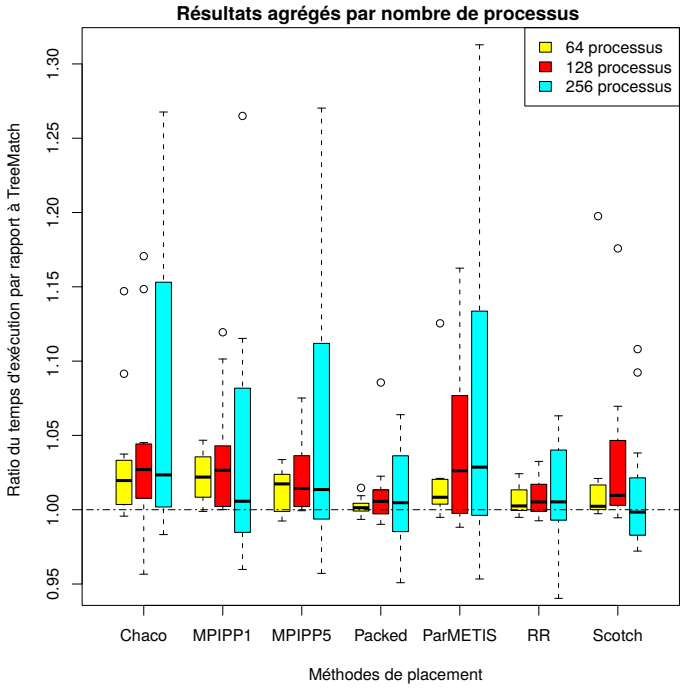


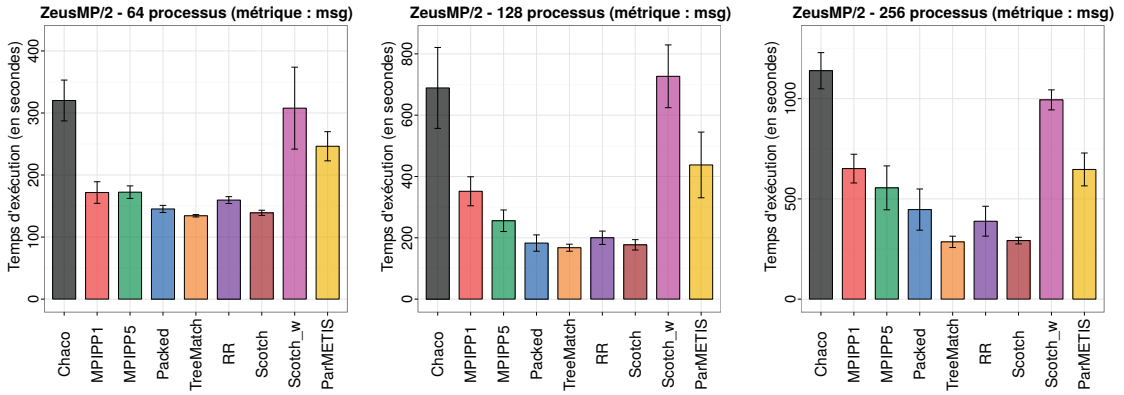
Figure 2.20 – Ratio du temps d'exécution moyen entre TreeMatch et les autres méthodes de placement sur trois noyaux de calcul (NAS Parallel Benchmarks). Résultats projetés par nombre de processus. Métrique *avg* exclue.

ci étant la plus efficace sur cette application. Chaque exécution a été réalisée 10 fois et se base sur 3000 itérations. Les figures 2.21(a), 2.21(b) et 2.21(c) illustrent les résultats pour différents nombres de processus.

Comme nous l'avons vu, le modèle de communication de ZeusMP/2 est plutôt irrégulier. Ceci implique qu'un bon placement de processus peut offrir des gains substantiels en terme de temps d'exécution. Les placements gloutons *Packed* et *RR* donnent malgré tout de bon résultats et se classent troisième et quatrième dans nos expériences. En outre, pour les exécutions sur 256 processus, TreeMatch surpasse *RR* de plus de 25% (285,62 s contre 388,61 s). Les autres méthodes donnent des temps d'exécution plutôt mauvais, en particulier les partitionneurs de graphes comme ParMETIS ou Chaco.

Dans ces résultats d'expériences, nous pouvons voir la différence entre la version de Scotch utilisant des poids faibles et la version utilisant des poids plus importants sur les arêtes de la topologie. Les performances de cette seconde version sont les plus mauvaises de ces résultats. TreeMatch ne souffre pas de cette contrainte puisqu'il se base uniquement sur des caractéristiques de structure de la topologie. De plus, comme nous pouvons le voir en figure 2.23, que nous détaillerons plus loin, le temps de communication entre différentes

unités de calcul n'est pas linéaire en fonction de la taille des messages. Il devient alors difficile de pondérer les liens de la topologie.



(a) (b) (c)

Figure 2.21 – Temps d'exécution moyen de l'application ZeusMP/2 sur 64, 128 et 256 processus (moyenne sur 10 exécutions, métrique *msg*, 3000 itérations).

2.5.3 Discussions

2.5.3.1 L'impact du réseau

En figure 2.22, nous présentons les mesures de temps d'exécution de différents noyaux de calcul des NAS pour les classes C et D. Deux cas sont étudiés. Dans le premier cas (en bleu), tous les nœuds sont connectés au même commutateur. Dans le second cas (en rouge), les nœuds sont répartis équitablement entre deux commutateurs, eux-mêmes reliés entre eux, ce qui entraîne de fait un déficit de performances. Le placement calculé par TreeMatch est le même dans les deux cas. Les résultats montrent néanmoins que l'impact sur les performances, bien que présent, est plutôt faible. Ceci nous laisse penser que considérer le réseau comme plat n'est pas nécessairement contre-productif.

2.5.3.2 De l'approche quantitative

Plusieurs des solutions de placement que nous avons présenté se basent sur une approche quantitative. Sous-entendu, ces solutions utilisent des mesures théoriques ou expérimentales afin de pondérer les arêtes de la topologie. Des mesures de latence ou de bande passante par exemple peuvent permettre de déterminer ces poids. L'approche que nous proposons avec TreeMatch se base, elle, sur une approche dite *qualitative*, c'est-à-dire que nous

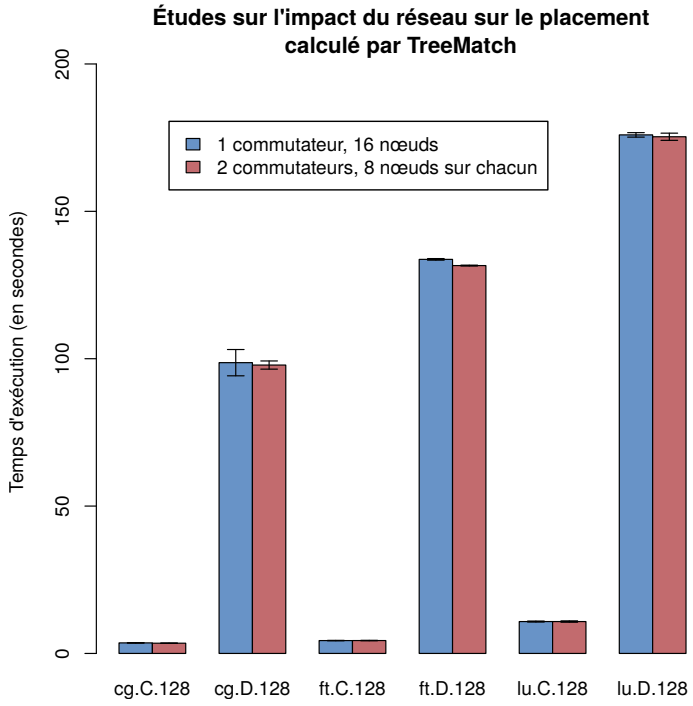


Figure 2.22 – Mesure de l'impact d'un commutateur dans un calcul distribué.

nous basons uniquement sur les caractéristiques de structure de l'arbre de la topologie. Dans le cas d'un arbre équilibré et symétrique, nous nous basons sur la distance entre deux feuilles de l'arbre pour définir le placement des processus d'une application. Ce choix algorithmique n'est pas anodin. La figure 2.23 montre les mesures de bande passante (en Mo par seconde) obtenues en envoyant des messages de tailles différentes. Ces mesures ont été faites sur un nœud de la plate-forme PlaFRIM et grâce à l'outil NetPIPE [73]. Trois cas sont présentés selon que si l'expéditeur et les destinataire partagent le même cache L3, partagent la même mémoire ou sont sur des nœuds différents. La première remarque que nous pouvons faire est que l'ordre "logique" est respecté, quelle que soit la taille du message. La bande passante est toujours la plus importante dans le premier cas (cache L3 partagé) et toujours la plus faible dans le troisième cas (inter-nœud). La seconde remarque est que les performances ne sont linéaires (ni affines) dans aucun des cas. Il apparaît alors difficile de fournir des informations de pondération sur les liens de la topologie tant ces valeurs peuvent dépendre de beaucoup de facteurs (on pourrait à ce sujet également évoquer les problèmes de contention). Ceci renforce notre approche qualitative dans TreeMatch.

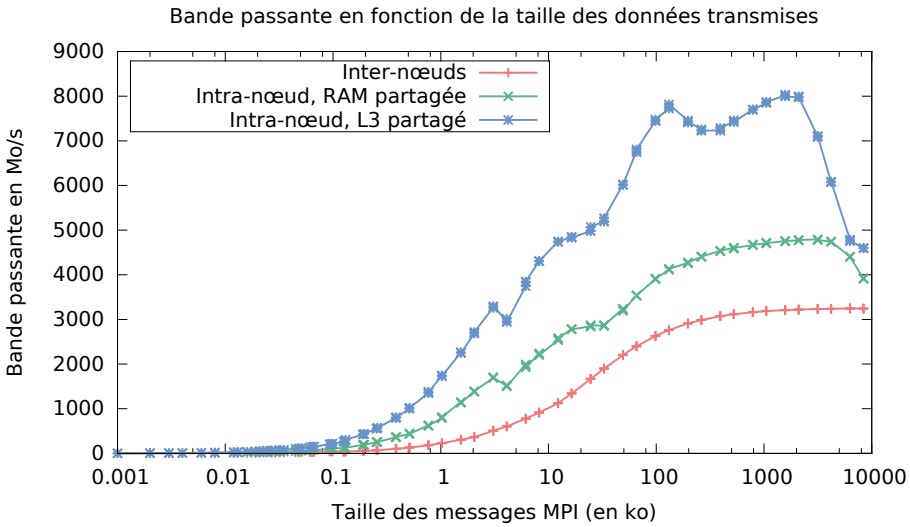


Figure 2.23 – Bande passante en Mo par seconde en fonction de la taille des messages envoyés.

2.6 Conclusion partielle

Parmi les différents leviers possibles pour optimiser l'exécution des applications parallèles, le placement statique a un impact important. Comme nous l'avons vu, plusieurs solutions sont possibles mais montrent souvent des limitations, que ce soit en termes de passage à l'échelle, de généricité ou encore de prise en compte des architectures multi-cœurs.

TreeMatch est capable de répondre en grande partie aux critères que nous avons énoncé dans le tableau 2.1. Cet algorithme est capable de trouver un placement efficace en un temps raisonnable, ce qui ouvre des perspectives pour du placement dynamique. Le chapitre suivant en présentera d'ailleurs une mise en œuvre. Nous avons validé cet algorithme grâce a une importante série d'expériences qui constituent une grande part de mes contributions sur le placement statique. Nous avons vu en particulier que la topologie hiérarchique des architectures multi-cœurs doit être prise en compte. Nous avons pu constater également une corrélation entre l'irrégularité d'un modèle de communication et les gains substantiels que l'ont peut obtenir avec un placement adéquat. Enfin, en se basant sur une métrique de temps de communication, nous avons émis des constatations sur les architectures matérielles : comportement inattendus, numérotation physique des cœurs complexe, etc.

L'algorithme TreeMatch, bien qu'il ait bénéficié de plusieurs améliorations n'est cependant pas exempt de limitations. Parmi elles, nous pouvons noter la prise en compte des cas où les entités logicielles sont en plus grand nombre que les unités de calcul, ou encore

la gestion des contraintes, c'est-à-dire la possibilité de forcer certaines ressources à ne pas être utilisées lors du calcul du placement. Ces limitations sont abordées dans le prochain chapitre et des solutions y sont apportées. La prise en compte de la topologie du réseau reste également un point bloquant. Bien que TreeMatch soit capable de considérer des réseaux arborescents suivants certaines contraintes (équilibre, symétrie), l'algorithme ne peut pas gérer d'autres types de structures comme les grilles ou les tore 3D. L'intégration de LibTopoMap dans TreeMatch pour réaliser cette étape pourrait être un compromis intéressant. Enfin, des améliorations peuvent aussi être apportée en amont et spécifiquement sur la récupération du modèle d'affinité. Un outil de simulation de systèmes distribués comme SimGrid [12] pourrait offrir une perspective à ce sujet. Une autre approche consiste à se baser sur un paradigme capable de fournir ces informations en cours d'exécution. Dès lors, s'il est également possible d'obtenir des informations sur l'architecture matérielle et si des techniques de renumérotation ou de migration des entités logicielles sont fournies, effectuer un placement dynamique des entités d'une application devient envisageable. C'est cette approche que nous allons développer dans le chapitre 3.

Sommaire

Chapitre 3

Équilibrage de charge et placement de processus

Dans le chapitre précédent, nous avons présenté l'algorithme TreeMatch permettant de déterminer un placement adéquat de processus en fonction de leur affinité et de l'architecture sous-jacente. Nous avons également présenté un certain nombre d'expériences montrant l'impact du placement produit par TreeMatch et la comparaison de ces résultats à d'autres méthodes. L'approche que nous avons introduite est dite statique puisque la politique de placement des entités logicielles est fixée au lancement de l'application. Une autre technique, dite dynamique, consiste à modifier ce placement durant l'exécution. Plusieurs paradigmes de programmation offrent cette possibilité : le placement est calculé en cours d'exécution et les rangs des entités logicielles sont réordonnés en fonction.

Dans ce chapitre, nous allons détailler nos travaux autour du placement dynamique. Nous nous baserons pour cela sur le modèle de programmation Charm++. Nous utiliserons également ce modèle pour effectuer de l'équilibrage de charge, tout en tenant compte des nécessités de placement réduisant les coûts de communication. Des expériences, dont certaines à grande échelle, permettront de valider notre méthode mais également de montrer ses limites en fonction du type d'application. Enfin, il est à noter qu'une partie de ce travail

a été réalisée dans le cadre du laboratoire commun Inria-UIUC (*University of Illinois at Urbana-Champaign*) et a donné lieu à plusieurs visites qui ont débouché sur la publication d'un article co-écrit avec des membres du PPL (*Parallel Programming Laboratory*) [B1].

3.1 Problématique et motivation

Structurer les applications et exposer autant que possible leur parallélisme est une solution pertinente, d'ores et déjà proposée par plusieurs modèles de programmation comme Cilk [8] ou Charm++ [42, 43]. Ces modèles se basent sur un parallélisme à grain fin : le découpage de l'application se fait sous forme de petites entités logicielles communicantes, dont le nombre dépasse en général le nombre d'unités de calcul disponibles. Ceci induit de nouvelles problématiques dans l'exécution des applications qui doivent être prises en charge par le support d'exécution. Par exemple, la charge de travail peut varier d'une unité de calcul à l'autre, provoquant un déséquilibre global qui peut impacter fortement les performances de l'application. De ce fait, un équilibrage de charge est en général proposé dans ces environnements, ce qui permet d'améliorer les performances en optimisant l'utilisation des ressources disponibles. Néanmoins, les architectures modernes et leur aspect multi-cœurs impliquent également des contraintes de localité des données qu'il peut être intéressant de prendre en compte au sein de cet équilibrage. Un compromis entre répartition de charge et localité peut être trouvé et adapté en fonction du type d'application.

Nous avons présenté l'algorithme TreeMatch dans le chapitre précédent. Nous allons ici nous intéresser à son utilisation dans le cas du placement dynamique dans le but de réaliser de l'équilibrage de charge.

3.2 État de l'art

Le problème du placement de processus en fonction de la topologie a déjà été étudié [13, 28]. Les différentes implémentations du standard MPI permettent notamment d'affecter les processus sur les unités de calcul au sein du support d'exécution grâce à leur gestionnaires de processus. Depuis septembre 2009, le standard MPI 2.2 définit des fonctions permettant de réordonnancer les processus [29]. Certains travaux se sont ainsi focalisés sur le réordonnancement des processus en cours d'exécution, afin de réduire l'impact des effets NUMA ([57], [9]). Des études se sont davantage orientées vers la prise en compte du réseau afin de minimiser les coûts de communication sur ce niveau de la topologie. C'est le cas de LibTopoMap [28] qui se base sur le standard MPI-2.2 et en implémente l'interface des graphes d'affinité [30], ou de travaux spécifiques à la plate-forme Blue Gene/L et son tore 3D [80]. Cette solution utilise pour le plongement de graphe plusieurs heuristiques comme la bissection récursive, du k-partitionnement, des stratégies gloutonnes ou encore l'algorithme de Cuthill-McKee [14]. Enfin, Dümmler [35] a exploré la voie des applications hybrides à base de MPI et d'OpenMP.

Charm++ [40, 43, 42] est un modèle de programmation basé sur le passage de message et qui utilise l'approche orientée objet et le langage C++. Cependant, là où MPI considère les processus dans son modèle (avec une granularité assez importante), Charm++ est basé sur une granularité plus fine en découpant les calculs en tâches plus petites appelées *chares*. Ces chares, en plus des données qui leur sont attribuées et de leur capacité à envoyer et recevoir des messages de manière asynchrone, sont caractérisés par leur charge CPU, leurs volumes d'entrées et sorties en termes de communication et quelques autres valeurs utiles. Afin de gérer ces entités logicielles et les contraintes qu'elles imposent, un support d'exécution adaptatif est essentiel [41]. Charm++ en propose un, capable de gérer les ressources physiques, les chares et de prendre des décisions sur leur affectation, la tolérance aux pannes ou encore l'équilibrage de charge [46, 45].

Optimiser l'utilisation des ressources physiques est un objectif indispensable pour tirer profit des architectures multi-cœurs actuelles. Les applications parallèles, par exemple dans le domaine de la mécanique des fluides ou de la simulation astronomique ou météorologique, sont de plus en plus complexes et traitent des problèmes aux structures irrégulières qui peuvent aisément créer un déséquilibre de charge CPU. Tous les paradigmes de programmation ne sont pas aptes à prendre en compte cette problématique. Charm++, de par la conception de son support d'exécution, permet d'effectuer de l'équilibrage de charge à différentes fréquences durant l'exécution de l'application [11, 83]. Il est possible d'écrire et de tester l'efficacité d'un algorithme d'équilibrage de façon transparente sans interférer avec le code de l'application. Plusieurs mécanismes d'équilibrage de charge génériques existent nativement dans Charm++ [68] mais ce sont surtout les solutions dédiées à un domaine d'optimisation qui donnent les meilleurs résultats. Notons par exemple des algorithmes d'équilibrage de charge orientés vers la réduction de température des cœurs et de la consommation d'énergie [55]. D'autres méthodes s'attachent à être distribuées en vue d'un bon passage à l'échelle sur des super-calculateurs de très grande taille [54]. Des travaux sur de la répartition de charge hiérarchique (c'est-à-dire à plusieurs niveaux de la topologie) ont également été réalisés [84] [85]. Enfin, des études existent prenant en compte à la fois la charge CPU et la topologie de l'architecture matérielle [6]. Nous pouvons par exemple citer une méthode montrant les gains sur une application de raffinement de maillage d'un placement de chares en fonction de la structure d'un tore 3D [7]. NucoLB et HwTopoLB [65, 64] proposent de l'équilibrage de charge en ne se basant que sur une approche quantitative des liens de la topologie (des informations de bande passante et de latence sont nécessaires). Ces stratégies requièrent de collecter des informations à propos des performances de communication de l'architecture au moyen d'outils appropriés. Notre solution basée sur TreeMatch est à l'inverse davantage automatique puisqu'elle n'utilise qu'une approche qualitative.

Charm++ est un support d'exécution mais aussi un environnement de programmation parallèle dérivé du C++ visant le calcul haute performance [44]. Ainsi, plusieurs applica-

tions sont écrites en Charm++. Le domaine de la dynamique moléculaire avec LeanMD [52] ou NAMD [61], ou encore de la cosmologie avec ChaNGa [38], bénéficient de ce paradigme.

3.3 Solution

TMLB_Min_Weight et TMLB_TreeBased sont deux algorithmes d'équilibrage de charge que nous avons développés pour Charm++. Ils sont capables de répartir la charge sur les unités de calcul d'une architecture tout en tenant compte, grâce à TreeMatch, de la topologie matérielle et de l'affinité des entités logicielles. Ces deux algorithmes ont des objectifs bien distincts. TMLB_Min_Weight est davantage optimisé pour les applications communicant peu. Ainsi, cet algorithme va favoriser une répartition de charge équilibrée tout en essayant de minimiser les communications. La réduction des coûts de communication passant au second plan, cet algorithme tente également de minimiser les migrations de chares. Le second algorithme, TMLB_TreeBased, va quant à lui favoriser l'optimisation des communications. Cet algorithme va être capable de réduire les volumes de communication sur les liens à différents niveaux de hiérarchie dans la topologie, puis de lisser la consommation CPU entre les unités de calcul. Nous allons présenter dans cette section les mécanismes de Charm++ qui entrent en compte dans l'exécution de ces algorithmes. Nous allons détailler ensuite une amélioration importante apportée à TreeMatch faisant intervenir la notion de contraintes sur des unités de calcul.

3.3.1 Charm++

Charm++ est développé au PPL (*Parallel Programming Laboratory*) à l'UIUC (*University of Illinois at Urbana-Champaign*). Il s'agit à la fois d'un modèle de programmation basé sur le langage orienté objet C++, qui suit le paradigme à passage de message, et d'un support d'exécution adaptatif. Le tout cible le développement et l'exécution d'applications parallèles, notamment en fournissant une abstraction de haut niveau de ce parallélisme.

Contrairement à MPI, Charm++ ne manipule pas des processus mais de petites entités de calcul communicantes appelées *chares*. Notons toutefois qu'une implémentation spécifique de MPI, *Adaptive MPI* ou AMPI [31] au sein de Charm++ gère les processus qui eux-mêmes gèrent les chares. Cette implémentation donne ainsi accès aux fonctions du standard MPI. Quoi qu'il en soit, ce découpage à fine granularité implique que le nombre d'objets peut être plus important que le nombre d'unités de calcul. La figure 3.1 montre le fonctionnement d'une application Charm++. On peut y voir un certain nombre de chares, affectés à des unités de calcul physique. Certaines occurrences sont des éléments d'un tableau de chares. Ces chares communiquent entre eux au moyen de messages envoyés et reçus de manière asynchrone et non bloquante. Plus précisément, ces communications se

font grâce à une méthode distante : l'envoi d'un message se fait en réalisant un appel non bloquant du point de réception (fonction d'entrée) du chare destinataire. C'est la brique logicielle *Converse* qui se charge ensuite de gérer les files de messages et de traiter leurs exécutions. Grâce au support d'exécution de Charm++, le nombre d'unités de calcul et le type d'interconnexion n'ont pas d'importance dans l'écriture des applications.

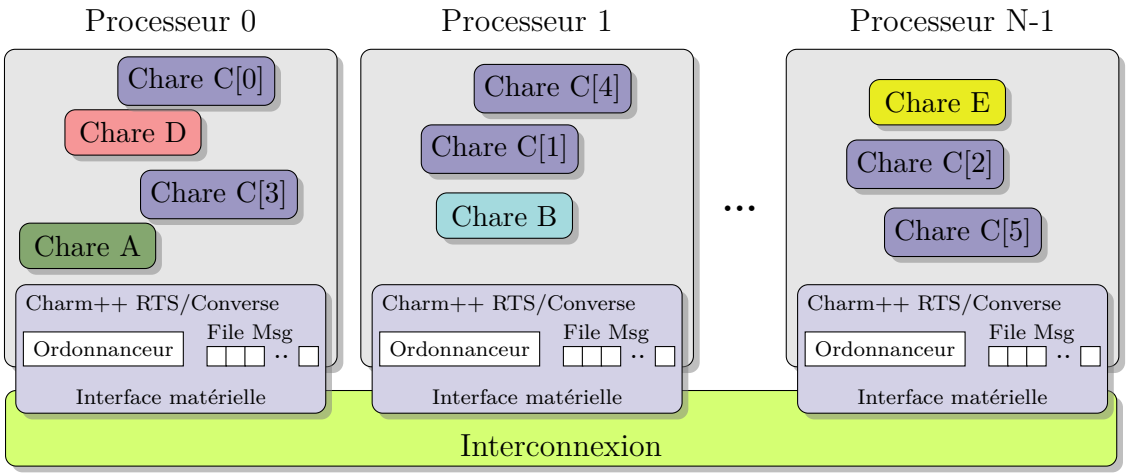


Figure 3.1 – Schéma inspiré de la documentation de Charm++ décrivant le principe général d'une application Charm++.

3.3.1.1 L'équilibrage de charge dans Charm++

L'équilibrage de charge, dans le contexte Charm++, consiste à placer ou migrer des chares ou des ensembles de chares. Ce système s'adapte particulièrement aux applications itératives, répandues dans le domaine de la simulation numérique. Le support d'exécution de Charm++ offre l'avantage de collecter des informations sur les chares durant toute la durée de l'exécution. Ces informations sont stockées dans des structures de données (objets C++) accessibles depuis le code de la méthode d'équilibrage de charge. La figure 3.2 présente ces objets et les méthodes associées. L'objet de type *ProcArray* contient essentiellement les statistiques d'utilisation des unités de calcul (appelées *Pe* ou *Processing Elements* dans la terminologie Charm++). L'objet de type *ObjGraph* donne accès à un graphe de chares. Ces chares sont stockés sous forme de listes d'adjacences : chacun dispose d'une liste de chares destinataires de messages et d'une liste de chares émetteurs. Il est possible pour chaque entité d'obtenir des données statistiques concernant la charge, l'unité de calcul sur laquelle est affecté le chare ou encore les volumes de données échangés avec les chares adjacents. Une méthode de l'objet doit être appelée pour appliquer les décisions de réordonnancement des chares. Il s'agit d'un mécanisme important puisqu'il autorise la manipulation des chares durant toute la durée de l'algorithme d'équilibrage de charge et leur

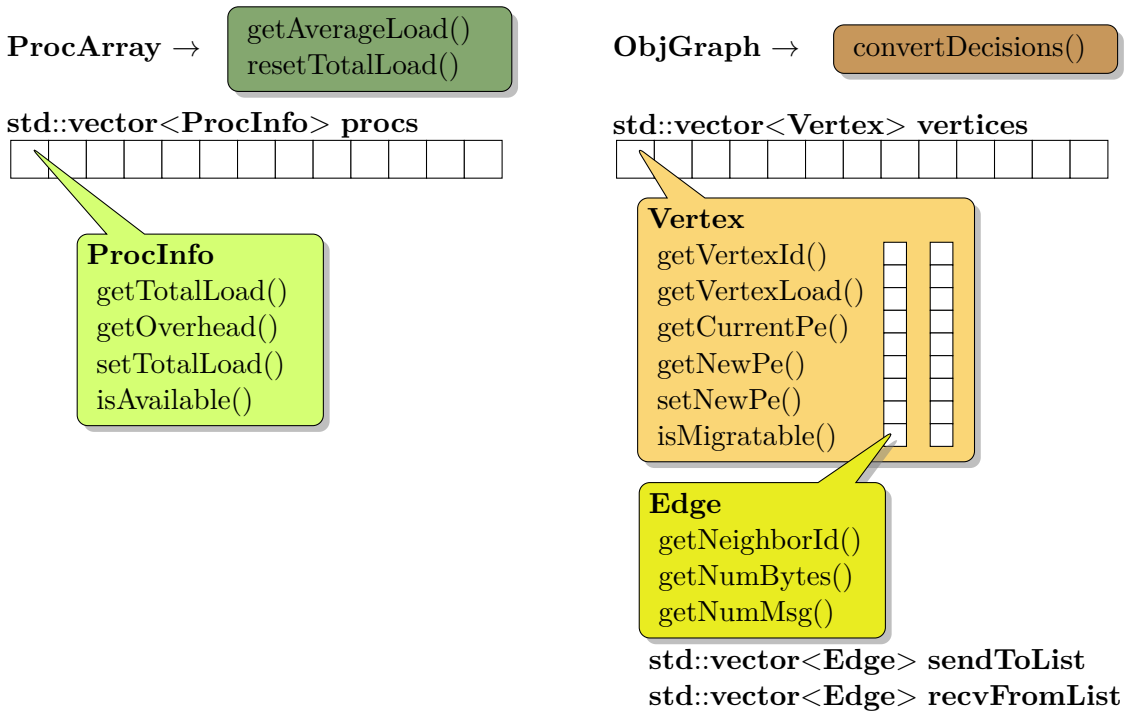


Figure 3.2 – Objets Charm++ fournissant les informations nécessaires à de l'équilibrage de charge.

migration uniquement en fin de procédure. Notons enfin que l'utilisation d'un mécanisme d'équilibrage de charge se définit par l'utilisateur sur la ligne de commande exécutant l'application. L'algorithme souhaité et sa fréquence d'utilisation (toutes les dix itérations d'une application itérative par exemple) sont les paramètres minimums.

3.3.2 Les contraintes dans TreeMatch

Nous allons présenter ici une amélioration de l'algorithme TreeMatch consistant à imposer des contraintes de non-disponibilité sur des ressources de calcul. Comme nous l'avons expliqué, TreeMatch a pour but d'affecter un nombre p de processus sur un nombre n d'unités de calcul tel que $p \leq n$. Lorsque $p < n$ (*c-à-d.* il y a moins de processus que d'unités de calcul), l'algorithme intial de TreeMatch ne permet pas d'intervenir sur le choix des unités de calcul qui seront utilisées. Une telle approche n'entraîne aucun problème particulier dès lors qu'on dispose de machines à mémoire partagée et qu'aucun équilibrage de charge n'est nécessaire. Cependant, nos travaux autour du placement de processus ont pour cible des grappes de nœuds multi-cœurs et ont pour but d'être complémentaires avec

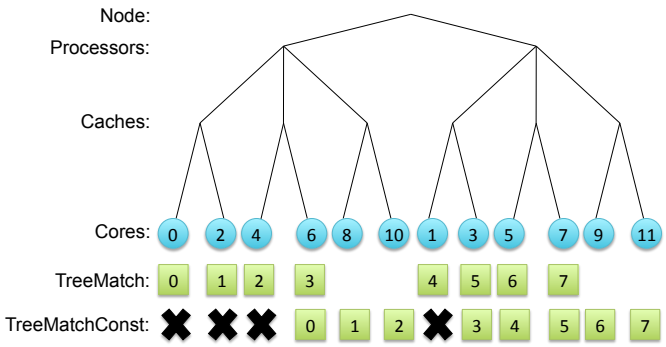


Figure 3.3 – TreeMatchConstraints.

des contraintes de répartition de charge. Afin de s'adapter à ce type d'usage, nous avons travaillé sur une version améliorée de l'algorithme, appelée TreeMatchConstraints, qui permet désormais de prendre en compte des contraintes de manière explicite en listant les unités de calcul qui ne doivent pas être utilisées lors du calcul du placement.

Imaginons qu'un utilisateur souhaite que les unités de calcul numérotées physiquement 0, 2, 4 et 1 ne soient pas utilisées pour certaines raisons. Il peut alors donner ces informations en entrée de l'algorithme, en complément de la description de la topologie et du modèle de communication comme illustré dans l'algorithme 2. Par souci de simplicité, nous posons que le nombre de contraintes plus l'ordre de la matrice de communication est égal au nombre de feuilles de l'arbre de la topologie. Si ce n'est pas le cas, la matrice de communication sera étendue avec des valeurs nulles jusqu'à atteindre la dimension souhaitée et le placement de ces processus fictifs sera ignoré.

Algorithme 2: Algorithme TreeMatchConstraints

Input: T // L'arbre de la topologie
Input: m // La matrice de communication
Input: C // La liste des contraintes
1 $k \leftarrow$ arité du sommet racine de l'arbre T.;
2 $p \leftarrow$constraint_k_partition(k, m, C)// Trouve une partition de taille k en considérant les contraintes
3 tab_m $\leftarrow$ split_com_mat(C, k, p)// Découpe la matrice de communication en k parties en fonciton de la partition trouvée ci-dessus
4 tab_C $\leftarrow$ split_constraints (C, k, T) // Crée un tableau de contraintes de taille k
5 **if** *T n'est pas une feuille* **then**
    // Appeler récursivement TreeMatchConstraints sur les k sous-arbres de la racine de T;
6   **foreach** i *in* $0..k-1$ **do**
7   Appeler TreeMatchConstraints sur le i^{th} sous-arbre de T, tab_m[i], tab_C[i].
8   $r \leftarrow$ aggrège les résultats de chaque sous-arbre;
9 **else**
10   $r \leftarrow$ assigne les processus/contraintes a T;
11 retourne r comme résultat pour T;

TreeMatchConstraints est un algorithme récursif. Dans notre exemple, pour le premier appel nous avons comme donnée en entrée T, l'arbre de la topologie représenté en figure 3.3, m la matrice de communication en figure 3.4(a) et $C = \{0, 4, 2, 1\}$ la liste des contraintes. Soit k l'arité de l'arbre T à l'étape de récursivité correspondante ($k = 2$ au premier appel). En ligne 2 de l'algorithme, nous effectuons un *k-partitionnement* de la matrice de communication en tenant compte des contraintes imposées. Les contraintes définissent que trois feuilles du sous-arbre gauche et une feuille du sous-arbre droit ne peuvent pas être utilisées. Nous devons donc partitionner la matrice de communication de telle façon que trois processus soient attribués au sous-arbre gauche et cinq au sous-arbre droit. Le découpage de ces partitions doit se faire en minimisant au maximum les communications entre elles (coupe minimale). Malheureusement, aucun partitionneur de graphe n'est capable de fournir en même temps un partitionnement déséquilibré et dont le nombre de sommets de chaque partition est déterminé. Par exemple, ParMETIS [47] ne peut pas garantir qu'une partition donnée aura exactement la taille souhaitée. Par conséquent, nous avons implémenté un algorithme probabiliste très simple de k-partitionnement afin d'effectuer cette tâche. Compte tenu de son aspect aléatoire, l'algorithme en question ne fournit pas une excellente solution. Nous l'exécutons donc dix fois et conservons le meilleur partitionnement. Dans notre exemple, les premiers trois processus sont affectés au sous-arbre de gauche et les cinq derniers à celui de droite. Nous pouvons alors appeler récursivement TreeMatchConstraints avec les nouvelles entrées de chaque sous-arbre. Par exemple, pour le sous-arbre de gauche, nous avons seulement la matrice de communication correspondant aux trois premiers processus (comme illustré en figure 3.4(b), l'arbre T est l'arbre commençant au niveau des "processeurs", l'arité est $k = 3$ et la liste des contraintes $C = 0, 4, 2$. Une fois que l'algorithme atteint le bas de l'arbre, les résultats peuvent être agrégés. Dans notre exemple, le résultat est présenté en figure 3.3.

Il est important de noter que, quelle que soit la version de l'algorithme de TreeMatch utilisée (avec ou sans contraintes), seules les informations de structures de la topologie sont nécessaires. À aucun moment, l'algorithme n'a besoin d'informations quantitatives sur l'architecture cible (vitesse du bus de communication, bande passante, latence, etc) contrairement à d'autres approches comme Scotch [20] ou NucoLB [64].

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

(a) Matrice de communication.

Proc	0	1	2
0	0	1000	10
1	1000	0	1000
2	10	1000	0

(b) Matrice de communication du sous-arbre gauche

Figure 3.4 – Matrices de communication durant l'appel à TreeMatchConstraints

3.4 Implémentation

3.4.1 Équilibrage de charge pour les applications à calcul prédominant

Comme expliqué précédemment, certaines applications sont très déséquilibrées d'un point de vue de la charge CPU. Malgré tout, ces applications produisent un certain volume de transfert de données qu'il peut être intéressant de prendre en considération. Notre but ici est de trouver un compromis entre ces deux caractéristiques, c'est-à-dire de rééquilibrer la charge afin d'optimiser l'utilisation des ressources et réduire le temps d'exécution de l'application, mais aussi de minimiser autant que possible les coûts de communication pour un objectif similaire. L'algorithme TMLB_Min_Weight est capable de répartir la charge des chares tout en conservant ceux qui s'échangent le plus de données les plus proches possible les uns des autres dans la topologie. Cependant, bien que cette solution offre déjà des améliorations intéressantes, l'impact de la migration des chares d'une unité de calcul à une autre n'est pas négligeable. Notre algorithme prend également en compte ces migrations. L'objectif de TMLB_Min_Weight est donc triple : équilibrer la charge CPU des chares, réduire les coûts de communication et minimiser les migrations. La résolution du problème de couplage de poids minimal nous permet d'appréhender ce dernier objectif.

Algorithme 3: L'algorithme TMLB_Min_Weight

```
   Input: m_chares La matrice de communication entre chares
   Input: n Ordre de m_chares
   Input: fake_T Une fausse topologie arborescente à n feuilles
   Input: m_mig La matrice de migrations entre chares
 1 p ←Permutation(m_chares, fake_T);
 2 SortEachPartDesc(p, nb_cores);
 3 foreach i in 0..n do
 4     c ←LessLoadedCore();
 5     chare ←ChooseChare(c);
 6     AssignChareOnCore(chare, c);
 7     UpdateMigrationMatrix(m_mig, chare)
 8
 9 h ←HungarianAlgorithm(m_mig);
10 foreach chare do
11     SetNewPe(chare, h)
```

Ce premier algorithme est présenté en algorithme 3 et son illustration est montrée en figure 3.5. Afin d'en expliquer le fonctionnement, considérons une application qui crée cent chares et qui est exécutée sur quatre unités de calcul. Après quelques itérations, Charm++ appelle l'algorithme d'équilibrage de charge défini sur la ligne de commande. Dans un premier temps, le modèle de communication est extrait de l'instrumentation de l'application (fournit par Charm++ pendant l'exécution). Le résultat est une matrice d'ordre 100. Nous créons ensuite une fausse topologie arborescente en décomposant l'ordre de la matrice en produit de facteurs premiers. Dans notre exemple, l'arbre de la topologie sera : 2:2:5:5 (cette

notation définit l'arité de chaque niveau de l'arbre en commençant par la racine). Nous exécutons alors TreeMatch afin de trouver une bonne permutation des chares réduisant les coûts de communication (ligne 1 de l'algorithme). Une fois cette permutation obtenue, nous la découpons équitablement en nombre de parts égal au nombre d'unités de calcul sur l'architecture cible (quatre, dans notre exemple). Dans ce cas, chaque part correspond au groupe de chares de chacun des quatre sous-arbres du second niveau de la fausse topologie. Ces groupes de chares sont ensuite triés de façons décroissante en fonction de leur charge. Cette étape correspond à la ligne 2 de l'algorithme.

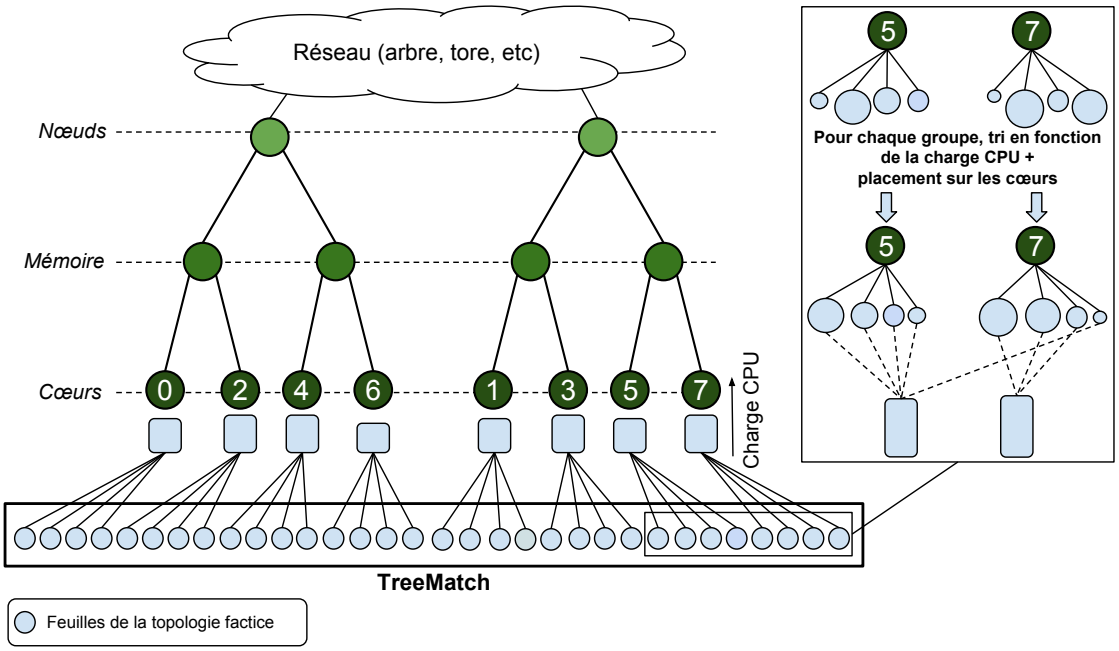


Figure 3.5 – Fonctionnement de l'algorithme TMLB_Min_Weight. Les chares sont d'abord réordonnés en fonction de leur schéma de communication et d'une topologie factice. Ils sont ensuite attribués par groupe à des unités de calcul en équilibrant l'utilisation CPU grâce aux chares les moins chargés.

La boucle principale va ensuite affecter chaque chare à l'unité de calcul correspondante en considérant à chaque itération l'unité de calcul la moins chargée et le chare le plus chargé comme le montrent les lignes 3 à 7. Quand une unité de calcul a reçu tous les chares de sa part mais reste malgré tout l'unité la moins chargée, nous sélectionnons un chare d'une autre part (en commençant par la plus proche). Cet algorithme permet de conserver les chares communicants le plus ensemble et d'appliquer une répartition de charge fine en utilisant les chares possédant les plus petites charges. De plus, dès qu'un chare est affecté, nous mettons à jour une matrice de migration en incrémentant $m\_mig[old\_core][new\_core]$ lorsque le chare a été déplacé de l'unité de calcul old_core vers l'unité new_core (ligne 7

$$\begin{pmatrix} 8 & 5 & 5 & 3 \\ 6 & 9 & 7 & 9 \\ 3 & 8 & 11 & 4 \\ 8 & 3 & 1 & 10 \end{pmatrix}$$

Figure 3.6 – Matrice m_mig de migrations de chares telle que $m\_mig[i][j]$ donne le nombre de migrations dans le cas où le groupe i est assigné à l'unité de calcul j.

de l'algorithme).

À la fin de cette phase, nous avons autant de groupes de chares que d'unités de calcul et ces groupes ont des sommes de charge les plus proches possibles. Reste la question de savoir comment réellement attacher ces groupes de chares sur les unités de calcul physiques. Nous résolvons ce problème grâce à notre algorithme de minimisation des migrations (ligne 9). En effet, l'objectif est alors d'affecter les groupes de chares de façon à réduire autant que possible les mouvements d'objets. Minimiser les migrations revient à résoudre un problème de couplage de poids minimal sur la matrice de migrations. En effet, chaque entrée $m\_mig[i][j]$ donne le coût de migration dans le cas où on alloue le groupe i sur l'unité de calcul j. Un couplage de poids minimal de cette matrice donne l'affectation des groupes sur les unités de calcul de telle façon que la somme des migrations soit minimisée. Trouver un tel couplage peut être effectué en temps polynomial par l'algorithme hongrois [58].

3.4.1.1 La Méthode Hongroise

Prenons en exemple la matrice de migration présentée en figure 3.6, qui illustre le type de matrice que l'ont pourrait obtenir une fois les groupes finaux constitués. Cette matrice est une représentation du graphe biparti complet exposé en figure 3.7. L'algorithme hongrois consiste donc à trouver un couplage de poids minimal de ce graphe. Plus concrètement, l'objectif est de trouver un ensemble d'arêtes indépendantes tel que l'un des sommets incident appartienne au sous-ensemble *groupes*, l'autre sommet incident appartienne au sous-ensemble *cœurs* et la somme des poids des arêtes soit minimale. Un résultat optimal possible correspond aux arêtes colorées en noir sur la figure 3.7. Les couples choisis sont {0,2}, {1,1}, {2,3} et {3,0}, ce qui signifie qu'afin de minimiser les migrations, il est avantageux d'affecter le groupe 0 sur l'unité de calcul 2, le groupe 1 sur l'unité 1, etc. Une autre solution de poids identique (16 dans cet exemple) est également possible en formant les couples {0,1}, {1,0}, {2,3} et {3,2}. La matrice en figure 3.8 montre la sélection effectuée sur la matrice initiale.

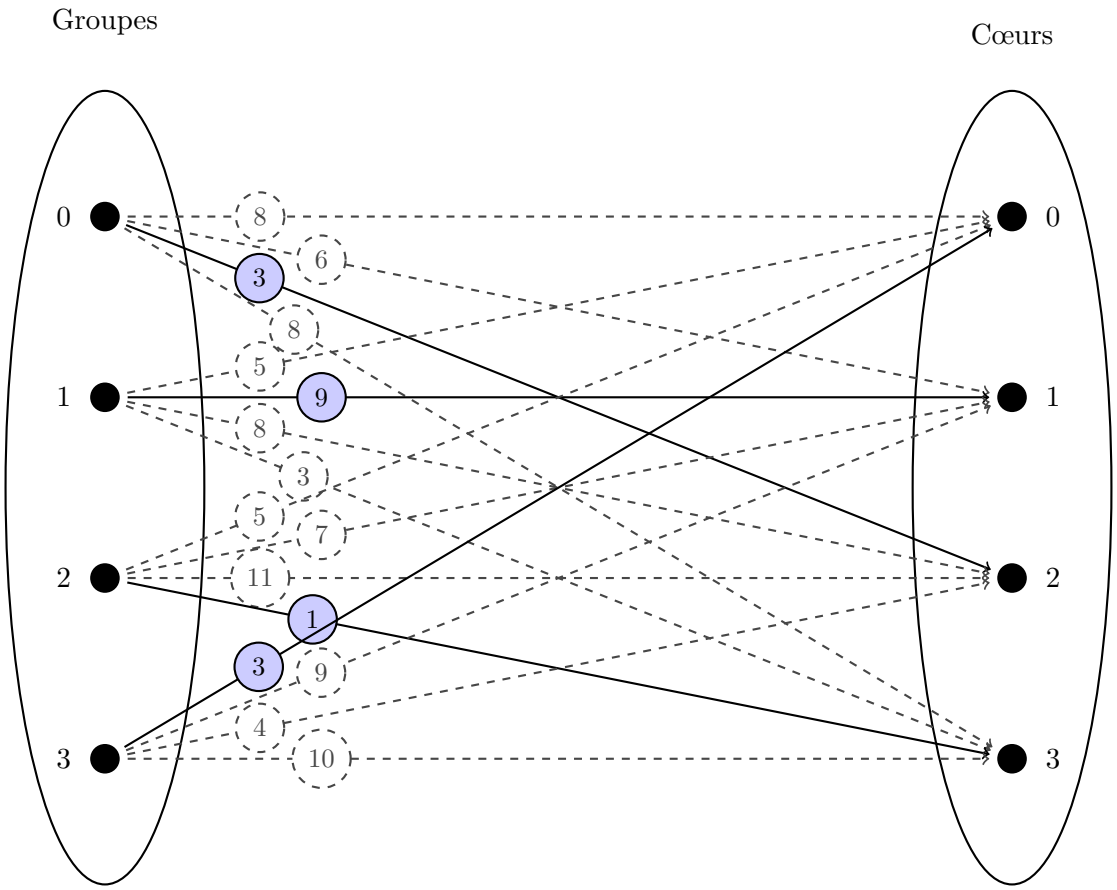


Figure 3.7 – Couplage de poids minimal d'un graphe biparti déterminé grâce à l'algorithme hongrois.

$$\begin{pmatrix} 8 & 5 & 5 & \boxed{3} \\ 6 & \boxed{9} & 7 & 9 \\ \boxed{3} & 8 & 11 & 4 \\ 8 & 3 & \boxed{1} & 10 \end{pmatrix}$$

Figure 3.8 – Couplage de poids minimal déterminé grâce à l'algorithme hongrois représenté sur la matrice de migration.

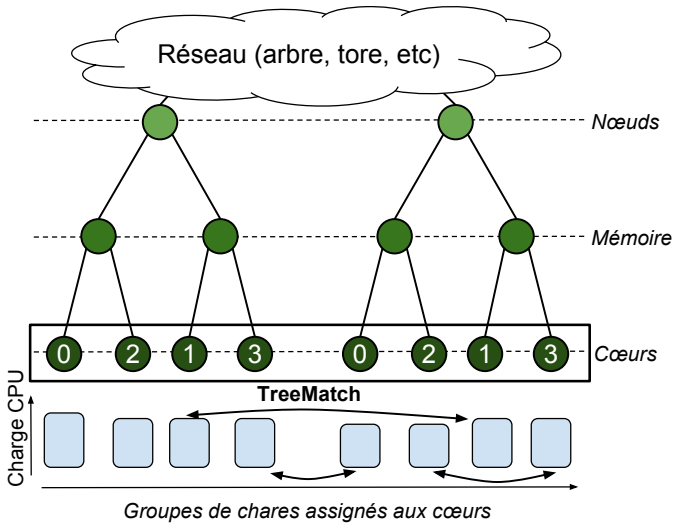
3.4.2 Équilibrage de charge pour les applications fortement communicantes

L'algorithme présenté précédemment est optimisé pour les applications dont les phases de communication sont peu importantes. Dans cette partie, nous allons nous pencher sur la répartition de charge des applications fortement communicantes. Nous avons développé un mécanisme d'équilibrage de charge basé sur TreeMatch qui va donner la priorité à la réduction des coûts de communication tout en tâchant de niveler la charge CPU. L'algorithme général est présenté en algorithme 4 et une illustration de ses différentes étapes est présentée en figure 3.9. Nous allons détailler cet algorithme par l'exemple puis nous reviendrons sur les améliorations que nous lui avons apportées afin d'en améliorer le passage à l'échelle.

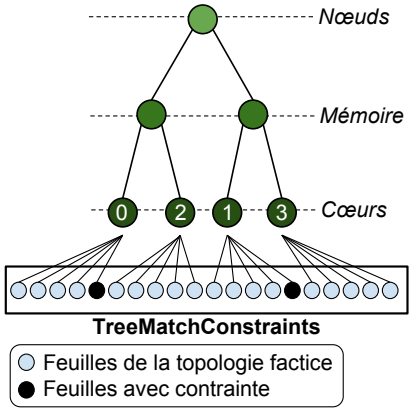
Algorithme 4: L'algorithme TMLB_TreeBased

```
   Input: m_pu La matrice de communication entre Pe (unités de calcul)
   Input: T L'arbre de la topologie
2  p ←Permutation(m_pu, T);
3  foreach chare do
4      SetNewPe(chare, p)
5
6  foreach node in parallel do
       Input: m_chares La matrice de communication entre chares dans le nœud courant
       Input: n Order of m_chares
7      fake_T ←CreateFakeTopology();
8      p ←PermutationWithConstraints(m_chare, fake_T);
9
10     foreach i in 0..n do
11         c ←LessLoadedCore();
12         chare ←ChooseChare(c);
13         AssignChareOnCore(chare, c);
```

Considérons une application découpée en 110 chares et exécutée sur deux nœuds de calcul, chacun hébergeant quatre cœurs. Au début de l'exécution, Charm++ groupe les chares sur les différents cœurs. À chaque appel de l'algorithme d'équilibrage de charge (toutes les n itérations), la première partie de l'algorithme consiste à extraire le modèle de communication de ces groupes de chares, à récupérer les informations de topologie et à appliquer l'algorithme TreeMatch afin de permuter ces groupes de chares. Cette étape correspond aux ligne 2, 3 et 4 de l'algorithme 4. Un exemple est également présent en figure 3.9(a). Les coûts de communication entre les cœurs sont ainsi minimisés. Dans les améliorations que nous avons apportées par la suite à cet algorithme, il est inclus la possibilité d'effectuer cette étape à différents niveaux de hiérarchie. En effet, sur des exécutions à grande échelle où le nombre de cœurs est très important, le temps de calcul de la permutation effectuée par TreeMatch peut être trop coûteux. Il est alors intéressant de considérer les groupes de chares affectés à un niveau supérieur de hiérarchie. Par exemple, exécuter cette étape en prenant en compte les groupes de chares sur les nœuds permettra



(a) Les groupes de chares assignés aux cœurs de la plate-forme sont permutés grâce à TreeMatch



(b) Sur chaque nœud de calcul, les chares sont réordonnés avec l'algorithme TreeMatchConstraints sur une topologie factice

Figure 3.9 – Illustration des étapes du fonctionnement de l'algorithme d'équilibrage de charge TMLB_TreeBased

de produire un premier placement dans un temps raisonnable qui réduira les coûts de communication au niveau du réseau. Ce principe consistant à agir à différents niveaux de l'arborescence classe notre mécanisme d'équilibrage de charge dans la catégorie des mécanismes hiérarchiques, à l'instar d'autres travaux [65, 84, 85].

La seconde partie de l'algorithme, à partir de la ligne 6, intervient à un niveau supérieur ou égal dans la topologie. Dans notre exemple, nous remontons au niveau des nœuds dans la hiérarchie. Pour chaque nœud de notre architecture, nous appliquons un algorithme d'équilibrage de charge local. Par exemple, considérons qu'après la première étape 55 chares ont été affectés à chaque nœud (répartis sur les quatre cœurs). Afin d'effectuer l'étape de répartition de charge, nous nous basons sur le modèle de communication de ces chares et sur une topologie factice sur laquelle il sera possible d'affecter les chares (ligne 7). Ceci implique donc une topologie dont le nombre de feuilles est supérieur ou égal au nombre de chares concernés. L'algorithme TreeMatch inclut une phase de création de groupes qui peut être très coûteuse en temps si la décomposition en produit de facteurs premiers du nombre de feuilles de la topologie contient un nombre premier important. Pour rappel, en débutant l'algorithme à la profondeur maximum n de la topologie (au niveau des feuilles), soit p le nombre de feuilles de la topologie et k l'arité du niveau $n-1$, le nombre de groupes à générer par l'algorithme est $\binom{p}{k}$. Dans notre exemple, 55 se décompose en $5 * 11$.

Dans le pire des cas, il faudra générer $\binom{55}{11}$ groupes, c'est-à-dire près de 120 milliards de possibilités. Afin de passer outre cette limitation, nous utilisons la version améliorée de l'algorithme TreeMatch, appelée TreeMatchConstraints telle que présentée en sous-section 3.3.2. Ainsi, nous créons une topologie factice dont les caractéristiques sont les suivantes : le nombre de feuilles doit être supérieur ou égal au nombre de chares et le nombre de feuilles doit pouvoir être décomposé en produit de 2 et 3. Dans notre exemple, cette topologie aura 64 feuilles, décomposé en produit de facteur 2. À la première étape de création des groupes, l'algorithme devra ainsi générer au pire des cas $\binom{64}{2} = 2016$ groupes. Afin d'affecter par la suite les 55 chares sur les 64 feuilles de la topologie tout en gardant un bon équilibrage, nous appliquons des contraintes de façon régulière sur chaque plus petit sous-arbre. Ces contraintes sont fournies en entrée de TreeMatchConstraints. Une fois la permutation calculée, nous pouvons affecter les chares sur les différentes unités de calcul physiques en utilisant la même technique que pour TMLB_Min_Weight avec la fonction `AssignChareOnCore` : l'ordre des chares est conservé autant que possible pour maintenir les entités communiquant beaucoup proches les unes des autres et l'équilibrage est effectué sur les chares les moins chargés.

3.4.2.1 Améliorations en vue du passage à l'échelle

Placement en fonction de la topologie réseau

Les premiers résultats que nous avons obtenus avec cette méthode d'équilibrage de charge ont été plutôt concluants sur des cas de taille moyenne sur 64 cœurs. Cependant, plusieurs limitations sont apparues et nous ont amené à travailler sur des améliorations de l'algorithme afin de prévoir des expériences de plus grande envergure. Les principaux points bloquants ont tout d'abord concerné la prise en charge du réseau. Comme nous l'avons expliqué précédemment, dans le cas d'une topologie réseau arborescente, l'algorithme TreeMatch est capable de fournir un placement de groupes de chares intéressant. Cependant, force est de constater que ce type de topologie se raréfie dès lors que la taille des plate-forme augmente. Il est alors courant de rencontrer des réseaux organisés sous forme de maillage ou de tore 3d par exemple. TreeMatch n'étant pas conçu pour prendre en compte ces structures, nous avons dû proposer une solution alternative.

LibTopoMap [28] est une bibliothèque permettant d'assigner des processus MPI sur des grappes de nœuds de calcul en prenant en compte des topologies réseau quelconques. LibTopoMap utilise les mécanismes de topologie virtuelle internes au standard MPI pour prendre sa décision. Dans une version améliorée de TMLB_TreeBased nous avons implémenté l'utilisation de LibTopoMap afin d'effectuer la première étape de placement (groupes de chares) au niveau du réseau. Charm++ se basant sur MPI (avec AMPI) afin de traiter les communications entre chares, nous avons pu utiliser les fonctions natives du standard comme le montre l'algorithme distribué 5. La première étape de l'algorithme consiste donc à récupérer pour chaque processus MPI l'identifiant de sa machine hôte (ligne 2). Cette liste

affectée à *hostnames* comprend ainsi des doublons (plusieurs processus MPI sont présents sur chaque nœud). Or, LibTopoMap a besoin d'un processus MPI par nœud pour calculer le placement en fonction de la topologie réseau. L'algorithme élit donc un processus MPI maître pour chaque nœud. Cette étape correspond à la ligne 3 de l'algorithme. Les lignes 6 à 11 permettent à chaque processus MPI de vérifier s'il est le processus maître de son nœud. S'ensuit l'appel à la fonction standard `MPI_Comm_split` en ligne 13 qui va créer un nouveau communicateur MPI contenant uniquement les processus maîtres de chaque nœud. L'appel collectif à LibTopoMap (ligne 16) peut alors se faire et l'ensemble des chares de chaque nœud est alors réordonné en fonction du placement du processus MPI maître correspondant.

Algorithme 5: Appel de LibTopoMap

```
   Input: m La matrice de communication des nœuds
2  myHostname ←MPI_Get_processor_name();
3  hostnames[myRank] ← myHostname MPI_Allgather(hostnames, MPI_COMM_WORLD);
4  nodeMaster ← argfirst(hostnames, myHostname);
5
6  if myRank = nodeMaster then
7  |   color ← 1;
8  |   key ← myRank;
9  else
10 |   color ← 0;
11 |   key ← 0;
12
13 MPI_Comm_split(MPI_COMM_WORLD, color, key, newComm);
14
15 if newComm ≠ MPI_COMM_NULL then
16 |   libtopomap(m, newComm);
```

Tout comme TreeMatch, LibTopoMap a besoin en entrée du modèle d'affinité des processus (ici, le graphe pondéré de processus) et d'une représentation de la topologie du réseau. Cependant, bien que la récupération des informations d'architecture d'un nœud soit possible avec un outil comme hwloc, aucune solution générique n'existe à ce jour permettant de collecter les informations de topologie d'un réseau. Notons malgré tout à ce sujet les travaux préliminaires récents autour de netloc [24] [1], un projet similaire à hwloc mais dont l'objectif est la collecte des informations du réseau. Quoi qu'il en soit, nous nous sommes penchés sur l'élaboration d'une solution permettant de créer la structure du réseau dont LibTopoMap a besoin, en particulier pour les réseaux Cray Gemini [2] de type tore 3D, support d'une partie de nos expériences. Cette topologie relie entre eux des routeurs, deux nœuds de calcul étant connectés à chacun. Pour ce cas précis, nous avons procédé de la façon suivante :

1. Extraction, grâce aux outils de Cray, de la plus petite grille de routeurs en trois di-

1. http://www.open-mpi.org/projects/netloc/ .

mensions comprenant les nœuds de calcul attribués par le gestionnaire de ressources. Chaque routeur a une coordonnée en (x, y, z) dans le tore ;

2. Renumérotation des routeurs en fonction de leur position en (x, y, z) dans la grille afin de faciliter leur manipulation ;
3. Création d'une liste d'adjacence de tous les nœuds (routeurs et nœuds de calcul) de cette grille;
4. Suppression des nœuds ne faisant pas partie de la réservation ;
5. Sortie au format requis par LibTopoMap.

LibTopoMap se basant sur des informations quantitatives, nous avons défini la pondération des liens en fonction du critère suivant : des poids plus faibles sur les axes x et z et des poids plus forts sur l'axe y. La raison provient des caractéristiques des routeurs Cray Gemini connectant deux machines, comme illustré en figure 3.10. Dans chaque axe de ces routeurs, deux liens sont présents pour la communication inter-nœud sauf en y où l'un des deux liens est utilisé pour connecter les machines.

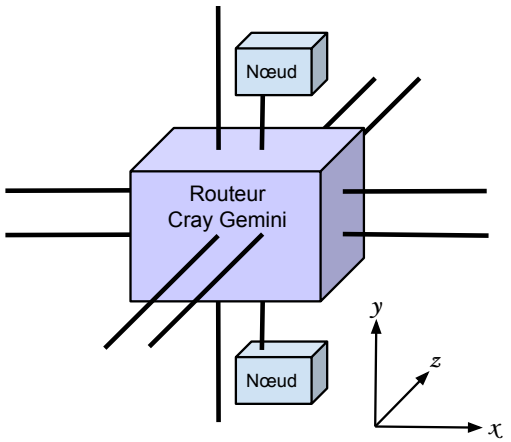


Figure 3.10 – Routeur d'un tore 3d de type Cray Gemini. Un des liens en axe y est réservé à la connexion d'un nœud de calcul.

La figure 3.11 montre un exemple de cette étape de création de la topologie pour LibTopoMap. On peut voir la grille en trois dimensions incluant les routeurs (en bleu) connectant les nœuds de calcul d'une réservation. Ces routeurs sont temporairement numérotés en suivant l'ordre (x, y, z). La liste d'adjacence correspondante est ensuite fournie à LibTopoMap pour le calcul du placement. Notons qu'il s'agit ici des routeurs du réseau Gemini. Il convient d'ajouter dans la liste d'adjacence les nœuds de calcul appropriés. Ce modèle est néanmoins perfectible. En effet, nous pouvons noter deux limitations importantes :

— certains liens ne sont pas pris en compte. Dans l'exemple de la figure 3.11, les nœuds 0 et 4 sont reliés via le nœud 2. Cependant, s'agissant d'un tore 3D, un autre chemin sur le même axe existe. Nous ne prenons pas en compte cette information ;
— les règles de routage au niveau des routeurs, gérées par un protocole proche de BGP, ont très certainement des spécificités qu'il serait intéressant de prendre en compte (axe favorisé, routes connues, etc).

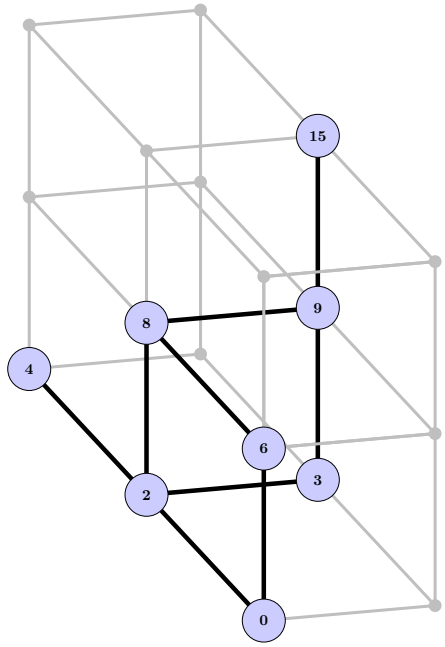


Figure 3.11 – Extraction des informations de topologie réseau d'une sous-partie d'un tore 3D. Chaque sommet correspond à un routeur (renuméroté) sur lequel deux nœuds de calcul sont connectés.

Parallélisation et distribution du placement intra-nœud

Durant la validation de notre modèle, nous avons détecté une autre limitation à l'algorithme TMLB_TreeBased. Celle-ci concerne le calcul du placement des chares à l'intérieur de chaque nœud. Dans la première version de l'algorithme, ce calcul se faisait de façon séquentielle pour chaque nœud. Ce temps de calcul, qui suit une courbe linéaire, devient vite substantiel dès lors que le nombre de ressources utilisées est grand. Pour pallier ces problèmes de passage à l'échelle, nous nous sommes penchés sur la parallélisation de cette section de l'algorithme (lignes 6 à 13 de l'algorithme 4. En particulier, nous avons apporté deux améliorations majeures. Dans un premier temps, à l'aide d'OpenMP, nous avons créé une section parallèle qui extrait le schéma de communication propre à chaque groupe de chares et qui calcule, pour chacun de ces groupes, un placement adéquat ré-

duisant les coûts de communication. Sur l'algorithme 4, cela correspond à ajouter une directive de type *pragma* pour la boucle débutant à la ligne 6. Cependant, là encore, en cas d'expériences d'envergure, le nombre de calculs de permutation exécutés en parallèle pose un réel problème. Une solution a donc été de conserver cette parallélisation d'une part et de distribuer l'appel à TreeMatchConstraints d'autre part (ligne 8) sur les différents nœuds. Pour ce faire, nous avons utilisé les possibilités offertes par Charm++. Une illustration de cette amélioration de l'algorithme d'équilibrage de charge TMLB_TreeBased est présentée en figure 3.12. Comme nous pouvons le voir sur ce schéma, un nœud de calcul maître extrait et distribue de façon parallèle pour chaque nœud (dont lui-même) la matrice de communication des chares qui y sont assignés (étape 1). Sur une unité de calcul de chacun de ces nœuds, le calcul de la permutation des chares est effectué comme détaillé précédemment. Il s'agit de la seconde étape sur la figure. Enfin, la troisième étape consiste à rapatrier les permutations calculées vers le nœud maître afin que l'algorithme général puisse se poursuivre. Les chares sont alors migrés en fonction de ces résultats et suivant la méthode décrite en algorithme 4.

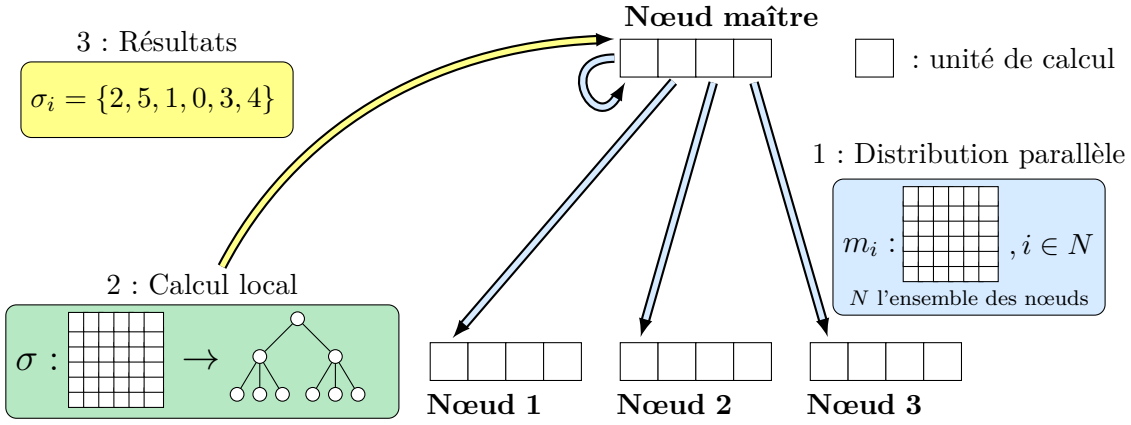


Figure 3.12 – Schéma de fonctionnement de la version parallèle et distribuée de TMLB_TreeBased. Les étapes sont numérotés de 1 à 3.

La figure 3.13 compare les temps d'exécution de l'algorithme TMLB_TreeBased dans sa version séquentielle et dans sa version parallèle distribuée en fonction du nombre de chares qu'il faut replacer. Dans un premier temps, nous pouvons voir que les sections d'initialisation (récupération du modèle de communication et de l'architecture) et la partie séquentielle de l'algorithme qui correspond à la première phase de placement sont logiquement stables. La seconde remarque est que le gain obtenu par la version distribuée face à la version séquentielle, bien qu'il ne soit pas proportionnel, est fixe, autour de 1,5 secondes. Ce résultat s'explique par le fait qu'en augmentant le nombre de chares, la taille des données à distribuer sur chaque nœud augmente de façon quadratique. Notons cependant qu'un cas

à 16384 chares répartis sur 64 cœurs (soit 2048 chares par nœud) est déjà un cas très peu réaliste.

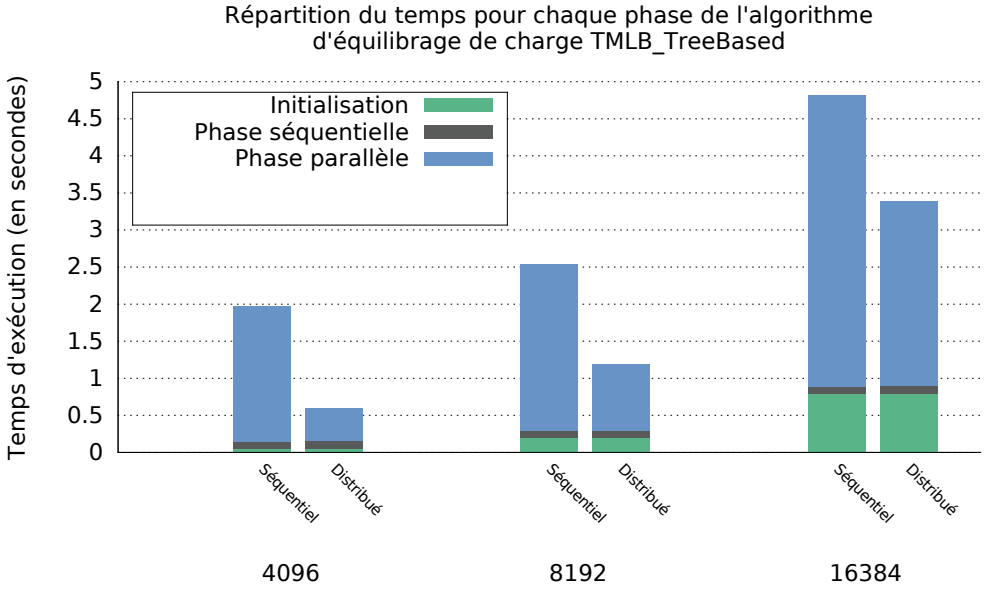


Figure 3.13 – Impact de la distribution des calculs de permutation sur la boucle parallèle de l'algorithme TMLB_TreeBased en fonction du nombre de chares total, répartis sur 64 cœurs (8 cœurs par nœud).

La figure 3.14 illustre la parallélisation avec OpenMP et la distribution grâce aux mécanismes de Charm++ de la seconde phase de l'algorithme TMLB_TreeBased. Sur ce schéma, le temps déterminé à partir du lancement de l'application est en abscisse tandis que les différentes unités de calcul utilisées sont en ordonnée. *(Maître)_0* correspond au cœur physique 0 du nœud 0. Les unités de calcul numérotées de 0 à 7 correspondent à un cœur libre (en général le cœur physique 0, sauf pour le nœud 0) sur chaque nœud de la réservation. Les différentes phases de l'algorithme représentées sur ce graphique sont les suivantes :

— *Initialisation* : Phase pendant laquelle les différents schémas de communication sont extraits. Cette phase est exécutée dans une section parallèle OpenMP, ce qui explique les chevauchements visibles sur la figure ;
— *Distribution* : Cette phase illustre le temps de transfert des données du nœud maître vers les autres nœuds en vue de procéder au calcul de la permutation des chares ;
— *Calcul placement* : C'est le calcul de la permutation. Ceci inclut la création de la fausse topologie et l'appel à TreeMatchConstraints;
— *Retour résultats* : Cette phase illustre l'envoi, pour chaque nœud, des résultats de

permutation obtenus vers le nœud maître ;

— *Traitement résultats* : Ici, les résultats sont traités au fur et à mesure de leur arrivée. Cette phase consiste à réassigner tous les chares en fonction de leur nouveau placement, tout en effectuant un nivellement de la charge sur les chares les moins chargés.

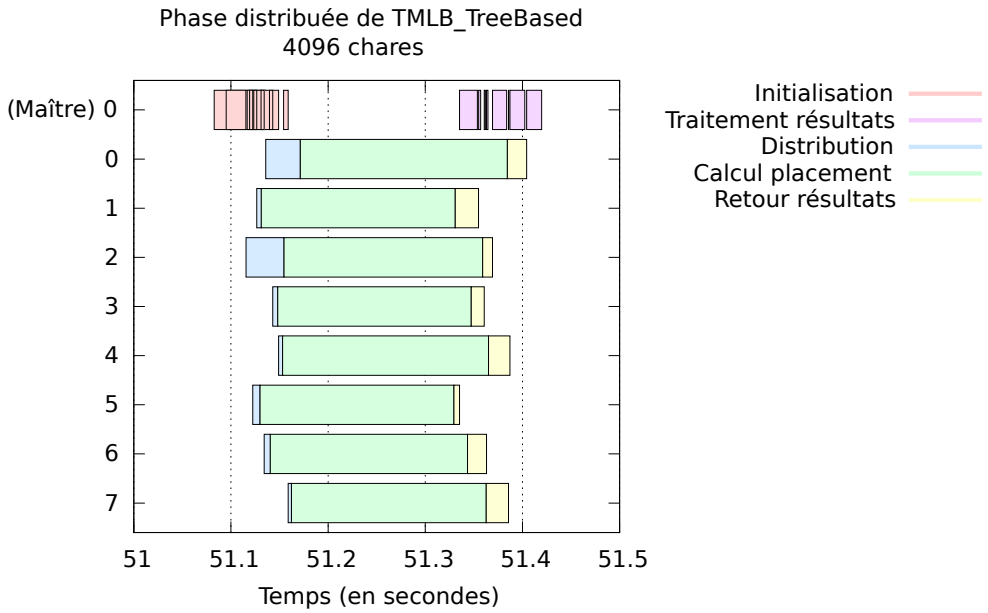


Figure 3.14 – Temps de calcul des permutations distribuées sur chaque nœud de calcul. Master_0 correspond à un cœur du nœud 0, les indices de 0 à 7 correspondent à un cœur disponible sur chaque nœud.

3.5 Expériences

Nous allons présenter dans cette section les résultats obtenus avec nos différentes versions de mécanismes d'équilibrage de charge. Nous avons sélectionné quatre applications pour nos tests. LeanMD [52] est une application de dynamique moléculaire connue pour avoir un déséquilibre de charge très important. Cette application a été la cible de notre algorithme TMLB_Min_Weight. Les trois autres applications, commBench (simulation de communications intensives irrégulières), kNeighbor (une application itérative où chaque chare communique avec k voisins) et Stencil3D (un stencil en trois dimensions), sont connues pour échanger beaucoup de données. Ces applications nous on servi à valider notre approche TMLB_TreeBased. Les expériences ont été conduites sur deux plate-formes : PlaFRIM et Blue Waters [53].

PlaFRIM

Les détails de cette plate-forme sont décrits en section 2.5.1.

Blue Waters

Blue Waters est un super calculateur installé à Urbana-Champaign dans l'Illinois dont la performance crête atteint 13,34 PetaFlops. Dans le cadre de notre collaboration au sein du Joint Laboratory for Petascale Computing, nous avons eu accès à cette plate-forme. Pour nos expériences, nous avons travaillé sur des nœuds XE6 (22640 disponibles) embarquant chacun deux processeurs AMD 6276 Interlagos. Chaque processeur se compose de seize cœurs. Les nœuds comptent 64 Go de mémoire RAM. Le réseau reliant ces différentes machines est un tore 3D de type Cray Gemini. Le système d'exploitation est une distribution SuSE proposant un noyau Linux en version 2.6.32.

Toutes nos expériences ont utilisé une métrique de nombre de messages envoyés comme base du modèle d'affinité des chares. Nous avons comparé nos solutions à une exécution sans aucune répartition de charge (DummyLB ou Baseline) et à plusieurs solutions d'équilibrage de charge disponibles nativement avec Charm++. Ces solutions proposent des stratégies gloutonnes et d'autres plus sophistiquées. En particulier, GreedyLB (resp. GreedyCommLB) se base sur la charge CPU (resp. la charge CPU et la communication) pour assigner les chares sur les unités de calcul en suivant la stratégie suivante : le chare ayant la charge la plus importante est affecté au cœur ayant la charge la plus faible. RefineCommLB déplace les chares des cœurs surchargés vers les cœurs les moins chargés de façon à atteindre une charge moyenne. RefineCommLB est une des stratégie suggérées pour l'application kNeighbor.

3.5.1 LeanMD

LeanMD [52] est une application de dynamique moléculaire écrite en Charm++, continuité de l'application NAMD dont elle est une évolution prévue pour un meilleur passage à l'échelle. Dans cette application, les atomes sont divisés en cellules contenant un certain nombre de particules et chaque cellule va calculer ses interactions locales avec ses voisins les plus proches. LeanMD est connue pour être une application dont la répartition de charge initiale est très déséquilibrée. Parmi les paramètres que l'on peut définir en entrée de cette application, le nombre de particules par cellule (par chare dans le contexte logiciel) peut être modifié afin de générer davantage de calcul et de communications. Cependant, le volume de communication est extrêmement faible comparé au volume de calcul. De ce fait, la consommation CPU devient un critère prioritaire lors de l'équilibrage de charge. Notre algorithme TMLB_Min_Weight répond à ce type de problématique en favorisant le nivellement de la charge CPU tout en réduisant autant que possible les coûts de communication. Les résultats que nous avons obtenus avec cette solution sont présentés en

figure 3.15. Sur cette figure, nous avons tracé le temps d'exécution total (incluant la répartition de charge) en fonction de la taille du problème, c'est-à-dire du nombre de particules par cellule. Nous pouvons voir qu'excepté pour les petits cas, notre algorithme donne de meilleures performances que les algorithmes natifs. Sur les problèmes supérieurs ou égaux à 2000 particules par cellule, nous réduisons le temps d'exécution de près de 30% par rapport à l'exécution sans aucun équilibrage de charge.

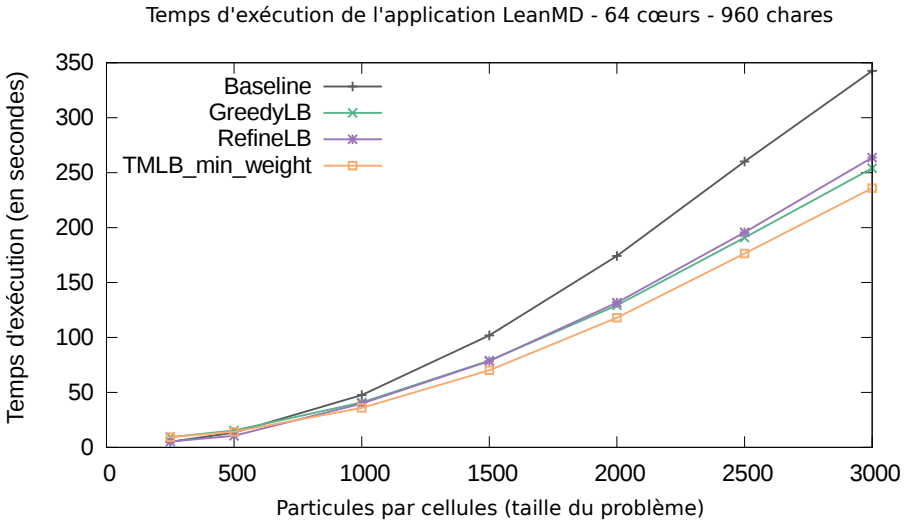


Figure 3.15 – Temps d'exécution (incluant l'équilibrage de charge) en fonction du nombre de la taille du problème (nombre de particules par cellules) de l'application LeanMD sur 64 cœurs de la plate-forme PlaFRIM.

Si nous comparons le temps du calcul de l'équilibrage de charge pour chaque stratégie, nous pouvons noter que TMLB_Min_Weight prend en moyenne 233 ms tandis que RefineLB prend 1 ms. GreedyLB nécessite 14,5 ms afin de déterminer la répartition de charge. Cependant, bien que TMLB_Min_Weight soit la solution la plus lente, ce temps de calcul est largement contrebalancé par les bénéfices obtenus sur le temps d'exécution total de l'application.

La figure 3.16 présente le nombre de chares migrant d'une unité de calcul à une autre pour la même série d'expériences illustrée en figure 3.15. Nous pouvons voir dans un premier temps que la stratégie GreedyLB ne tient aucun compte de cet aspect puisque la quasi totalité des chares est déplacée. RefineLB est principalement une stratégie incrémentale qui équilibre la charge en ne déplaçant que quelques chares des unités de calcul les plus chargées vers les moins chargées. Ainsi, le nombre de migrations est très faible. Notre stratégie TMLB_Min_Weight tente de trouver un bon compromis entre le coût de migration et les autres critères impactant le temps d'exécution (charge CPU, communication, topologie).

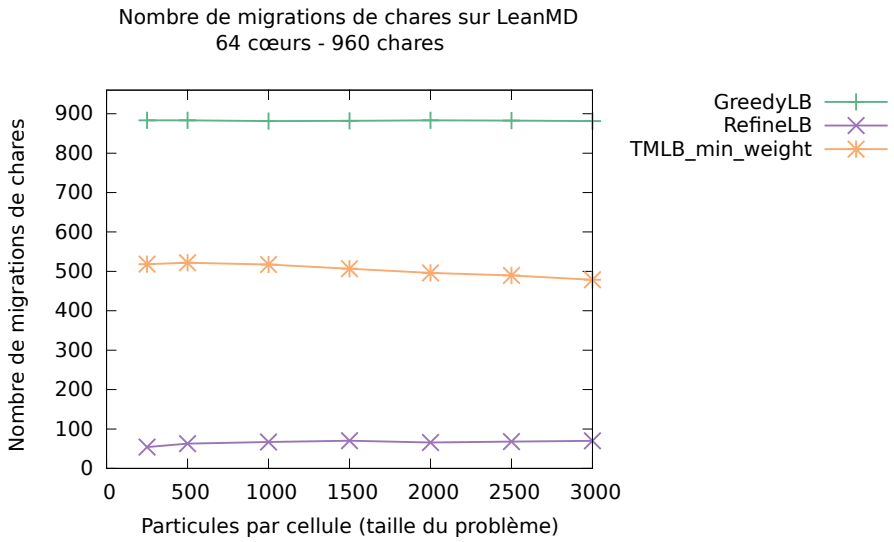


Figure 3.16 – Nombre moyen de chares migrés sur un total de 960 chares pour chaque algorithme d'équilibrage de charge et pour chaque jeu d'expérience de LeanMD.

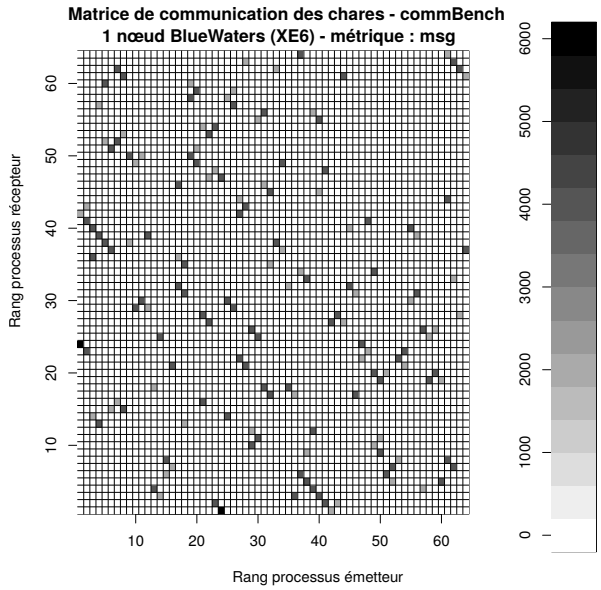
Sans minimisation des migrations, le volume des migrations est plus important (en moyenne 682 migrations) et le temps d'exécution est majoré de près de 5%. Ainsi, cette section de l'algorithme apparaît nécessaire pour obtenir de bonnes performances.

3.5.2 commBench

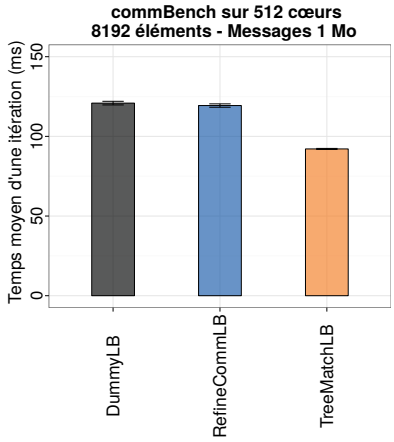
CommBench est une application d'évaluation des performances que nous avons écrite en nous basant sur l'application kNeighbor proposée par Charm++. Cette application simule des communications fortement irrégulières entre plusieurs chares. La matrice présentée en figure 3.17(a) illustre un schéma de communication des chares de l'application commBench, basé sur le nombre de messages envoyés et réalisé sur un nœud de la plate-forme Blue Waters.

La figure 3.17(b) présente des résultats obtenus sur cette application en utilisant notre algorithme d'équilibrage de charge TMLB_TreeBased. Les mesures présentées sont des moyennes sur 10 exécutions. Ces expériences ont été réalisées sur la plate-forme Blue Waters sur un total de 512 cœurs répartis sur 16 nœuds. Nous avons découpé notre application en 8192 chares et les messages envoyés ont été définis à une taille de 1 Mo. Les gains offerts par notre solution sont substantiels. Le gain en temps d'exécution par rapport à DummyLB (aucune répartition de charge) et à RefineCommLB sont de l'ordre de 25%.

Enfin, nous avons réalisé le même type d'expériences sur un nombre plus important



(a) Matrice de communication des chares de l'application commBench (64 chares).



(b) Résultats obtenus sur la plate-forme Blue Waters de l'application commBench en fonction des algorithmes d'équilibrage de charge.

de nœuds de calcul de la plate-forme Blue Waters afin d'estimer le passage à l'échelle de notre algorithme compte tenu des améliorations que nous lui avons apportées. Les résultats sont décrits en figure 3.17. Notons que les algorithmes d'équilibrage de charge natifs de Charm++ (GreedyLB, GreedyCommLB ou RefineCommLB) ne passent pas à l'échelle, ce qui explique leur absence ici. Nous nous sommes donc exclusivement comparés à une exécution sans aucune répartition de charge. D'une part les temps d'exécutions sont très intéressants dans le cas où TMLB_TreeBased est utilisé (gains de près de 30% par rapport aux temps obtenus sans répartition de charge) et d'autre part, des mesures ont pu être réalisées sur des cas très importants. En effet, notre algorithme a été capable de calculer une solution de placement profitable pour 65536 chares répartis sur 8192 cœurs de calcul.

3.5.3 kNeighbor

kNeighbor est une application fournie dans la suite logicielle de Charm++. Il s'agit d'une application de simulation intensive de communications suivant le schéma suivant : chaque chare communique un certain volume de données avec ses k voisins les plus proches et effectue un certain volume de calcul. Par défaut, $k = 7$. Contrairement à commBench,

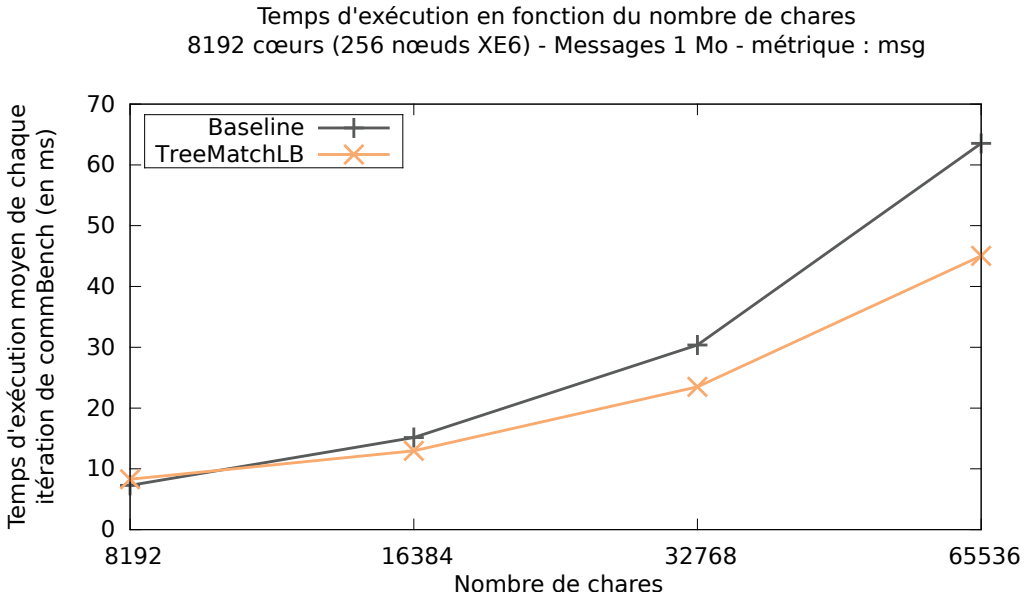
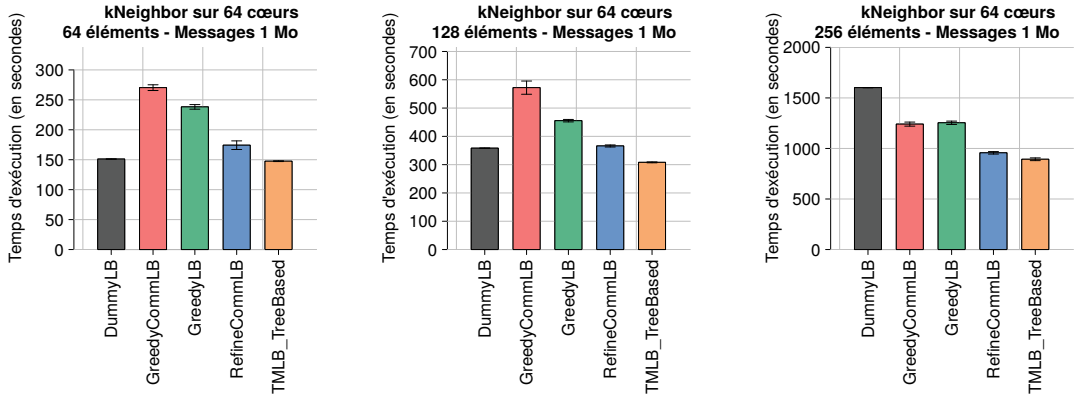


Figure 3.17 – Temps d'exécution moyen d'une itération obtenus sur la plate-forme Blue Waters sur 8192 cœurs (256 nœuds XE6) en fonction du nombre de chares.

les communications sont très régulières et les gains que l'on peut espérer avec du placement efficace de chares sont assez faibles. Cependant, comme nous l'avons évoqué dans le chapitre 1, bien que le schéma de communication soit régulier, la manière dont sont placés les objets sur l'architecture physique impose de connaître de façon très fine les caractéristiques du matériel. En effet, la numérotation physique des unités de calcul peut fortement varier en fonction de l'architecture ce qui complique la maîtrise que peut avoir l'utilisateur sur le placement initial des éléments de son application (que nous parlions de chare ou de processus). Notre algorithme TMLB_TreeBased se propose de prendre à sa charge cette difficulté.

3.5.3.1 Impact de la répartition de charge considérant un placement initial non optimal

Les résultats présentés sur les figures 3.18(a), 3.18(b) et 3.18(c) sont issues d'expériences réalisées sur la plate-forme PlaFRIM. L'architecture des nœuds propose une numérotation physique des cœurs ne suivant pas une numérotation linéaire. Plus concrètement, en parcourant l'arbre de la topologie par un parcours en profondeur, la numérotation physique des unités de calcul est 0, 2, 4, 6, 1, 3, 5, 7. Pour cette série d'expériences, nous avons considéré cette information comme inconnue de l'utilisateur et avons donc utilisé le placement des chares par défaut, c'est-à-dire que les chares ont été assignés dans l'ordre sur les



(a) (b) (c)

Figure 3.18 – Temps d'exécution de l'application kNeighbor sur la plate-forme PlaFRIM en fonction du nombre de chares et de plusieurs algorithmes de répartition de charge.

cœurs physiques 0, 1, 2, 3, 4, 5, 6 et 7. Cette situation initiale crée de fait de l'irrégularité dans les communications puisque les chares communiquant beaucoup se trouvent éloignés.

Les figures 3.18(a), 3.18(b) et 3.18(c) illustrent respectivement les moyennes de temps d'exécution sur 64 cœurs (8 nœuds) pour 64, 128 et 256 chares. La taille des messages échangés a été définie à 1 Mo. Dans un premier temps on peut voir que dans tous les cas de figure, TMLB_TreeBased offre les meilleures performances. Le faible écart type, illustré par les barres d'erreur, montre également une bonne stabilité des résultats. La seconde remarque que nous pouvons faire concerne les très mauvais résultats des autres méthodes d'équilibrage de charge pour les cas où le nombre de chares est faible. En effet, dans ces conditions, l'équilibrage de charge est inexistant voire négligeable et seule l'optimisation des coûts de communication peut avoir un réel impact. À partir de 256 chares, ces méthodes obtiennent de bien meilleures performances et offrent un gain conséquent par rapport à une exécution classique. Enfin, notons que les performances relatives acquises avec TMLB_TreeBased augmentent avec la taille du problème (nombre de chares).

3.5.3.2 Temps d'exécution de l'algorithme d'équilibrage de charge

Le temps d'exécution de l'algorithme d'équilibrage de charge TMLB_TreeBased, comparé aux méthodes natives de Charm++, est présenté en figure 3.19. Notons tout d'abord qu'excepté pour RefineCommLB, les différentes stratégies semblent suivre une courbe linéaire tandis que le nombre de chares est doublé à chaque graduation de l'axe des abscisses. On peut noter également qu'au delà de 4096 chares, RefineCommLB n'est plus capable de fournir un placement en un temps raisonnable. Enfin, TMLB_TreeBased est

clairement plus lent que les autres stratégies pour les cas inférieurs à 8192 et reste largement plus lent que les deux stratégies gloutonnes à 8192 chares. Cependant, le placement déterminé est bien meilleur et ce temps de calcul est très nettement compensé par les gains de performances sur le temps d'exécution total de l'application.

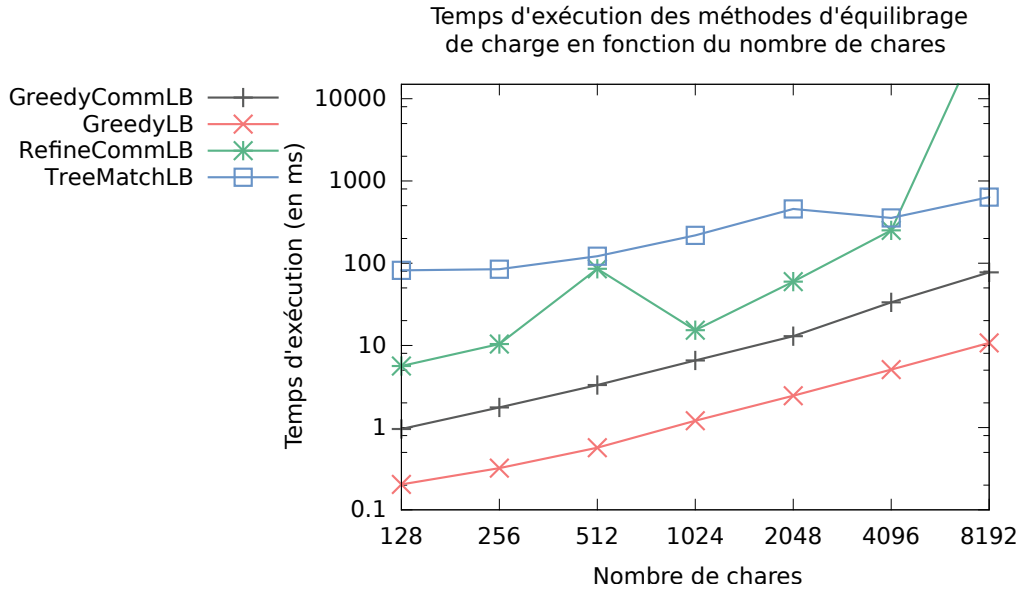
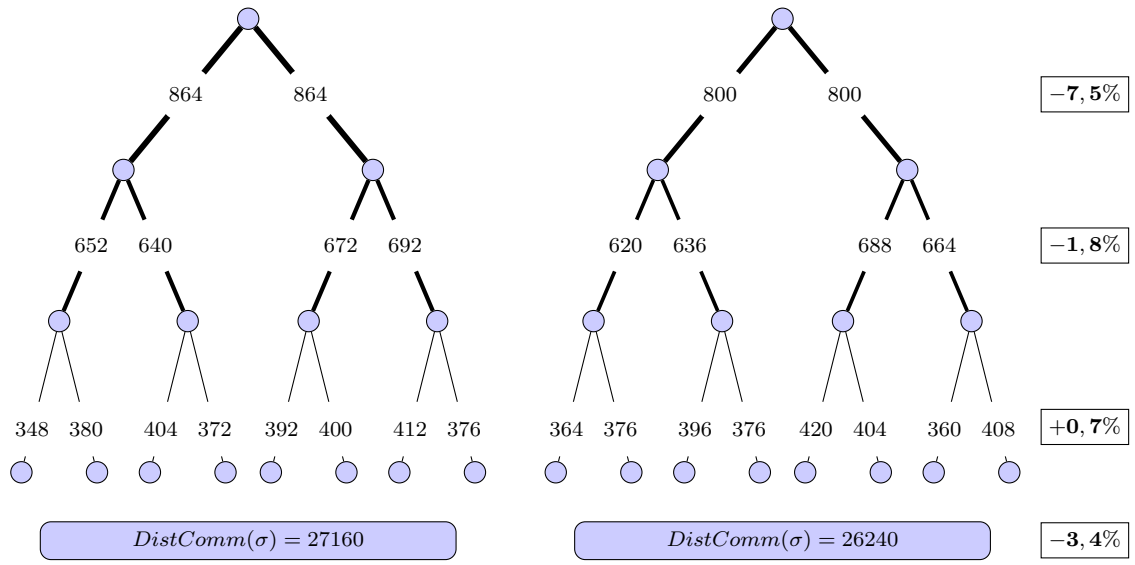


Figure 3.19 – Comparaison du temps d'exécution de la méthode d'équilibrage de charge entre TMLB_TreeBased et d'autres stratégies.

3.5.3.3 Répartition des communications sur les liens

Afin de comprendre l'impact du placement de chares sur les communications, nous avons extrait le nombre de messages échangés durant 10 itérations avant et après avoir effectué un équilibrage de charge avec TMLB_TreeBased sur chaque lien de la topologie d'un nœud. Les figures 3.20(a) et 3.20(b) montrent ces mesures. Le premier arbre correspond aux mesures avant l'étape d'équilibrage de charge tandis que le second correspond à celles effectuées après l'appel à TMLB_TreeBased. Nous pouvons tout d'abord constater que la somme des messages sur les liens (en milliers de messages envoyés) est plus faible après la phase d'équilibrage. En effet, la somme des messages sur les différentes arêtes de la topologie dans le premier cas est de 7468000 messages alors qu'elle est de 7312000 dans le deuxième cas. Cette baisse d'environ 2% est en réalité déplacée dans les communications intra-cœurs. Une autre remarque concerne le volume de messages échangés sur les liens les plus hauts dans l'arbre, c'est-à-dire les plus coûteux. Chaque arbre modélisant un nœud de calcul, la racine correspond donc à la mémoire RAM de ce nœud. On constate que le nombre



(a) Sans intervention.

(b) Après un appel à TMLB_TreeBased.

Figure 3.20 – Distribution des communications sur les liens de la topologie pour 10 itérations de l'application kNeighbor (en milliers de messages).

de messages transitant via cette mémoire (plus lente que les caches partagés) diminue de près de 7,5% après l'appel à TMLB_TreeBased. À mesure que l'on descend vers les feuilles, la différence des sommes des communications des arêtes d'un niveau avant et après équilibrage de charge diminue et s'inverse au niveau le plus bas. Ce comportement illustre bien l'impact d'un placement efficace qui répercute autant que possible les communications sur les liens les plus bas de la topologie. Enfin, nous avons calculé pour chaque cas la valeur de la fonction objectif *DistComm* (voir chapitre 1). Cette valeur a bien été minimisée grâce à notre algorithme de placement des chares.

3.5.3.4 Comportement face aux politiques de placement initiales

Bien que nous ayons obtenu de bons résultats sur l'application kNeighbor dans le cas où le placement initial des chares n'est pas optimal, nous avons souhaité évaluer le comportement de notre algorithme lorsque l'utilisateur a connaissance des spécificités de numérotation physique des unités de calcul de l'architecture. Nous avons donc réalisé une série d'expériences sur un nœud PlaFRIM en faisant varier les politiques de placement initial des chares. Nous avons tracé trois courbes sur la figure 3.21 : la première, "DummyLB 0-7" se base sur un placement par défaut des chares, c'est-à-dire, compte tenu de la numérotation physique du nœud, sur un placement non linéaire. La seconde courbe montre les résultats

obtenus avec TMLB_TreeBased, quel que soit le placement initial. Enfin, la dernière courbe affecte bien les chares de façon optimale pour kNeighbor en prenant en compte les caractéristiques de l'architecture. On constate l'influence du placement initial sur cette application en comparant les deux courbes de l'exécution sans équilibrage de charge. Une autre remarque est que, quel que soit le placement initial, TMLB_TreeBased obtient dans tous les cas au pire les mêmes performances que le placement optimal. D'une manière générale, en utilisant cet algorithme, l'utilisateur n'a pas à se soucier des détails de l'architecture. Cet aspect est primordial puisqu'il assure une bonne portabilité des performances.

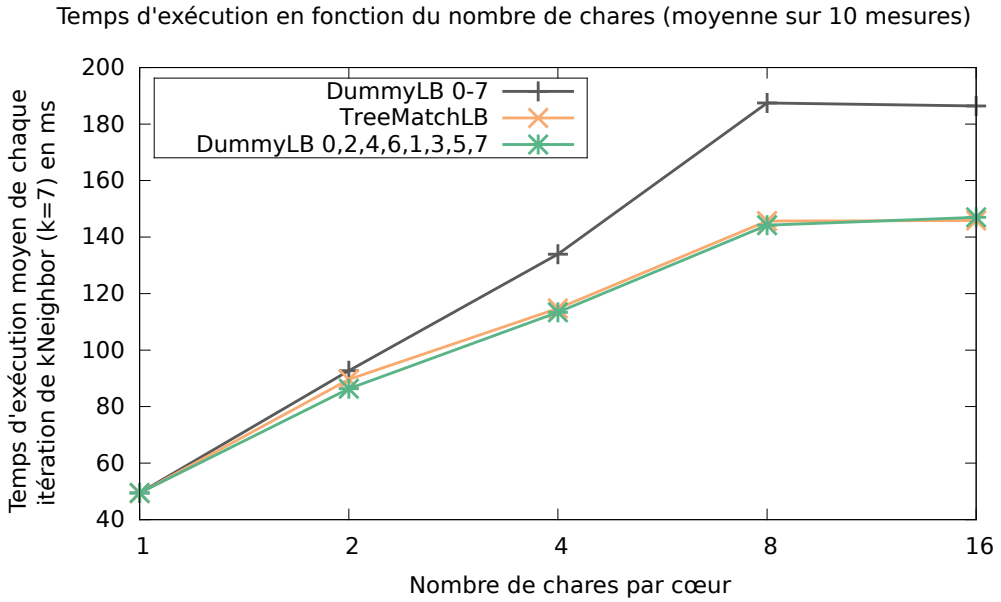


Figure 3.21 – Évaluation des gains obtenus avec TMLB_TreeBased sur kNeighbor en fonction des politiques de placement initiales.

3.5.4 Stencil3D

Stencil3D est une application modélisant un *stencil* en trois dimensions et générant des communications régulières avec des voisins fixes. Compte tenu de ces critères, une phase d'équilibrage de charge toutes les dix itérations n'est pas nécessaire. Ainsi, au bout des dix premières itérations, nous appelons l'algorithme d'équilibrage puis l'exécution de l'application se poursuit. Les meilleurs résultats obtenus, quelles que soient les stratégies utilisées, l'ont été en suivant ce schéma. Les expériences réalisées sont présentées en figure 3.22. Tout comme kNeighbor, Stencil3D est une application fortement communicante. C'est pourquoi nous avons procédé aux tests avec l'algorithme TMLB_TreeBased. Nous pouvons noter deux résultats importants. Premièrement, nous obtenons un gain de près de 18% comparé

aux autres stratégies. Deuxièmement, notre algorithme offre une fois encore une très bonne stabilité, comme le montre l'écart type.

Le temps d'équilibrage de charge est très court dans le cas des stratégies natives de Charm++ pour ces expériences sur Stencil3D. Excepté pour RefineCommLB qui prend environ 7,5 ms, les autres méthodes gloutonnes nécessitent moins de 1 ms pour déterminer la nouvelle affectation des chares. À l'inverse, TMLB_TreeBased prend environ 214,8 ms dans sa version séquentielle (avant les améliorations décrites en 3.4.2.1). Comme pour kNeighbor, ce temps d'équilibrage de charge doit être mis en perspective avec le temps d'exécution total de l'application. Le temps nécessaire à trouver un bon placement est complètement balancé par le gain que nous en retirons sur le temps d'exécution.

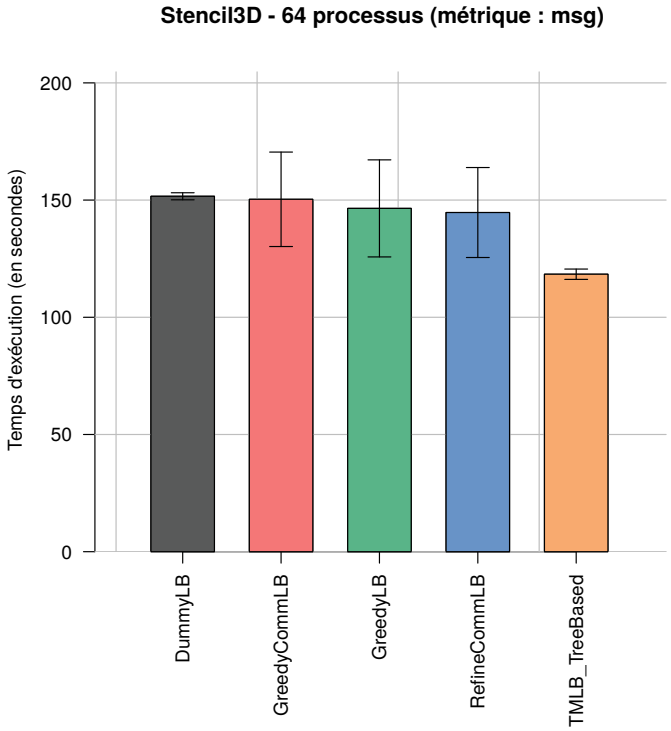


Figure 3.22 – Temps d'exécution de l'application Stencil3D sur la plate-forme PlaFRIM en fonction des méthodes d'équilibrage de charge.

3.6 Conclusion partielle

D'un côté, Charm++ propose un mécanisme d'équilibrage de charge extensible. De l'autre, les architectures multi-cœurs font croître le volume des communications dans les applications parallèles. Partant de ce constat, nous avons proposé deux algorithmes d'équilibrage de charge qui tiennent compte de la topologie et de l'affinité des entités logicielles. Pour cela, nous nous sommes basés sur l'algorithme TreeMatch, détaillé au chapitre précédent. En particulier, nous avons développé l'algorithme TMLB_Min_Weight conçu pour les applications à faible volume de communications. Cet algorithme a montré de bons gains de performance sur une application de dynamique moléculaire en nivelant la charge, en réduisant les coûts de communication et en minimisant les migrations de chares. Le second algorithme, TMLB_TreeBased, a été conçu pour les applications à fort volume de communications. Cette solution a été l'objet d'un certain nombre d'optimisations afin d'en favoriser le passage à l'échelle. Ces optimisations ont permis d'en faire un algorithme d'équilibrage de charge hiérarchique distribué qui a été validé sur une application exécutée sur 8192 cœurs. Là encore, sur diverses applications, notre méthode a montré des résultats satisfaisants dans l'optimisation de l'exécution. Notons également qu'un travail important a été réalisé afin d'abstraire la numérotation des unités de calcul des architectures.

Néanmoins, ces deux méthodes sont encore perfectibles. TMLB_Min_Weight n'a pas profité des mêmes améliorations que TMLB_TreeBased, et il serait intéressant d'en proposer une version distribuée qui puisse être validée sur des exécutions à plus grande échelle. Évaluer l'impact de la minimisation des migrations avec l'algorithme d'équilibrage TMLB_TreeBased fait également partie des objectifs. Toujours sur cette seconde stratégie, une évaluation du coût de calcul de la permutation à différents niveaux de la topologie pourrait permettre une décision automatique sur la granularité des deux étapes de placement. Enfin, deux perspectives à court terme concernent des expériences à plus grande échelle et des évaluations de TMLB_TreeBased sur de nouvelles applications. En particulier, ChaNGa [38] et OpenAtom [62] sont visées.

Conclusion

Cette thèse a été réalisée à l'Université de Bordeaux, au sein de l'équipe Runtime d'Inria, membre de l'équipe SATANAS du Laboratoire Bordelais de Recherche en Informatique (LaBRI). Elle a porté sur le placement statique et dynamique d'entités logicielles en fonction de la topologie.

3.7 Bilan

Dans ce manuscrit, nous avons dans un premier temps posé les bases du placement de processus. En particulier, nous avons estimé l'impact de la localité (et de l'échange) des données : plus les données sont proches physiquement de l'entité logicielle qui veut y accéder dans la topologie matérielle, plus le temps d'accès sera court. C'est cette localité que se propose de prendre en compte notre algorithme de placement. Pour ce faire, il est donc nécessaire de connaître les spécificités de l'architecture cible et l'affinité des entités logicielles. Plusieurs problèmes se posent alors. Les architectures matérielles, d'une part, sont très diverses et leur documentation est souvent lacunaire. Il convient de fournir une approche générique capable de gérer les particularités des plate-formes. D'autre part, l'affinité doit être définie selon une métrique. Quelle métrique choisir ou comment la calculer sont autant de questions auxquelles nous avons proposé des solutions.

L'algorithme TreeMatch est une solution de calcul de placement indépendante de l'architecture et de tout paradigme. TreeMatch est capable de prendre en compte l'affinité des entités de traitement logicielles d'une application et la topologie matérielle pour fournir une permutation de ces entités en vue de réduire les coûts de communication. Nous l'avons tout d'abord présenté sous l'angle du placement statique. Ce type de placement implique que les entités logicielles sont assignées sur les unités de calcul au lancement de l'application et ne migrent plus durant l'exécution. Les placements calculés par TreeMatch ont donné de très bons résultats sur les différentes applications MPI testées (ZeusMP/2 et certains noyaux de calcul des *NAS Parallel Benchmarks*). Parmi les métriques que nous avons déterminées, le nombre de messages échangés a permis d'obtenir des gains plus importants

que les métriques *size* et *avg*. Nous avons comparé les résultats obtenus à d'autres méthodes de placement : sur les nombreuses expériences réalisées, notre algorithme se montre majoritairement meilleur face aux autres techniques. Enfin, et c'est une donnée importante pour d'autres utilisations de TreeMatch, l'algorithme trouve un placement idoine en un temps raisonnable. De plus, son passage à l'échelle a été démontré.

L'algorithme de placement TreeMatch propose une approche générique qui ne dépend d'aucun paradigme de programmation. Ainsi, il peut être utilisé au sein de plusieurs modèles de programmation et notamment dans Charm++ pour effectuer du réordonnancement dynamique dont les objectifs sont multiples : équilibrer la charge CPU et réduire les coûts de communication. Ici, les entités logicielles appelées *chares* sont migrées, parfois à plusieurs reprises, en cours d'exécution. Pour réaliser cet équilibrage de charge prenant en compte la topologie et l'affinité, nous avons proposé deux algorithmes. TMLB_Min_Weight cible les applications générant beaucoup de calcul mais assez peu de communications. Testé sur une application de dynamique moléculaire, cet algorithme a permis de réduire le temps d'exécution de près de 30%. TMLB_TreeBased est un algorithme d'équilibrage de charge hiérarchique distribué. Hiérarchique car il intervient à plusieurs niveaux de la topologie matérielle et distribué puisqu'il se base sur OpenMP et des mécanismes de Charm++ pour que chaque nœud de la plate-forme calcule le placement de ses propres entités. Ces deux caractéristiques lui confèrent d'ailleurs une très bonne capacité de passage à l'échelle. TMLB_TreeBased a été évalué sur plusieurs applications et a donné à chaque fois des résultats satisfaisants. Enfin, notons que ces travaux ont été réalisés dans le cadre du laboratoire commun Inria-UIUC. Plusieurs séjours sur place nous ont permis de travailler avec les développeurs de Charm++, de façon à accroître l'efficacité de nos algorithmes. Une partie des résultats publiés dans ce manuscrit à ce sujet a débouché sur une publication commune dans une conférence internationale.

3.8 Perspectives

Le placement d'entités logicielles est un problème très discuté depuis plusieurs années. Cependant, force est de constater qu'il reste difficile de bien comprendre tous les mécanismes qui entrent en jeu. Un objectif à court terme pour nos travaux consiste à étudier le plus finement possible l'impact du placement. Des travaux en ce sens ont déjà débuté et ont donné des pistes intéressantes, en particulier au niveau du fonctionnement des caches. Sur les données nécessaire au calcul du placement, il serait intéressant de trouver des solutions alternatives à l'exécution préliminaire permettant de dégager le modèle d'affinité. Plusieurs pistes ont été évoquées, comme la simulation des applications, l'analyse statique du code ou l'exécution d'un squelette ôté de la majorité des sections générant du calcul. La métrique utilisée est aussi un sujet de travail important. Nos travaux se sont concentrés sur des métriques intuitives (volume de données, nombre de messages, etc) pour lesquelles aucune

notion de temporalité n'intervient. Des approches plus complexes, prenant par exemple en considération la fréquence d'envoi de messages, pourraient être évaluées.

L'algorithme TreeMatch en lui-même pourrait faire également l'objet d'améliorations. La prise en charge du réseau par exemple est un réel défi. Intégrer LibTopoMap dans notre implémentation est une solution envisagée. Nous réfléchissons aussi à développer notre propre algorithme de placement de façon à ne pas dépendre de MPI avec LibTopoMap et à pouvoir s'interfacer avec un outil comme netloc. Cette solution pourrait par exemple se baser sur Scotch pour le partitionnement de graphes.

Enfin, l'intégration de TreeMatch dans Charm++ ouvre des perspectives prometteuses. Dans Open MPI tout d'abord, puisque TreeMatch est en passe d'être intégré dans cette implémentation pour fournir un ensemble de fonctions de gestion de topologies virtuelles prenant en compte la topologie matérielle (fonction `MPI_Dist_graph_create` du standard MPI). Toutefois, TreeMatch pourrait aussi être utilisé dans cette implémentation dans le but d'optimiser les fonctions collectives ou pour diminuer les coûts de communication dans les protocoles de tolérance aux fautes. Enfin, nous avons utilisé l'algorithme TreeMatch dans cette thèse pour améliorer l'utilisation des ressources. Une étape supplémentaire pour influer sur l'exécution des applications serait de travailler sur l'allocation des ressources. C'est l'objectif de la thèse d'Adèle Villiermet qui vient de débuter dans notre équipe. Ses travaux portent sur la prise en compte des contraintes d'affinité et d'architecture dans l'allocation des ressources grâce à TreeMatch.

Bibliographie

[1] 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 (cf. p. 23, 26).

[2] Robert Alverson, Duncan Roweth et Larry Kaplan. “The Gemini System Interconnect”. In : *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*. HOTI ’10. Washington, DC, USA : IEEE Computer Society, 2010, p. 83–87 (cf. p. 74).

[3] 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 (cf. p. 22).

[4] D. H. Bailey et al. *NAS Parallel Benchmark Results*. Rapp. tech. 94-006. RNR, 1994 (cf. p. 42).

[5] P. 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 (cf. p. 22, 26).

[6] Abhinav Bhatele. “Topology Aware Task Mapping”. In : *Encyclopedia of Parallel Computing (to appear)*. Sous la dir. de D. Padua. Springer Verlag, 2011 (cf. p. 61).

[7] Abhinav Bhatelé et Laxmikant V. Kalé. “Benefits of Topology Aware Mapping for Mesh Interconnects”. In : *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)* 18.4 (2008), p. 549–566 (cf. p. 61).

[8] Robert D. Blumofe et al. “Cilk : an efficient multithreaded runtime system”. In : *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPOPP ’95. Santa Barbara, California, USA : ACM, 1995, p. 207–216 (cf. p. 60).

[9] B. Brandfass, T. Alrutz et T. Gerhold. “Rank Reordering for MPI Communication Optimization”. In : *Computer & Fluids* (jan. 2012) (cf. p. 23, 26, 60).

[10] F. 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, fév. 2010 (cf. p. 27).

[11] Robert K. Brunner et Laxmikant V. Kalé. “Handling Application-Induced Load Imbalance using Parallel Objects”. In : *Parallel and Distributed Computing for Symbolic and Irregular Applications*. World Scientific Publishing, 2000, p. 167–181 (cf. p. 61).

[12] Henri Casanova et al. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In : *Journal of Parallel and Distributed Computing* 74.10 (juin 2014), p. 2899–2917 (cf. p. 57).

[13] H. Chen et al. “MPIPP : an Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters”. In : *ICS*. 2006, p. 353–360 (cf. p. 23, 26, 44, 60).

[14] 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 (cf. p. 60).

[15] D. Buntinas, G. Mercier and W. Gropp. “Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem”. In : *Parallel Computing, Selected Papers from EuroPVM/MPI 2006* 33.9 (sept. 2007), p. 634–644 (cf. p. 31).

[16] D. Solt. *A Profile Based Approach for Topology Aware MPI Rank Placement*. http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt. 2007 (cf. p. 24).

[17] M. Deveci et al. “Exploiting Geometric Partitioning in Task Mapping for Parallel Computers”. In : *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. Mai 2014, p. 27–36 (cf. p. 24).

[18] Karen Devine et al. “Zoltan Data Management Services for Parallel Dynamic Applications”. In : *Computing in Science and Engineering* 4.2 (2002), p. 90–97 (cf. p. 24, 33).

[19] E. Duesterwald, R. W. Wisniewski, P. F. Sweeney, G. Cascaval and S. E. Smith. *Method and System for Optimizing Communication in MPI Programs for an Execution Environment*. http://www.faqs.org/patents/app/20080288957. 2008 (cf. p. 24).

[20] F. Pellegrini. Scotch *and* LibScotch 5.1 *User’s Guide*. http://www.labri.fr/perso/pelegrin/scotch/. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800. Août 2008 (cf. p. 66).

[21] F. Pellegrini. “Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs”. In : *Proceedings of SHPCC'94, Knoxville*. IEEE. Mai 1994, p. 486–493 (cf. p. 14, 23, 24, 26, 44).

[22] E. Gabriel et al. “Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation”. In : *Proceedings of the 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, sept. 2004, p. 97–104 (cf. p. 24).

[23] Michael R. Garey et David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990 (cf. p. 14).

[24] Brice Goglin, Joshua Hursey et Jeffrey M. Squyres. “netloc : Towards a Comprehensive View of the HPC System Topology”. In : *Proceedings of the fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014), held in conjunction with ICPP-2014*. Minneapolis, MN, sept. 2014 (cf. p. 28, 74).

[25] 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. 10.1007/BFb0056575. Springer Berlin / Heidelberg, 1998, p. 188–195 (cf. p. 22, 26).

[26] B. Hendrickson et R. Leland. *The Chaco User's Guide : Version 2.0*. Rapp. tech. SAND94–2692. Sandia National Laboratory, 1994 (cf. p. 23, 26, 44).

[27] Bruce Hendrickson et Robert Leland. “An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations”. In : *SIAM J. Sci. Comput.* 16.2 (mar. 1995), p. 452–469 (cf. p. 24).

[28] T. Hoefler et M. Snir. “Generic Topology Mapping Strategies for Large-Scale Parallel Architectures”. In : *ICS*. 2011, p. 75–84 (cf. p. 14, 22, 26, 60, 73).

[29] T. Hoefler et al. “The Scalable Process Topology Interface of MPI 2.2”. In : *Concurrency and Computation : Practice and Experience* 23.4 (août 2010), p. 293–310 (cf. p. 60).

[30] T. 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 (cf. p. 24, 60).

[31] Chao Huang, Orion Lawlor et L. V. Kalé. “Adaptive MPI”. In : *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*. College Station, Texas, oct. 2003, p. 306–322 (cf. p. 62).

[32] J. Hursey, J. M. Squyres et T. Dontje. “Locality-Aware Parallel Process Mapping for Multi-core HPC Systems”. In : *2011 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2011, p. 527–531 (cf. p. 24).

[33] S. Ito, K. Goto et K. Ono. “Automatically Optimized Core Mapping to Subdomains of Domain Decomposition Method on Multicore Parallel Environments”. In : *Computer & Fluids* (avr. 2012) (cf. p. 22, 26).

[34] J. C. Hayes and M. L. Norman and R A. Fiedler and J. O. Bordner and P. S. Li and S. E. Clark and A. ud-Doula and M-M. McLow. “Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP”. In : *The Astrophysical Journal Supplement* 165.1 (2006), p. 188–228 (cf. p. 44).

[35] J. Dümmler and T. Rauber and G. Rünger. “Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters”. In : *Proceedings of the 2008 37th International Conference on Parallel Processing*. 2008, p. 141–148 (cf. p. 24, 26, 60).

[36] J. L. Träff. “Implementing the MPI Process Topology Mechanism”. In : *Supercomputing ‘02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Baltimore, Maryland : IEEE Computer Society Press, 2002, p. 1–14 (cf. p. 23, 24, 26).

[37] J. L. Whitt, G. Brook and M. Fahey. *Cray MPT : MPI on the Cray XT*. http://www.nccs.gov/wp-content/uploads/2011/03/MPT-OLCF11.pdf. 2011 (cf. p. 24).

[38] Pritish Jetley et al. “Massively parallel cosmological simulations with ChaNGa”. In : *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*. 2008 (cf. p. 62, 90).

[39] A. Kako et al. “Approximation Algorithms for the Weighted Independent Set Problem”. In : *LNCS*. 3787. SPRINGER-VERLAG, 2005, p. 341–350 (cf. p. 36).

[40] L. V. Kale et Sanjeev Krishnan. “Charm++ : Parallel Programming with Message-Driven Objects”. In : *Parallel Programming using C++*. Sous la dir. de Gregory V. Wilson et Paul Lu. MIT Press, 1996, p. 175–213 (cf. p. 61).

[41] Laxmikant V. Kale. “Programming Models at Exascale : Adaptive Runtime Systems, Incomplete Simple Languages, and Interoperability”. In : *The International Journal of High Performance Computing Applications* 23.4 (oct. 2009), p. 344–346 (cf. p. 61).

[42] Laxmikant V. Kale et S. Krishnan. “CHARM++ : A Portable Concurrent Object Oriented System Based on C++”. In : *Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 93*. ACM Press, sept. 1993, p. 91–108 (cf. p. 25, 60, 61).

[43] Laxmikant V. Kale et Gengbin Zheng. “Charm++ and AMPI : Adaptive Runtime Strategies via Migratable Objects”. In : *Advanced Computational Infrastructures for Parallel and Distributed Applications*. Sous la dir. de M. Parashar. Wiley-Interscience, 2009, p. 265–282 (cf. p. 60, 61).

[44] Laxmikant V. Kale et al. “Programming Petascale Applications with Charm++ and AMPI”. In : *Petascale Computing : Algorithms and Applications*. Sous la dir. de D. Bader. Chapman & Hall / CRC Press, 2008, p. 421–441 (cf. p. 61).

[45] Laxmikant Kale et al. *Charm++ for Productivity and Performance : A Submission to the 2011 HPC Class II Challenge*. Rapp. tech. 11-49. Parallel Programming Laboratory, nov. 2011 (cf. p. 61).

[46] Laxmikant Kale et al. *Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance A Submission to 2012 HPC Class II Challenge*. Rapp. tech. 12-47. Parallel Programming Laboratory, nov. 2012 (cf. p. 61).

[47] G. Karypis et V. Kumar. *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Rapp. tech. 1995 (cf. p. 22, 23, 44, 66).

[48] George Karypis et Vipin Kumar. “Multilevel Algorithms for Multi-constraint Graph Partitioning”. In : *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. SC ’98. San Jose, CA : IEEE Computer Society, 1998, p. 1–13 (cf. p. 24).

[49] Georges Karypis et Vipin Kumar. “Multilevel Partitioning Scheme for Irregular Graphs”. In : *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING* 48 (1998), p. 96–129 (cf. p. 24).

[50] M. Kneser. “Aufgabe 300”. In : *Jahresber. Deutsch. Math. -Verein 58* (1955) (cf. p. 36).

[51] T. Ma et al. “Process Distance-Aware Adaptive MPI Collective Communications”. In : *2011 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2011, p. 196–204 (cf. p. 24).

[52] Vikas Mehta. “LeanMD : A Charm++ framework for high performance molecular dynamics simulation on large parallel machines”. Mém.de mast. University of Illinois at Urbana-Champaign, 2004 (cf. p. 62, 79, 80).

[53] Celso L. Mendes et al. “Deploying a Large Petascale System : The Blue Waters Experience”. In : *Procedia Computer Science* 29 (2014). 2014 International Conference on Computational Science, p. 198–209 (cf. p. 4, 79).

[54] Harshitha Menon et Laxmikant Kalé. “A Distributed Dynamic Load Balancer for Iterative Applications”. In : *Proceedings of SC13 : International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado : ACM, 2013, 15 :1–15 :11 (cf. p. 61).

[55] H. Menon et al. “Thermal aware automated load balancing for HPC applications”. In : *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*. Sept. 2013, p. 1–8 (cf. p. 61).

[56] G. Mercier et J. 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, sept. 2009, p. 104–115 (cf. p. 23, 26, 28).

[57] G. Mercier et E. Jeannot. “Improving MPI Applications Performance on Multicore Clusters with Rank Reordering”. In : *EuroMPI*. T. 6960. Lecture Notes in Computer Science. Santorini, Greece : Springer, sept. 2011, p. 39–49 (cf. p. 60).

[58] S. Micali et V. V. Vazirani. “An $O(\sqrt{(V)}E)$ algorithm for finding a maximum matching in general graphs.” In : *Proc. 21st Ann IEEE Symp. Foundations of Computer Science*. 1980, p. 17–27 (cf. p. 69).

[59] Argonne National Laboratory. *MPICH2*. http ://www.mcs.anl.gov/mpi/mpich2. 2004 (cf. p. 24).

[60] National Institute for Computational Sciences. *MPI Tips on Cray XT5*. http://www.nics.tennessee.edu/user-support/mpi-tips-for-cray-xt5 (cf. p. 24).

[61] Mark Nelson et al. “NAMD—a Parallel, Object-Oriented Molecular Dynamics Program”. In : *Intl. J. Supercomput. Applics. High Performance Computing* 10.4 (Winter 1996), p. 251–268 (cf. p. 62).

[62] Parallel Programing Laboratory, UIUC. *OpenAtom*. http://charm.cs.illinois.edu/OpenAtom/ (cf. p. 90).

[63] François Pellegrini et Jean Roman. *Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping*. Rapp. tech. TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I, 1996 (cf. p. 24).

[64] Laercio L Pilla et al. “Asymptotically Optimal Load Balancing for Hierarchical Multi-Core Systems”. In : *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*. IEEE. 2012, p. 236–243 (cf. p. 61, 66).

[65] Laércio L Pilla et al. “A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems”. In : *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE. 2012, p. 118–127 (cf. p. 61, 72).

[66] Bertrand Putigny, Brice Goglin et Denis Barthou. “A Benchmark-based Performance Model for Memory-bound HPC Applications”. In : *International Conference on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italie : IEEE, juil. 2014 (cf. p. 5).

[67] M. J. Rashti et al. “Multi-core and Network Aware MPI Topology Functions”. In : *EuroMPI*. 2011, p. 50–60 (cf. p. 22, 26).

[68] Eduardo R. Rodrigues et al. “A Comparative Analysis of Load Balancing Algorithms Applied to a Weather Forecast Model”. In : *Proceedings of 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Itaipava, Brazil, 2010 (cf. p. 61).

[69] E. Rodrigues et al. “Multicore Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications”. In : *Proceedings of the IEEE Symp. on Comp. and Comm.* Juil. 2009, p. 811–817 (cf. p. 23, 26).

[70] ArnoldL. Rosenberg. “Issues in the study of graph embeddings”. English. In : *Graphtheoretic Concepts in Computer Science*. Sous la dir. d’Hartmut Noltemeier. T. 100. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1981, p. 150–176 (cf. p. 22).

[71] 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 (cf. p. 23, 26).

[72] B. E. Smith et B. Bode. “Performance Effects of Node Mappings on the IBM BlueGene/L Machine”. In : *Euro-Par*. 2005, p. 1005–1013 (cf. p. 22, 26).

[73] Quinn O. Snell, Armin R. Mikler et John L. Gustafson. “NetPIPE : A Network Protocol Independent Performance Evaluator”. In : *in IASTED International Conference on Intelligent Information Management and Systems*. 1996 (cf. p. 55).

[74] 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 (cf. p. 22, 26).

[75] J.M. Tendler et al. “POWER4 system microarchitecture”. In : *IBM Journal of Research and Development* 46.1 (jan. 2002), p. 5–25 (cf. p. 1).

[76] François Trahay et al. “EZTrace : a generic framework for performance analysis”. Anglais. In : *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Poster Session. Newport Beach, CA, États-Unis, mai 2011 (cf. p. 7, 32).

[77] UPC Consortium. “UPC Language Specifications, v1.2”. In : *Lawrence Berkeley National Lab Tech Report LBNL-59208*. 2005 (cf. p. 25).

[78] Vishwanath Venkatesan et al. “Optimized Process Placement for Collective I/O Operations”. In : *EuroMPI*. Lecture Notes in Computer Science. to appear. Madrid, Spain : Springer, sept. 2013 (cf. p. 24, 26).

[79] H. Yu, I-H. Chung et J. E. Moreira. “Blue Gene System Software - Topology Mapping for Blue Gene/L Supercomputer”. In : *SC*. 2006, p. 116 (cf. p. 22, 26).

[80] Hao Yu, I-Hsin Chung et Jose Moreira. “Topology mapping for Blue Gene/L supercomputer”. In : *SC'06*. Tampa, Florida : ACM, 2006, p. 116 (cf. p. 60).

[81] J. Zhai et al. “FACT : Fast Communication Trace Collection for Parallel Applications Through Program Slicing”. In : *SC*. 2009 (cf. p. 32).

[82] J. Zhang et al. “Process Mapping for MPI Collective Communications”. In : *Euro-Par*. 2009, p. 81–92 (cf. p. 24).

[83] Gengbin Zheng. “Achieving high performance on extremely large parallel machines : performance prediction and load balancing”. Thèse de doct. Department of Computer Science, University of Illinois at Urbana-Champaign, 2005 (cf. p. 61).

[84] Gengbin Zheng et al. “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers”. In : *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*. San Diego, California, USA, sept. 2010 (cf. p. 61, 72).

[85] Gengbin Zheng et al. “Periodic Hierarchical Load Balancing for Large Supercomputers”. In : *International Journal of High Performance Computing Applications (IJHPCA)* (mar. 2011) (cf. p. 61, 72).

[86] H. Zhu et al. “Hierarchical Collectives in MPICH2”. In : *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Espoo, Finland : Springer-Verlag, 2009, p. 325–326 (cf. p. 24).

Liste des publications

Nos publications suivent toutes un ordre alphabétique par nom d'auteur.

Journal International avec Comité

[A1] Emmanuel Jeannot, Guillaume Mercier et François Tessier. "Process Placement in Multicore Clusters : Algorithmic Issues and Practical Techniques". Anglais. In : *IEEE Transactions on Parallel and Distributed Systems* (mai 2013).

Conférence Internationale avec Comité

[B1] Emmanuel Jeannot et al. "Communication and Topology-aware Load Balancing in Charm++ with TreeMatch". Anglais. In : *IEEE Cluster 2013*. Indianapolis, États-Unis : IEEE, sept. 2013 (cf. p. 60).

Conférence Nationale avec Comité

[C1] Emmanuel Jeannot, Guillaume Mercier et François Tessier. "TreeMatch : Un algorithme de placement de processus sur architectures multicœurs". Français. In : *RenPAR - 21e Rencontres Francophones du Parallélisme*. Grenoble, France, jan. 2013.

Communications sans actes

[D1] François Tessier. “Communication-aware load balancing with TreeMatch in Charm++”. In : Presented at the 9th workshop of the Joint Laboratory for Petascale Computing, Lyon, juin 2013.

[D2] François Tessier. “Distributed communication-aware load balancing with TreeMatch in Charm++”. In : Presented at the 11th workshop of the Joint Laboratory for Petascale Computing, Sophia-Antipolis, juin 2014.

[D3] François Tessier. “Distributed communication-aware load balancing with TreeMatch in Charm++”. In : Presented at the 9th Scheduling for Large Scale Systems Workshop, Lyon, juil. 2014.

[D4] François Tessier. “Load balacing and affinities between processes with TreeMatch in Charm++ : preliminary results and prospects”. In : Presented at the 7th workshop of the Joint Laboratory for Petascale Computing, Rennes, juin 2012.

[D5] François Tessier. “Processes placement on multicore. Dynamic load balancing in Charm++”. In : Presented at the 10th Annual Charm++ Workshop, Urbana-Champaign, IL, mai 2012.